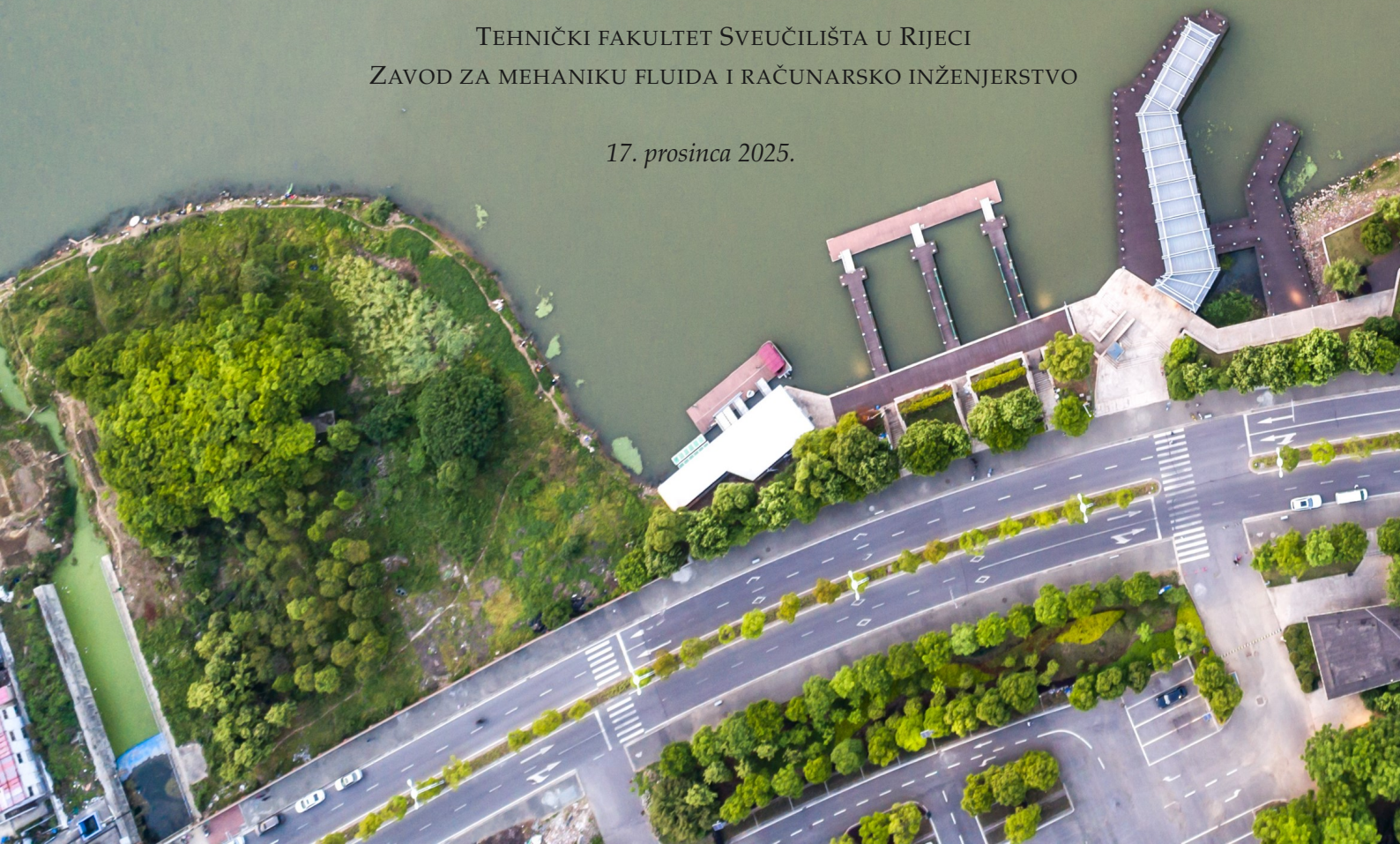
Računarsko inženjerstvo

uz programski jezik

Python

Tehnički fakultet Sveučilišta u Rijeci
Zavod za mehaniku fluida i računarsko inženjerstvo

17. prosinca 2025.



Zavod za mehaniku fluida i računarsko inženjerstvo,
Tehnički fakultet Sveučilišta u Rijeci

http://sim.riteh.hr/

Stefan Ivić
Jerko Škifić
Siniša Družeta
Bojan Crnković
Miran Tuhtan
Luka Grbčić
Ivana Lučin
Marko Čavrak

17. prosinca 2025.

Sadržaj

1. Osnove Pythona

1.1 Uvod

Stefan Ivić
Miran Tuhtan

Zašto Python? Postoje mnogi razlozi zašto odabrati Python kao inženjerski alat:

- Odličan za početnike, ali istodobno moćan za stručnjake
- Skalabilan, pogodan za velike projekte, kao i one male
- Omogućuje brz razvoj
- Platformski nezavisan (eng. *Cross-platform*), omogućuje razvoj i izvršavanje koda na gotovo svim platformama i arhitekturama
- Ugradiv u kodove pisane u drugim programskim jezicima
- Proširiv sa drugim programskim jezicima
- Objektno orjentiran
- Uredna i elegantna sintaksa
- Stabilan
- Bogatstvo dostupnih paketa i biblioteka
- Specijalizirani paketi za numeriku, statistiku, obradu podataka i vizualizacije
- Otvoreni besplatni kod koji održava velika zajednica programera i znanstvenika

1.1.1 Python 2 ili Python 3

Python 2 je stara i popularna verzija Pythona (objavljena 2000. godine), za koju službena podrška prestaje 2020. godine. Python 3 se pojavio 2008. godine i donio je neke novosti koje nisu bile kompatibilne s prethodnom verzijom, što je izazvalo otpor dijela korisnika koji su odbijali prijeći na novu verziju.

S vremenom se otpor smanjivao te je danas preporuka za nove korisnike i/ili za nove projekte da koriste Python 3, koji je u stalnom razvoju. U vrijeme pisanja ove knjige, najnovija verzija Pythona je 3.6.3.

1.1.2 Instalacija

Instalacija na Linux platformi

Gotovo sve Linux distribucije sadrže Python pakete u osnovnim repozitorijima, a većina distribucija ima inicijalno instaliran Python.

Na openSUSE distribuciji, Python se može instalirati jednostavnom naredbom:

```
user@machine:~> sudo zypper in python
```

Nakon instalacije Pythona, možemo provjeriti koja je putanja python izvršne datoteke

```
user@machine:~> type python3
python3 is /usr/bin/python3
```

Instalacija na Windows platformi

Za instalaciju Pythona na Windows platformi, jedan od načina instalacije je preuzimanje instalacijske datoteke sa http://www.python.org. Nakon preuzimanja, pokretanjem

instalacijske datoteke započinje instalacijski program koji omogućuje uređivanje postavki Python instalacije.

Uobičajena instalacija Pythona je zajedno s dodatnim paketima preko neke od dostupnih Python distribucija (vidi poglavlje 1.1.3).

1.1.3 Dodatni paketi

Velika snaga Pythona je dostupnost velikog broja paketa raznih namjena. Za inženjerske potrebe, potrebni su dodatni paketi koji omogućavaju matrični račun, numeričke metode, statističku obradu podataka, vizualizacije, izradu animacija i druge specijalizirane mogućnosti.

Osnovni paketi, koji će biti korišteni u ovoj knjizi, su:

- Numpy (`http://www.numpy.org/`),
- SciPy (`http://www.scipy.org`),
- matplotlib (`http://matplotlib.org/`) i
- IPython (`http://ipython.org/`).

Za instalaciju dodatnih Python paketa postoje alati za upravljanje paketima. Najpoznatiji su `pip`, koji dolazi predinstaliran u novijim verzijama Pythona, te `easy_install`.

Dodatni paketi na Linux platformi

Za instalaciju dodatnih paketa na openSUSE distribuciji pokrenite naredbu:

```
user@machine:~> sudo zypper in python-numpy python-scipy
python-matplotlib IPython
```

Dodatni paketi na Windows platformi

Python distribucije su objedinjene instalacije pythona, raznih dodatnih paketa te često i tekstualnih editora ili razvoijnih okruženja. Distribucije su većinom ciljane za Windows platformu zbog nepraktične višestruke instalacije svakog pojedinog softvera.

Neke od najpopularnijih Python distribucija su:

- ActivePython (`http://www.activestate.com/activepython`)
- Anaconda (`https://www.continuum.io/downloads`)
- Entought Python (`https://www.enthought.com/products/canopy/`)
- winpython (`https://winpython.github.io/`)

1.1.4 Instalacija pomoću upravitelja paketa

Python upravitelji paketa omogućuju jednostavno instalaciju, osvježavanje ili deinstalaciju raznih Python paketa (modula).

Standardno su u upotrebi tri upravitelja Python paketa:

- pip
- conda
- setuptools

Primjer instalacije paketa na Linux platformi pomoću upravitelja setuptools s naredbom `easy_install`:

```
user@machine:~> sudo easy_install ime_paketa
```

Primjer instalacije paketa na Windows platformi s naredbom `pip`:

```
C:\> pip install ime_paketa
```

Upravitelj conda je dostupan samo u Anaconda distribuciji, a naredba `conda` koristi se na sličan način kao i `pip`.

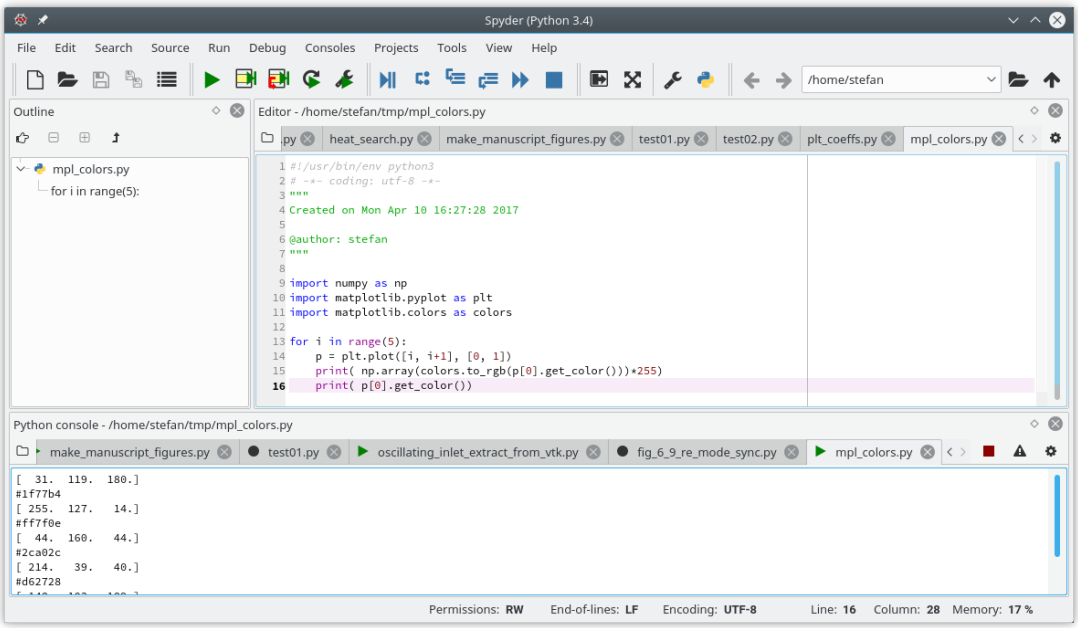
1.1.5 Editori i razvojna okruženja

Python pokreće kod iz običnih tekstualnih datoteka koje je moguće uređivati u mnogim tekstualnim editorima.

Razvojna okruženja (eng. *Integrated Development Environment*, *IDE*), osim što omogućavaju uređivanje koda, sadrže niza alata za lakši i brži razvoj kao što su jednostavno pokretanje koda iz IDE-a, debugiranje koda, praćenje ispisa i dr.

Spyder

Spyder (https://pythonhosted.org/spyder/) je interaktivno razvojno okruženje za Python namjenjeno za znastvenu i inženjersku primjenu. Spyder je dostupan za većinu platformi.



Slika 1.1 Spyder razvojno okruženje

PyDev

PyDev (http://pydev.org/) je dodatak za Eclipse (http://www.eclipse.org/) razvojno sučelje koji nudi mogućnosti za rad sa Pythonom.

Kate

Kate (http://kate-editor.org/) je moćan tekstualni editor, iako radi i na Windows platformi, primarno je namjenjen za Linux platformu.

Na openSUSE-u, Kate se može instalirati naredbom:

```
user@machine:~> sudo zypper in kate
```

VS Code

VS Code (https://code.visualstudio.com/) je generalni tekstualni editor za sve platforme (Windows, Mac, Linux), s ugrađenim debuggerom i version controlom (git). Za iskorištavanje punog potencijala editora u svrhu pisanja Python koda, potrebno je instalirati dodatak za Python jezik.

1.2 Python ljuska i .py skripte

Stefan Ivić
Miran Tuhtan

Python je interpretirani jezik, a jednostavni Python izrazi mogu se izvoditi u interaktivnom programskom okruženju pod nazivom ljuska (eng. *shell*).

1.2.1 Pokretanje Pythona

Za otvaranje Python ljuske na Linux platformi dovoljno je pokrenuti Python naredbu u terminalu/konzoli. Na Windows platformi moguće je upisati Python naredbu u konzolu ili pokrenuti IDLE.

Nakon pokretanja Pythona prikazuju se osnovne informacije o Pythonu (verzija) te tzv. prompt koji je simboliziran sa tri strelice u desno (>>>) i koji označava da interpreter čeka unos koda.

```
$ python
Python 3.6.3 | packaged by conda-forge | (default, Oct  5 2017, 14:07:33)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-15)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

1.2.2 Python ljuska kao kalkulator

Python ljuska odmah izvršava unešene naredbe nakon pritiska na tipku *Enter* (*Return*). Najjednostavniji Python izrazi su matematičke operacije pomoću kojih možemo koristiti Python ljusku kao kalkulator. Izračun te ispis rezultata matematičkih izrazi događa se u novom retku terminala ili IDLE-a:

```
>>> 1+1
2
>>>
```

Matematičke operacije, naravno, možemo međusobno kombinirati i pri tome koristiti zagrade za definiranje redoslijeda operacija.

```
>>> 11 / 4
2.75
>>> 11 // 4
2
>>> (2 * 10 - 5) / (4 + 1)
3.0
>>>
```

Ljuska može raditi i bazične operacije sa tekstom tj. nizovima znakova (*string*).

```
>>> "Kratki " + "probni " + "tekst."    # ovo je komentar
'Kratki probni tekst.'
>>>
```

Kao što vidimo, moguće je pisati komentare koji se ne izvršavaju, odnosno sve nakon znaka # smatra se komentarom.

1.2.3 IPython ljuska

IPython (https://ipython.org/) je interaktivna Python ljuska koja nudi nekoliko prednosti u odnosu na Python ljusku:

- povijest unosa
- tab-completition
- "magične" komande
- multi-line editiranje
- syntax highlighting

IPython ljuska dolazi predinstalirana u Anaconda distribuciji. Pokretanje IPython ljuske izvršava se naredbom `ipython` u terminalu:

```
$ ipython
Python 3.6.3 | packaged by conda-forge | (default, Oct  5 2017, 14:07:33)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

1.2.4 Python skripte

Python skripte su tekstualne datoteke s ekstenzijom .py koje sadrže Python kod. Izvršavanjem .py skripte, zapravo se pokreće linija po linija kao da ih se unosi u Python ljusku. Ovo je očita prednost, pogotovo prilikom pokretanja više od nekoliko naredbi.

Datoteke s ekstenzijom .py, osim kao skripte, koriste se i za definiranje vlastitih modula, što je detaljnije opisano u poglavlju 1.2.5.

Osnovni način pokretanja python skripte je pokretanje Pythona sa specificiranom .py datotekom:

```
C:\> python ime_skripte.py
```

Python će se pokrenuti i izvršiti kod zapisan u skripti. Nakon završetka izvođenja skripte, Python se gasi.

Ako je prva linija skripte:

```
#!/usr/bin/env python3
```

tada se na Unix sustavima može izvršavati u terminalu. Za pokretanje skripte u terminalu nužno je da skripta koristi Unix oznaku za kraj reda te da ima dozvole za izvršavanje:

```
user@machine:~$ chmod +x imeskripte.py
```

Ako je .py datoteka izvršna, tada se može direktno izvršavati:

```
user@machine:~$ ./imeskripte.py
```

Na Windows sustavima su datoteke s ekstenzijom .py kod intalacije Pythona pridružene programu python.exe. Python skripta se pokreće dvostrukim lijevim klikom miša na skriptu.

1.2.5 Moduli

Modul je Python objekt, definiran u određenoj datoteci, koji sadrži definicije varijabli, funkcija i klasa. Obično su varijable, funkcije i klase određenog modula specijalizirane namjene, pa se module može interpretirati kao grupiranje koda i najčešće je taj kod slične namjene.

Npr. modul `math` (obrađen u poglavlju 1.12.1) sadrži varijable `pi` i `e` koje predstavljaju matematičke konstante, te niz matematičkih funkcija kao što su `sqrt`, `log`, `pow`, `exp`, `sin`, `cos` itd.

Da bi se objekti određenog modula mogli koristiti u kodu, potrebno je učitati (engl. *import*) modul ili učitati određeni objekt iz modula.

Učitavanje modula vrši se pomoću naredbe `import` čija je osnovna sintaksa:

```
import module_name
```

gdje je `module_name` ime modula.

Nakon što je modul učitan, može se koristiti objekte modula (varijable, funkcije i klase) tako da se unese ime modula, točka te ime objekta koji se želi koristiti.

```
import module1
module1.object_name
```

Učitanom modulu se može dodijeliti novo ime. To se može napraviti pomoću ključne riječi `as`:

```
import module_name as new_name
```

nakon čega se modul može koristiti pomoću novog imena.

Moduli često sadržavaju velik broj objekata, a njihovim učitavanjem bespotrebno se troše memorijski i procesorski resursi. Kako bi se izbjeglo učitavanje cijelog modula, moguće je iz određenog modula učitati samo određene objekte:

```
from module_name import object_name
```

Ako se direktno učita objekte iz modula, tada ih se koristiti samo preko imena objekta. Kao i sam modul, moguće je učitati objekt nekog modula pod novim imenom:

```
from module_name import object_name as object_new_name
```

Python kod 1.1 Korištenje objekata modula

```
# učitaj sys modul
import sys
print(sys.platform)     # ispiši platformu (operativni sustav)
print(sys.version)      # ispiši verziju Pythona

# učitaj math modul kao m
import math as m
print(m.pi)             # ispiši vrijednost broja pi
print(m.sin(m.pi/2.0))  # ispiši vrijednost sinusa kuta pi/2

# iz modula datetime učitaj objekt date
from datetime import date
print(date.today())     # ispiši današnji datum

# iz modula random učitaj funkciju random kao nasumicni
from random import random as nasumicni
```

```
print(nasumicni())      # ispiši nasumicni broj

# učitaj tri objekta iz modula string
from string import ascii_uppercase, ascii_lowercase, digits
print(ascii_uppercase)  # ispis svih velikih slova
print(ascii_lowercase)  # ispis svih velikih slova
print(digits)           # ispis svih znamenaka
```

```
win32
Python 3.6.2 |Continuum Analytics, Inc.| (default, Jul 20 2017, 12:30:02)
[MSC v.1900 64 bit (AMD64)]
3.141592653589793
1.0
2017-07-14
0.24526187751076556
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789
```

1.3 Tipovi podataka

Stefan Ivić
Miran Tuhtan
Bojan Crnković

Python ima mnogo ugrađenih tipova podataka. Svaka vrijednost pripada nekom tipu podataka, koji joj se automatski dodjeljuje prilikom prve upotrebe varijable. Tip varijable nije potrebno eksplicitno deklarirati, već Python sam zaključuje o kojem tipu podataka se radi.

Osnovni tipovi podataka u Pythonu su:

- Boolean
- Number
- String
- List
- Tuple
- Set
- Dictionary

Tip podataka određene varijable može se dobiti funkcijom `type`:

```
>>> i = 10
>>> type(i)
<class 'int'>
>>> f = 3.14
>>> type(f)
<class 'float'>
>>> r = 'Rijeka'
>>> type(r)
<class 'str'>
>>> s = {1, 3, 5, 7}
>>> type(s)
<class 'set'>
>>> t = (2, 4, 6)
>>> type(t)
<class 'tuple'>
```

1.3.1 Boolean

Boolean je tip podataka koji može biti ili istinit (*true*) ili lažan (*false*). Python ima dvije konstante pomoću kojih je moguće direktno dodjeljivanje boolean varijabli, a to su `True` i `False`. (Napomena: Python je osjetljiv na velika i mala slova (eng. *case sensitive*), i bitno je za istaknuti da se navedene konstante pišu velikim početnim slovom.)

Osim direktnog dodjeljivanja, moguće je provjeravati istinitost raznih izraza:

```
>>> a = True
>>> a
True
>>> 5 < 7
True
>>> 15 > 20
False
```

1.3.2 Brojevi

U Pythonu je moguće zadavati tri vrste brojeva.

- Integer (*int*) su cijeli brojevi, npr. 5, 11, 35
- Floating point (*float*) su realni brojevi, npr. 2.73, 8.14, 5.223
- Complex su kompleksni brojevi koji se sastoje od realnog i imaginarnog dijela (koji se obilježava slovom j), npr. $3 + 5j$, $2.91 - 4.84j$, $9 + 1.34j$

```
>>> type(5)
<class 'int'>
>>> type(2.73)
<class 'float'>
>>> type(3+5j)
<class 'complex'>
```

Za razliku od nekih drugih programskih jezika (kao i Pythona 2), u Pythonu ne postoje tzv. *long* integeri, već se *int* ponaša kao *long* u tim programskim jezicima, odnosno integeri mogu biti proizvoljno veliki.

Sve tri vrste brojeva su međusobno kompatibilne i nije potrebno mijenjati/prilagođavati njihov tip kako bismo mogli izvoditi matematičke operacije između različitih tipova brojeva:

```
>>> a = 11
>>> b = 13.57
>>> c = 5+7j
>>> a + b
24.57
>>> b + c
(18.57+7j)
>>> a - c
(6-7j)
```

Ukoliko želimo ručno mijenjati vrstu brojeva, to možemo funkcijama `int()`, `float()` i `complex()`. Kompleksne brojeve nije moguće pretvoriti u *float* ni *int*.

```
>>> float(a)
11.0
>>> complex(a)
(11+0j)
>>> int(b)
13
>>> float(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't convert complex to float
```

1.3.3 Stringovi

Stringovi su definirani kao slijed znakova između jednostrukih (') ili dvostrukih (") navodnika. U Pythonu ne postoji poseban *char* tip podataka za znakove, već se i samo jedan znak smatra stringom.

Stringovi su nepromjenjivi tip podataka, odnosno jednom definirani string nije moguće mijenjati, a svaka promjena na stringu zapravo kreira kopiju izvornog stringa. Stringove je moguće spajati (eng. *concatenation*) znakom plus (+) te ponavljati sa zvjezdicom (*).

```
>>> 'ana' + 'marija'
'anamarija'
>>> 'š' + 3 * 'ti'
'štititi'
```

Iako se za ponavljanje stringova koristi sintaksa `integer * string`, stringove i brojeve nije moguće miješati niti se nad stringovima mogu vršiti klasične matematičke operacije:

```
>>> 2 + '2'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Indexing i slicing

Python ima *zero-based* indeksiranje (kao i programski jezici C, Java i drugi), što znači da prvi element niza ima index 0 (nula). Takav način indeksiranja traži privikavanje, pogotovo kod programera koji su do sada koristili programske jezike kod kojih prvi element ima index 1 (recimo, Fortran ili Matlab), kako bi se izbjegle tzv. *off by one* greške.

Za dohvat pojedinog znaka unutar stringa koristimo operator `[]` unutar kojeg navodimo indeks elementa kojeg želimo. Moguće je i indeksiranje u suprotnom smjeru, gdje posljednji element ima indeks -1, a svaki prijašnji element za jedan manje.

```
>>> p = 'Python rocks'
>>> p[1]
'y'
>>> p[-3]
'c'
```

Ukoliko želimo dohvatiti niz uzastopnih elemenata, to je moguće sa slicing sintaksom:

```
ime_varijable[start:stop:step]
```

gdje je `start` početak intervala, `stop` je gornja granica (koja nije uključena u interval), a `step` je korak. Ukoliko neka od te tri vrijednosti nije eksplicitno definirana, uzima se njena zadana vrijednost - za start je to početak stringa (indeks 0), za stop je kraj stringa, a za step je vrijednost 1 (uzima se svaki element između starta i stopa, bez preskakanja).

```
>>> p[2:5]
'tho'
>>> p[:4]
'Pyth'
>>> p[-3:]
'cks'
>>> p[::2]
'Pto ok'
```

1.3.4 Liste

Liste su tip podataka koji sadrži više vrijednosti, omeđenih uglatim zagradama i odvojenih zarezima. Za razliku od nekih drugih programskih jezika, liste u Pythonu

mogu sadržavati podatke različitih tipova, ali preporuča se koristiti liste kao spremnik homogenih podataka.

Za dohvat pojedinog elementa, ili niza elemenata, koristi se ista sintaksa kao i kod stringova.

```
>>> prazna_lista = []
>>> prazna_lista
[]
>>> lista_brojeva = [2, 3, 7, 8]
>>> lista_brojeva[-1]
8
>>> lista_slova = ['a', 'b', 'd']
>>> lista_slova[0]
'a'
>>> lista_brojeva[1:]
[3, 7, 8]
```

Za provjeru duljine liste, odnosno broja elemenata koje lista sadrži, koristi se funkcija `len()`. Naredba `append` koristi se za dodavanje novog elementa na posljednje mjesto liste. Ukoliko želimo dodati novi element na točno određeno mjesto unutar liste, to možemo naredbom `insert`, unutar koje definiramo indeks na koji želimo dodati element te element kojeg dodajemo.

```
>>> len(lista_brojeva)
4
>>> lista_slova.append('z')
>>> lista_slova
['a', 'b', 'd', 'z']
>>> lista_slova.insert(2, 'c')
>>> lista_slova
['a', 'b', 'c', 'd', 'z']
```

Operatori `+` i `*` se mogu upotrebljavati na analogni način kao i kod stringova. Operator `+` spaja dvije liste, a `*` ponavlja sve elemente liste određeni broj puta.

```
>>> lista_brojeva + lista_slova
[2, 3, 7, 8, 'a', 'b', 'c', 'z']
>>> 2 * lista_slova
['a', 'b', 'c', 'z', 'a', 'b', 'c', 'z']
```

Liste su, za razliku od stringova, promjenjivi tip podataka, što znači da je moguće mijenjati pojedine elemente liste koristeći sintaksu za dohvat elemenata.

```
>>> s = 'Pzthon'
>>> s[1] = 'y'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> lista_brojeva[2] = 999
>>> lista_brojeva
[2, 3, 999, 8]
>>> lista_slova[:3] = ['w', 'x', 'y']
>>> lista_slova
['w', 'x', 'y', 'z']
```

Range

Funkcija `range` omogućava lakše stvaranje aritmetičkog niza brojeva. Sintaksa funkcije `range` je

```
range(start, stop, step)
```

gdje je `start` prvi element, `stop` je gornja granica (nije uključena u niz), a `step` je korak aritmetičkog niza. Jedino je gornju granicu obavezno definirati (kako ne bi nastao beskonačni niz brojeva). Ako prvi element nije definiran, niz će krenuti od nule, a ako ne definiramo korak, niz će se sastojati od uzastopnih brojeva.

```
>>> list(range(4))
[0, 1, 2, 3]
>>> list(range(2, 7))
[2, 3, 4, 5, 6]
>>> list(range(0, 12, 3))
[0, 3, 6, 9]
```

1.3.5 Tuple

Tuple (hrv. n-torka) je tip podataka sličan listi, ali s tom razlikom da je *tuple* nepromjenjiv - nemoguće je dodavati ili brisati članove te nije moguće mijenjati vrijednosti postojećih članova. Tuple se sastoji od vrijednosti odvojenih zarezom omeđenih zagradama (moguće je zadati tuple i bez zagrada, ali one se preporučaju zbog čitljivosti koda). Najčešće se koristi kao skup heterogenih, međusobno povezanih podataka.

```
>>> ntorka = ('Marko', 27, 182)
>>> ntorka
('Marko', 27, 182)
>> ntorka[0]
'Marko'
>>> ntorka[1:]
(27, 182)
>>> ntorka[1] = 38
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Osim dohvaćanja pojedinih elemenata tuplea korištenjem uglatih zagrada, često se koristi tzv. *tuple unpacking*, koji nam omogućava dodjeljivanje dijelova tuplea točno određenoj varijabli. Jedna od čestih primjena tuple unpackinga je kod zamjene dviju varijabli.

```
>>> ime, godine, visina = ntorka
>>> ime
'Marko'
>>> godine
27
>>> visina
182
>>> a = 5
>>> b = 10
>>> a, b = b, a
>>> a
10
>>> b
5
```

1.3.6 Skupovi

Objekti tipa `set` predstavljaju heterogenu kolekciju (skup) elemenata odvojenih zarezima i omeđenih vitičastim zagradama, koji nemaju poredak. Važno je napomenuti da su elementi skupa jednistveni (nema ponavljanja elemenata), ali skupovi ne zadržavaju poredak elemenata te nije moguće doći do pojedinog elementa koristeći indexing sintaksu kao kod lista, tupleova i stringova.

```
>>> slova = set('banana')
>>> slova
{'b', 'n', 'a'}
>>> slova[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
```

Provjera pripadnosti elementa skupu vrlo je brza (mnogo brža nego kod lista) pa su skupovi vrlo pogodni za takve operacije. Upotrebom skupova moguće je izvoditi uobičajene skupovne operacije poput presjeka (`&`), unije (`|`), razlike (`-`), simetrične razlike (`^`) i sl. Novi elementi dodaju se funkcijom `add`, a postojeći brišu funkcijom `remove`.

```
>>> mali = {1, 2, 3, 4}
>>> veliki = {3, 5, 7, 9}
>>> 3 in mali
True
>>> 6 in veliki
False
>>> mali & veliki
{3}
>>> mali | veliki
{1, 2, 3, 4, 5, 7, 9}
>>> mali - veliki
{1, 2, 4}
>>> mali ^ veliki
{1, 2, 4, 5, 7, 9}
>>> veliki.add(11)
>>> veliki.remove(3)
>>> veliki
{5, 7, 9, 11}
```

1.3.7 Dictionary

Dictionary je mapping tip podataka, koji nam omogućava preslikavanje (mapiranje) `keys` u `values`. `Keys` moraju biti nepromjenjivi tip podataka (stringovi, brojevi, tuple), dok `values` mogu biti podaci bilo kojeg tipa. `Key:value` parovi odvojeni su zarezima i omeđeni vitičastim zagradama.

```
>>> imena = {'a': ['Andrija', 'Anđelka'], 'j': ['Josip', 'Juraj']}
>>> imena.keys()
dict_keys(['a', 'j'])
>>> imena.values()
dict_values([['Andrija', 'Anđelka'], ['Josip', 'Juraj']])
```

Redoslijed elemenata unutar dictionarya nije sačuvan, a vrijednostima (values) pojedinog elementa (key) pristupamo sintaksom za dohvat elemenata, ali ne koristeći

indeks elementa (elementi u dictionaryju nemaju indeks) već ime elementa (keya) jer oni su jedinstveni unutar dictionarya (ne postoje dva elementa istog imena).

Ukoliko pokušamo dohvatiti vrijednosti elementa koji ne postoji, to će proizvesti grešku. Elegantnije rješenje, kada nismo sigurni postoji li željeni element, je korištenjem funkcije `get`, unutar koje definiramo element koji želimo te što za slučaj da tog elementa nema.

```
>>> imena['a']
['Andrija', 'Anđelka']
>>> imena['z']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'z'
>>> imena.get('z', 'Nema nikoga pod tim slovom')
'Nema nikoga pod tim slovom'
>>> imena.get('j', 'Nema nikoga pod tim slovom')
['Josip', 'Juraj']
```

Dodavanje novih elemenata u dictionary moguće je dodjeljivanjem vrijednosti novom keyu. Postojeće keyeve moguće je mijenjati na isti način.

```
>>> imena['m'] = ['Mate', 'Matilda']
>>> imena['j'] = ['Jelena', 'Janko']
>>> imena
{'a': ['Andrija', 'Anđelka'], 'j': ['Jelena', 'Janko'],
'm': ['Mate', 'Matilda']}
```

1.4 Operatori

Stefan Ivić
Miran Tuhtan

Operatori su osnovni dio Pythona i omogućuju operacije na varijablama (operandima). Najčešće se povezuju s algebarskim matematičkim operacijama, međutim operatori mogu biti i "nematematičke" naravi.

Python operatore možemo podjeliti prema namjeni:

- Aritmetički operatori
- Relacijski operatori ili operatori uspoređivanja
- Operatori dodjeljivanja
- Logički operatori
- Bitovni operatori
- Članski operatori
- Operatori identiteta

ili s obzirom na broj operanada:

- Unarni operatori (jedan operand)
- Binarni operatori (dva operanada)

1.4.1 Aritmetički operatori

Aritmetički operatori omogućuju osnovne matematičke operacije na brojevima. Rezultat aritmetičkog operatora je uvijek broj tj. varijabla tipa `float` ili `int`, ovisno o tipovima operanada. Bitno je napomenuti da je tip rezultata aritmetičkih operacija uvijek višeg tipa operanada.

Tablica 1.1 Aritmetički operatori

Operator	Opis	Primjer za `x=9` i `y=2`
`+`	Zbrajanje	`x+y` daje `11`
`-`	Oduzimanje	`x-y` daje `7`
`*`	Množenje	`x*y` daje `18`
`/`	Dijeljenje	`x/y` daje `4.5`
`%`	Ostatak dijeljenja	`x%y` daje `1`
`//`	Cjelobrojno dijeljenje	`x//y` daje `4`
`**`	Potenciranje	`x**y` daje `81`

1.4.2 Relacijski operatori (operatori uspoređivanja)

Relacijski operatori omogućavaju uspoređivanje vrijednosti varijabli. Rezultat relacijskih operatora je varijabla tipa boolean i može poprimiti vrijednosti `True` ili `False`.

Tablica 1.2 Relacijski operatori

Operator	Opis	Primjer za `x=9` i `y=2`
`==`	Jednakost	`x==y` daje `False`
`!=`	Nejednakost, različitost	`x!=y` daje `True`
`>`	Veće	`x>y` daje `True`
`>=`	Veće ili jednako	`x>=y` daje `True`
`<`	Manje	`x<y` daje `False`
`<=`	Manje ili jednako	`x<=y` daje `False`

1.4.3 Operatori dodjeljivanja

Operator dodjeljivanja služi za dodjeljivanje vrijednosti varijabli. Postoje proširenja kojima se može kombinirati aritmetičke operatore i operator dodjeljivanja.

Tablica 1.3 Operatori dodjeljivanja

Operator	Opis	Primjer za `x=9` i `y=2`
`=`	Dodjeljivanje	`x=y` dodjeljuje vrijednost `2.0` varijabli `x`, rezultat: `x=2`
`+=`	Zbrajanje i dodjeljivanje	`x+=y` je isto što i `x=x+y`, rezultat: `x=11`
`-=`	Oduzimanje i dodjeljivanje	`x-=y` je isto što i `x=x-y`, rezultat: `x=7`
`*=`	Množenje i dodjeljivanje	`x*=y` je isto što i `x=x*y`, rezultat: `x=18`
`/=`	Dijeljenje i dodjeljivanje	`x/=y` je isto što i `x=x/y`, rezultat: `x=4.5`
`%=`	Ostatak dijeljenja i dodjeljivanje	`x%=y` je isto što i `x=x%y`, rezultat: `x=1`
`//=`	Cjelobrojno dijeljenje i dodjeljivanje	`x//=y` je isto što i `x=x//y`, rezultat: `x=4`
`**=`	Potenciranje i dodjeljivanje	`x**=y` je isto što i `x=x**y`, rezultat: `x=81`

1.4.4 Logički operatori

Logički operatori `and`, `or` i `not` omogućavaju operacije na boolean varijablama. `and` i `or` su binarni operatori *logičko i* i *logičko ili* dok je `not` unarni operator *logičko ne*. Rezultat logičkih operatora je uvijek boolean varijabla.

Tablica 1.4 Logički operatori

`a`	`b`	`a and b`	`a or b`	`not b`
`True`	`True`	`True`	`True`	`False`
`True`	`False`	`False`	`True`	`True`
`False`	`True`	`False`	`True`	
`False`	`False`	`False`	`False`	

Python kod 1.2 Primjer kombiniranja operatora dodjeljivanja, operatora uspoređivanja i logičkih operatora

```
>>> a = 10
>>> b = 1
>>> c = 15
>>> a < b
False
>>> a > b and c > b
True
>>> a > c or not b > c
True
```

1.4.5 Članski operatori

Članski opratori služe za određivanje pripadnosti određenom skupu podataka (npr. listi). Osnovni članski operator je operator `in`, i može se proširiti s logičkim operatorom `not`, dajući `not in` operator. Rezultat članskih opratora je boolean varijabla koja daje saznanje je li zadani element dio sekvenvce.

```
>>> a = [1, 2, 3, 8, 6]
>>> 2 in a
True
>>> 3 not in a
False
>>> 5 not in a
True
```

1.4.6 Operatori identiteta

Operatori identita uspoređuju jesu li Python objekti zapisani na istom mjestu u memoriji. Postoje operatori `is` i `is not` koji uspoređuju `id` Python objekta. `id` Python objekta se može dohvatiti funkcijom `id`.

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
139967012502704
>>> id(b)
139967012502848
>>> a is b
False
>>> b = a
>>> id(b)
139967012502704
>>> a is b
True
```

1.5 Osnovne naredbe za unos i ispis podataka

Stefan Ivić
Miran Tuhtan

1.5.1 Ispis podataka

Iz Python koda moguće je ispisivati poruke u konzolu pomoću funkcije `print()`. Osnovna sintaksa za ispisivanje je

```
print(variable)
```

gdje je `variable` ima varijable čiju vrijednost želimo ispisati.

Moguće je ispisivati vrijednosti više varijabli, odrediti kako odvojiti varijable (parametar `sep`, standardna vrijednost je razmak), te što kada se dođe do kraja varijabli (parametar `end`, standardna vrijednost je novi red:

```
print(variable1, variable2, sep=' ', end='\n')
```

```
>>> print('Ivan', 'Goran', 'Tin')
Ivan Goran Tin
>>> print('Ivan', 'Goran', 'Tin', sep=' je stariji nego ')
Ivan je stariji nego Goran je stariji nego Tin
>>> imena = ['Ana', 'Marko', 'Filip']
>>> for ime in imena:
...     print(ime)
...
Ana
Marko
Filip
>>> for ime in imena:
...     print(ime, end='; ')
...
Ana; Marko; Filip;
```

Naprednije mogućnosti ispisa moguće je kontrolirati ubacivanjem vrijednosti varijabli u string operatorom `%` ili pomoću metode `format`, što je detaljnije obrađeno u poglavlju 1.10.

1.5.2 Unos podataka

Funkcija `input()` omogućava unošenje vrijednosti tipa `str` (string).

```
>>> a = input()
moj prvi unos
>>> print(a)
moj prvi unos
>>> print(type(a))
<class 'str'>
```

Dodatno, funkcija `input` može primiti poruku, u obliku stringa, koja se ispisuje na ekranu prilikom unosa.

```
>>> ime = input('Unesite svoje ime: ')
Unesite svoje ime: Ivo
```

```
>>> print(ime)
Ivo
```

Svaki korisnički unos je string, koji prije daljnjeg korištenja treba pretvoriti u željeni tip podataka.

```
>>> a = input('Unesite vasu visinu u metrima: ')
Unesite vasu visinu u metrima: 1.85
>>> b = input('Unesite vasu masu u kilogramima: ')
Unesite vasu masu u kilogramima: 79
>>> type(a), type(b)
(<class 'str'>, <class 'str'>)
>>> visina = float(a)
>>> masa = int(b)
>>> print('Vas BMI je:', masa / visina**2)
Vas BMI je: 23.08254200146092
```

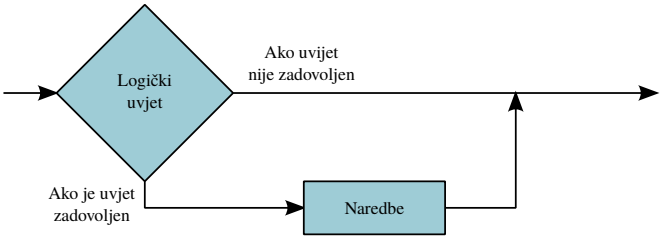
1.6 Uvjetna grananja

Stefan Ivić
Miran Tuhtan

Određene naredbe ili djelove koda moguće je izvršavati ili preskočiti ovisno o vrijednosti postavljenog logičkog uvjeta.

1.6.1 Naredba if

Slika 1.2 prikazuje jednostavno grananje u programskom kodu.



Slika 1.2
Dijagram toka `if` naredbe

Osnovna sintaksa `if` grananja je

```
if uvjet: naredba
```

Navedeni izraz uvjetuje izvršavanje koda `naredba` na temelju logičkog uvjeta `uvjet`. Zadane naredbe će se izvršiti samo ako je logički uvijet zadovoljen.

Python kod 1.3 Jednostavni primjer upotrebe `if` naredbe

```
a = int(input("a: "))
if a < 0: a = -a
print(a)
```

Pokretanjem Izvornog koda 1.3 i utipkavanjem pozitivnog broja ispiše se utipkani broj:

```
a: 5
5
```

Međutim, utipkavanjem negativnog broja zadovoljava se postavljeni logički uvjet i izvrši se operacija `a = -a` te se potom ispiše pozitivna vrijednost utipkanog broja:

```
a: -11
11
```

U slučaju kad postoji više naredbi uvjetovanih istim logičkim uvjetom, naredbe pišemo u uvučenim recima nakon `if` naredbe. Uobičajeno je da retke uvučemo sa četiri razmaka. Sve uvučene naredbe (do prvog sljedećeg retka koji je poravnat sa `if` naredbom) su uvjetovane.

Python kod 1.4 Pisanje složenijeg uvjetovanog grananja

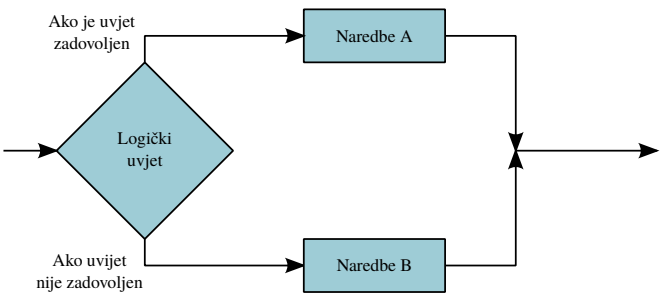
```
a = 0
b = int(input("b: "))
c = int(input("c: "))
```

```
if c < b:
    a = 2
    b = c - b
    c = c + 1

a = b + c
print(a, b, c)
```

```
b: -6
c: 4
-2 -6 4
```

1.6.2 Naredba else

Ova naredba omogućuje složeniji oblik grananja od naredbe `if`. Naredba ima dvije grane koje se mogu izvršiti u ovisnosti o uvjetu. Svaka grana/mogućnost omogućuje izvršavanje pripadajućih naredbi (Slika 1.3).



Slika 1.3
Dijagram toka `if-else` grananja

Naredba `else` koristi se u paru sa `if` i služi za grananje u slučaju da uvjet nije zadovoljen.

Sintaksa naredbe je:

```
if condition: command1
else: command2
```

Python kod 1.5 Primjer upotrebe `else` naredbe

```
a = int(input("a: "))
b = int(input("b: "))

if a < b:
    b = a
    a = 0
else:
    a = b
    b = 0

print(a, b)
```

```
a: 1
b: 8
0 1
```

```
a: 5
b: 3
3 0
```

1.6.3 Naredba elif

Za dodatna grananja uvjetovana novim logičkim uvjetima (Slika 1.4) služi naredba `elif`. Iako se sa ugniježdenim grananjem može postići isti rezultat, `elif` omogućuje jednostavnije i preglednije grananje algoritma.

Naredba `elif` omogućuje grananje u alogritmu samo ako svi prethodni uvjeti (`if` i `elif`) nisu zadovoljeni. Treba napomenuti da se u slučaju korištenja `if-elif-else` grananja uvijek izvrši samo jedan set uvjetovanih naredbi.

Iz same definicije dodatnog uvjetnog grananja, naredba `elif` uvijek dolazi u kombinaciji sa naredbom `if`, te se upotrebljava u sljedećoj osnovnoj sintaksi:

```
if condition1: command1
elif condition2: command2
elif condition3: command3
```

Kao i u slučaju upotrebe `else` naredbe, dodatno grananje se mora napisati u novom retku. Uobičajeno je da se naredbe koje slijede nakon uvjeta pišu u novom uvučenom retku.

Grananje se može napraviti i kombinacijom sve tri navedene naredbe:

```
if condition1: command1
elif condition2: command2
else: command2
```

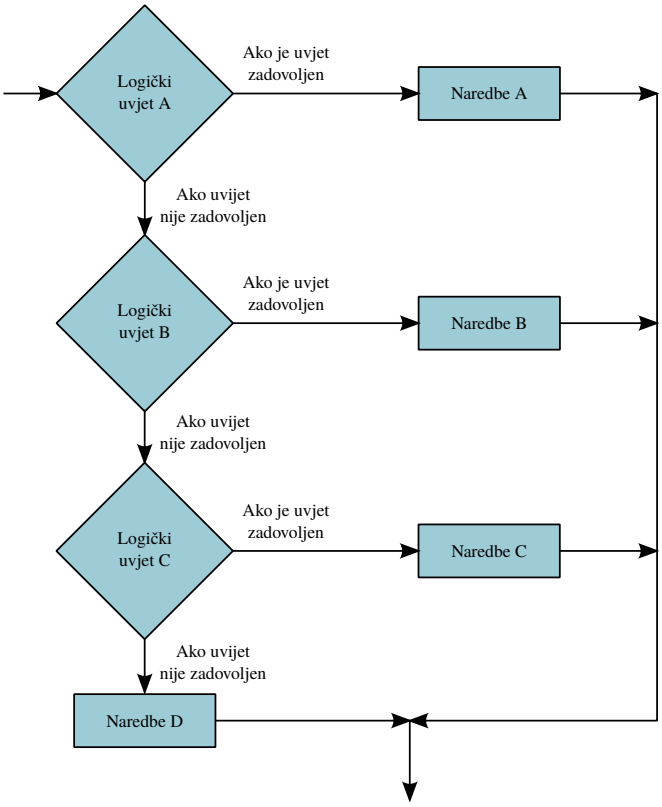
U Izvornom kodu 1.6 napisan je jednostavan primjer grananja koristeći navedene naredbe.

Python kod 1.6 Primjer kombinacije naredbi `if`, `elif` i `else`

```
a = int(input("a: "))

if a < 0:
    print('a je negativan')
elif a > 0:
    print('a je pozitivan')
else:
    print('a je nula')
```

Unosom pozitivnog broja izvršava se naredba koja je uvjetovana uvjetom pod `elif` naredbom:

```
a: 33
a je pozitivan
```



Slika 1.4
Dijagram toka za složeni uvjet grananja

1.7 Petlje

Stefan Ivić
Miran Tuhtan

1.7.1 Naredba for

Naredba `for` koristi se za prolazak kroz elemente nekog iterabilnog objekta (lista, tuple, string, itd.). Njezina osnovna sintaksa je

```
for element in iterable: commands
```

Naredbe `commands` izvršavaju se jednom za svaki element iz niza `iterable`. Objekt `element` je iterator koji pokazuje na elemente niza `iterable`. Za razliku od programskih jezika C i C++, `element` se ne briše nakon što `for` naredba završi.

```
>>> for broj in range(2, 5):
...     print(broj)
2
3
4
>>> print(broj)
4
```

U slijedećem primjeru vidimo upotrebu `for` naredbe kod lista i stringova.

```
>>> rijeci = ['ide', 'patka', 'preko', 'save']
>>> for rijec in rijeci:
...     print(rijec)
ide
patka
preko
save
>>> for slovo in 'patka':
...     print(slovo)
p
a
t
k
a
```

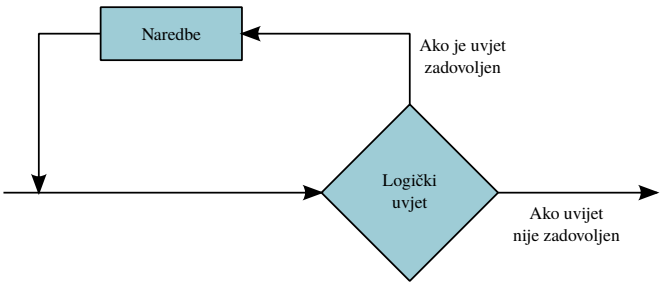
1.7.2 Naredba while

Osnovna sintaksa naredbe `while` je

```
while conditon: commands
```

Naredbe `commands` izvršavat će se sve dok je uvjet `condition` istinit. While petlja se koristi kada ne znamo unaprijed koliko će se puta blok naredbi morati izvršiti. Općenito, ne možemo garantirati da će while petlja završiti pa programer mora sam osigurati da se ne dogode beskonačne petlje, odnosno petlje koje nikada neće završiti.

```
>>> i = 0
>>> while i*i <= 9:
...     print(i)
...     i += 1
```



Slika 1.5
Dijagram toka `while` petlje

```
...
0
1
2
3
```

Primjer 1.1 Jedan od tipičnih primjera za upotrebu while petlje je Collatzova hipoteza. Lothar Collatz je 1937. postavio hipotezu da ako formiramo niz brojeva tako da je prvi član niza $a_0 = n$, a ostali prate pravilo

$$a_n = \begin{cases} n/2 & \text{ako } n \bmod 2 = 0 \\ 3n+1 & \text{ako } n \bmod 2 = 1, \end{cases}$$

da će takav niz u konačno mnogo koraka doći do vrijednosti 1.

Programska implementacija računanja niza po ovom pravilu dana je u Python kodu 1.7.

Python kod 1.7 Upotreba `while` naredbe na Collatzovoj hipotezi

```
n = 5
print(n)

while n != 1:
    if n % 2 == 0:
        n //= 2
    else:
        n = 3*n + 1
    print(n)
```

Izvođenje ovog koda daje:

```
5
16
8
4
2
1
```

U programu koji testira Collatzovu hipotezu nismo koristili zaštitu od beskonačne petlje. Iako hipoteza još nije dokazana, do sada nitko nije uspio naći takav početni n za koji petlja ne bi završila u konačno mnogo koraka. ■

1.7.3 Naredba break

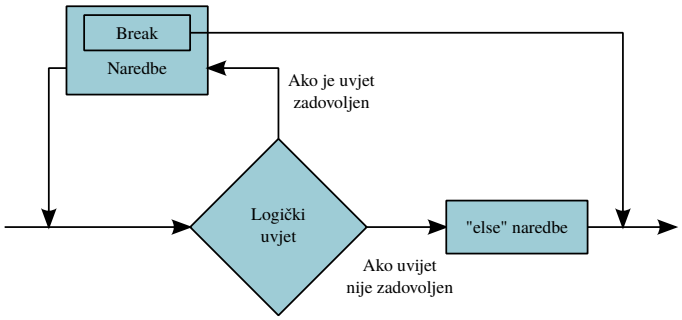
Proširena sintaksa `for` i `while` naredbi uključuje i naredbu `else`:

```
for iter in iterable: commands1
else: comands2
```

```
while condition: commands1
else: comands2
```

Nakon što se iscrpe svi elementi niza `iterable` ili nakon što uvjet `condition` postane neistinit, izvršit će se naredbe `commands2`.

Naredbe specificirane u `else` bloku izvode se kada se petlja završi zbog nezadovoljavanja uvjeta sprecificiranog za `while` petlju, ali ne i u slučaju prekida petlje zbog `break` naredbe (Slika 1.6).



Slika 1.6
Dijagram toka while petlje sa else blokom

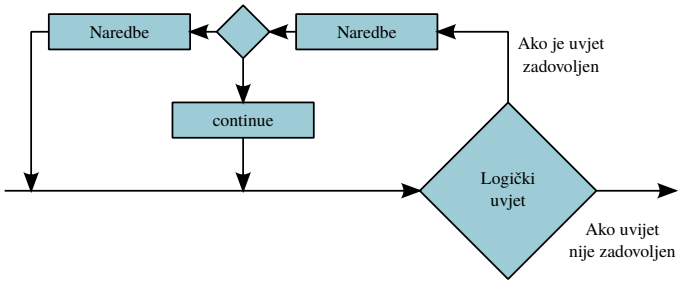
```
>>> for broj in range(3):
...     print(broj)
... else:
...     print('Kraj')
...
0
1
2
Kraj
```

Naredbom `break` završava se `for` ili `while` naredba. Na primjeru slijedećeg može se vidjeti da se nakon `break` naredbe ne izvrši dio koda unutar `else` naredbe.

```
>>> i = 0
>>> while i*i <= 9:
...     print(i)
...     i += 1
...     if i == 2:
...         break
... else:
...     print('Kraj')
...
0
1
```

1.7.4 Naredba continue

Naredbom `continue` preskače se izvršavanje slijedećih naredbi u tijelu `for` ili `while` naredbe i prelazi se na novu iteraciju (Slika 1.7).



Slika 1.7
Dijagram toka `while` petlje sa `continue` naredbom

```
>>> for broj in range(2, 5):
...     if broj == 3:
...         continue
...     print(broj)
...
2
4
```

1.7.5 Paralelno iteriranje

Funkcijom `zip` moguće je elemente više sekvenci pohraniti u jednu listu koja sadrži tupleove pripadajućih elemenata. Takva mogućnost restrukturiranja podataka vrlo je korisna kod potrebe za istovremenim iteriranjem kroz više nizova podataka.

```
>>> imena = ['Vlatko', 'Ivana', 'Darko', 'Igor']
>>> prezimena = ['Hoorvat', 'Bali', 'Copljar', 'Worry']
>>> list(zip(imena, prezimena))
[('Vlatko', 'Hoorvat'), ('Ivana', 'Bali'), ('Darko', 'Copljar'),
('Igor', 'Worry')]
>>> for ime, prezime in zip(imena, prezimena):
...     print(ime, prezime)
...
Vlatko Hoorvat
Ivana Bali
Darko Copljar
Igor Worry
```

Ukoliko liste ne sadrže jednak broj elemenata, formira se lista jednake duljine kao i najkraća lista dana kao argument u `zip` funkciji.

```
>>> a = [1, 2, 3, 4, 5, 6, 7]
>>> b = ['a', 'b', 'c']
>>> list(zip(a, b))
[(1, 'a'), (2, 'b'), (3, 'c')]
```

Broj argumenta funkcije `zip` može biti i više od dva, a rezultat je analogan korištenju funkcije `zip` s dva argumenta.

```
>>> visine = [1.79, 1.86, 2.10, 2.03]
>>> for i, p, v in zip(imena, prezimena, visine):
...     print('{} {} je visok {:.2f} m'.format(i, p, v))
...
```

```
Vlatko Hoorvat je visok 1.79 m
Ivana Bali je visok 1.86 m
Darko Copljar je visok 2.10 m
Igor Worry je visok 2.03 m
```

1.7.6 Numeriranje liste

Funkcija `enumerate` omogućava numeraciju elemenata liste na način da kreira iterabilni enumerate objekt koja sadrži tuplove elemenata izvorne liste te njihve indekse. Dodatno, `enumerate` može primiti argument koji ukazuje od kojeg broj počinje numeracija.

```
>>> brojevi = [12.3, 18.6, 2.1, 12.3]
>>> for indeks, broj in enumerate(brojevi):
...     print('Na indeksu {} je broj {}'.format(indeks, broj))
...
Na indeksu 0 je broj 12.3
Na indeksu 1 je broj 18.6
Na indeksu 2 je broj 2.1
Na indeksu 3 je broj 12.3
>>> for mjesto, broj in enumerate(brojevi, 1):
...     print('Na {}. mjestu je broj {}'.format(mjesto, broj))
...
Na 1. mjestu je broj 12.3
Na 2. mjestu je broj 18.6
Na 3. mjestu je broj 2.1
Na 4. mjestu je broj 12.3
```

1.7.7 Apstraktno generiranje lista, skupova i dictionaryja

Python je usvojio koncept koji se naziva *list comprehension*, odnosno apstraktno generiranje lista, koristeći notaciju kojom se opisuju skupovi u matematičkim tekstovima. Liste je moguće stvoriti na vrlo prirodan i kompaktan način koji koriste i matematičari kada opisuju skupove, n-torke i vektore.

Uzmimo primjer dva skupa:

$$\begin{aligned} S &= \{\, x^2 \mid x \in \mathbb{N} \wedge x < 16 \,\} \\ G &= \{\, x \mid x \in S \wedge x \bmod 2 = 0 \,\} \end{aligned} \tag{1.1}$$

Python omogućava apstraktno generiranje listi koje se sastoje od istih elemenata kao i skupovi u (1.1):

```
>>> S = [x**2 for x in range(1, 16)]
>>> S
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225]
>>> G = [x for x in S if x % 2 == 0]
>>> G
[4, 16, 36, 64, 100, 144, 196]
```

Opisani način generiranja liste je samo skraćeni postupak koji bismo morali napraviti pomoću više naredbi:

```
>>> dvostruki = []
>>> for broj in range(10):
...     if broj % 2 == 0:
...         dvostruki.append(2 * broj)
```

```
...
>>> dvostruki
[0, 4, 8, 12, 16]
>>> dupli = [2*x for x in range(10) if x % 2 == 0]
>>> dupli
[0, 4, 8, 12, 16]
```

U prethodnom primjeru obje liste sadrže iste elemente, ali je upotreba skupovne notacije znatno jednostavnija i lakša za čitanje.

Apstraktno generiranje liste je nešto složenije ako moramo for petlju ugnjezditi u drugu petlju. U slijedećem primjeru vidimo usporedbu generiranja liste pomoću ugnježđenih petlji, sa i bez apstraktnog generiranja.

```
>>> m = [[2, 3, 5],
...      [7, 9, 10],
...      [32, 33, 35]]
>>> t = []
>>> for red in m:
...     for broj in red:
...         t.append(broj**2)
...
>>> t
[4, 9, 25, 49, 81, 100, 1024, 1089, 1225]
>>> t2 = [broj**2 for red in m for broj in red]
>>> t2
[4, 9, 25, 49, 81, 100, 1024, 1089, 1225]
```

Vidimo da je redoslijed for petlji kod apstraktnog generiranja isti kao i kod normalnog iteriranja – prvo ide vanjska petlja, a zatim unutarnja.

Isti princip apstraktnog generiranja može se upotrijebiti na skupovima i dictionaryjima:

```
>>> rijeci = ['ide', 'patka', 'preko', 'save']
>>> unique_slova = {slovo for rijec in rijeci for slovo in rijec}
>>> unique_slova
{'t', 'a', 'k', 'p', 'v', 'd', 'i', 's', 'r', 'o', 'e'}
>>> duljina_rijeci = {rijec: len(rijec) for rijec in rijeci}
>>> duljina_rijeci
{'ide': 3, 'patka': 5, 'preko': 5, 'save': 4}
```

1.8 Funkcije

Marko Čavrak
Stefan Ivić
Miran Tuhtan
Luka Grbčić
todo: update author list

Funkcije su jedne od fundamentalnih stavki Python programskog jezika. Funkcije omogućuju modularnost, višekratnu upotrebu i generalno bolju organizaciju programskog koda. Točnije, omogućuju rastavljanje kompleksniji dijelova Python koda u manje segmente kojima se može lakše kontrolirati. Modularnost je svojstvo koje čini Python kod lakšim za čitanje, pisanje i održavanje. Svaka funkcija se može napisati, nezavisno testirati te omogućuje lakši pronalazak potencijalnih grešaka u kodu. Višekratna upotreba Python funkcija omogućuje da se uštedi vrijeme i trud jer se ista funkcija može upotrebiti više puta u programskom kodu. Ovo svojstvo je iznimno bitno kod izgradnje veliki aplikacija gdje se izvode operacije koje se često ponavljaju.

Ako Python funkcije kategoriziramo prema izvoru njihove definicije, postoje tri osnovna tipa Python funkcija:

- Funkcije koje su ugrađene u Python programski jezik
- Funkcije koje je korisnik definirao
- Funkcije koje je korisnik definirao–dostupne u Python modulima

Funkcije koje su ugrađene u Python programski jezik su dostupne u svakom Python okruženju bez dodavanja modula te su to funkcije koje omogućuju osnovne operacije poput ispisa tipa varijable (`type`) ili ispisa poruke u konzolu (`print`).

Funkcije koje je korisnik definirao su funkcije koje se koriste kada korisnik želi funkcionalnost u kodu koja nije dostupna u ugrađenim funkcijama. Korisnik mora sam definirati logiku unutar funkcije na temelju potrebne operacije. Funkcije koje je korisnik definirao, ali koje postoje u Python modulima se isto mogu smatrati podskupinom prethodne kategorije. To su funkcije koje već postoje u Python modulima te su vrlo često visoko optimizirane. Korisnik ih može koristiti u vlastitom kodu kada je potrebno pod uvjetom da je dodao potreban modul.

1.8.1 Definicija i pozivanje funkcija

U ovom poglavlju pokazati ćemo kako korisnik sam može definirati funkciju te kako možemo koristiti ili pozvati istu funkciju unutar Python koda. Ključna riječ u Python kodu kojom se definira funkcija je `def` te sintaksa defincije je:

```
def funkcija():
    naredba
```

gdje je `funkcija` odabrano ime funkcije, a `naredba` je naredba koja se izvršava unutar funkcije. Ime funkcije može biti bilo što, ali se moraju pratiti ista pravila koja postoje kod definiranja imena varijabli.

U slučaju da želimo izvršiti naredbu unutar funkcije, potrebno je funkciju pozvati. Funkcija se mora pozvati uvijek nakon što je definirana jer ne možemo pozvati funkciju koju prethodno nismo definirali. Da bi smo pozvali prethodno definiranu funkciju moramo koristiti sintaksu:

```
funkcija()
```

gdje je `funkcija` ime definirane funkcije. Ono što se može uočiti je da su nakon imena zagrade te je to jedan od znakova da se u Python kodu radi o funkciji.

Python kod 1.8 Definicija i pozivanje jednostavne funkcije

```
def pozdrav():
    print('Pozdrav!')

pozdrav()
```

```
Pozdrav!
```

Funkciju je moguće proširiti na način da prima jedan ili više argumenata. Argumenti se definiraju pomoću imena argumenata razdvojenih zarezom u zagradama nakon imena funkcije:

```
def function_name(arg1, arg2, ... , argn):
    commands
```

Funkcija sa argumentima se poziva tako da se u zagradama nakon imena funkcije upišu vrijednosti argumenata. Argumenti se u pozivu funkcije razdvajaju zarezom, kao i prilikom definicije funkcije.

Python kod 1.9 Definicija funkcije koja prima argument

```
def pozdrav(ime):
    print('Pozdrav', ime, '!')

def zbroji(a, b):
    c = a + b
    print(a, '+', b, '=', c)

pozdrav('Marko')
zbroji(2, 5)
```

```
Pozdrav Marko !
2 + 5 = 7
```

1.8.2 Vraćanje vrijednosti iz funkcije

Funkcija može vratiti vrijednost nakon poziva funkcije. Vrijednost koju funkcija vraća doslovno zamjenjuje funkciju u kontekstu u kojem je ona pozvana, a tip vraćenog podatka može biti bilo koji Python objekt. Vraćanje vrijednosti se definira naredbom `return` nakon koje se zaustavlja izvršavanje funkcije:

```
def function_name(arg1, arg2, ... , argn):
    commands
    return value
```

Python kod 1.10 Definicija funkcije koja prima više argumenata i vraća vrijednost

```
def zbroji(a, b):
```

```
    c = a + b
    return c

x = zbroji(2, 6)
y = zbroji(x, -5)

print(x, y)
```

```
8 3
```

Funkcija može sadržavati više `return` naredbi (obično u različitim uvjetnim granajima) i svaka može vratiti različiti tip podatka (Izvorni kod 1.11).

Python kod 1.11 Funkcija koja vraća različite tipove

```
def luda_funkcija(a, b):
    if a > b:
        return a + b
    else:
        return 'malo teksta'

x = luda_funkcija(6, 2)
y = luda_funkcija(x, 10)

print(x, y)
```

```
8 malo teksta
```

Vrlo često javlja se potreba da funkcija vraća više vrijednosti. To se može ostvariti vraćanjem *tuplea*.

Python kod 1.12 Funkcija koja vraća više vrijednosti koristeći *tuple*

```
def f(a, b):
    z = a + b
    r = a - b
    return z, r

x = f(6.0, 2.0)
print(x)

c, d = f(6.0, 2.0)
print(c)
print(d)
```

```
(8.0, 4.0)
8.0
4.0
```

1.8.3 Argumenti sa zadanim vrijednostima

Funkcije mogu biti definirane i sa zadanim vrijednostima argumenata. Argumenti sa zadanim vrijednostima moraju biti definirani nakon argumenata za koje nije zadana vrijednost.

```
def function_name(arg1=val1, arg2=val2, ... , argn=valn):
```

```
    commands
    return value
```

Prilikom poziva funkcije, moguće je funkciju pozvati tako da se zadaju samo oni argumenti koji nemaju zadanu vrijednost.

Python kod 1.13 Funkcija sa zadanim argumentom

```
def f(x, y=0.0):
    if x > y:
        return x - y
    else:
        return y

a = f(6.0, 2.0)
b = f(20.0)
c = f(-3.0)

print(a, b, c)
```

```
4.0 20.0 0.0
```

1.8.4 Keyword i non-keyword argumenti

Funkciju se može pozivati na način da se točno specificira vrijednost argumenta preko njihovog naziva pomoću tzv. *keyword argumenata*. Keyword argumenti su oni argumenti koji se, pri pozivu funkcije, zadaju imenom i vrijednošću, za razliku od non-keyword argumenata koji se zadaju samo kao vrijednost. Redosljed zadavanja keyword argumenata može biti proizvoljan. Kod kombinacije non-keyword i keyword argumenata prvo moraju biti specificirani non-keyword argumenti.

```
# Definiranje funkcije
def f(x, a, b):
    return x * (a + b)

# Poziv funkcije s keyword argumentima
f(x=10, b=0, a=8)

# Poziv funkcije s mjesovitim keyword i non-keyword argumentima
f(10, b=0, a=8)
```

Posebna sintaksa definiranja funkcije pomoću `*` i `**` omogućuje korištenje varijabilnog broja argumenata funkcije. Argument definiran s jednom zvjezdicom, uobičajeno `*args`, obuhvaća sve non-keyword argumente u jedan tuple. Upotrebom dvostruke zvijezdice `**kwargs` omogućuje pohranu svih keyword argumenata u dictionary gdje su ključevi imena argumenata, a vrijednosti argumenata su pohranjene u vrijednosti pod pripadajućim ključem.

Python kod 1.14 Funkcija s `*args` i `**kwargs` argumentima

```
def funkcija(*args, **kwargs):
    print('non-keyword argumenti:', args)
    print('keyword argumenti:', kwargs)

    for key, value in kwargs.items():
        print("%s = %s" % (key, value))
```

```
funkcija(10, 'bla bla', a=3, b=True, c=(0, 0))
```

```
non-keyword argumenti: (10, 'bla bla')
keyword argumenti: {'a': 3, 'c': (0, 0), 'b': True}
a = 3
c = (0, 0)
b = True
```

Prilikom korištenja oba operatora za "raspakiravanje" argumenata (`*` i `**`) važno je poštivati redoslijed. Kod definicije funkcije, kao i kod poziva funkcije, prvo se koriste non-keyword argumenti, a potom keyword argumenti.

Osim kod definicije funkcije, sintaksa `*args` i `**kwargs` može se koristiti i pri pozivu funkcije. Zvijezdicom (`*`) se tuple vrijednosti "raspakiraju" u argumente funkcije. Analogno, keyword argumente definirane u dictionaryu može se proslijediti funkciji pomoću dvije zvijezdice (`**`). U izvornom kodu 1.15 prikazano je korištenje `*args` i `**kwargs` prilikom poziva funkcije.

Python kod 1.15 `*args` i `**kwargs` argumenti prilikom poziva funkcije

```
def funkcija(a, b, c, d, e):
    print('argumenti:')
    print(' a:', a)
    print(' b:', b)
    print(' c:', c)
    print(' d:', d)
    print(' e:', e)

a = (1, 2, 3, 4, 5)
funkcija(*a)

kwa = {'d': 11, 'c': 12, 'a': 13, 'e': 14, 'b': 15}
funkcija(**kwa)

a = (5, 4)
kwa = {'e': 10, 'd': 20, 'c': 30}
funkcija(*a, **kwa)
```

```
argumenti:
 a: 1
 b: 2
 c: 3
 d: 4
 e: 5
argumenti:
 a: 13
 b: 15
 c: 12
 d: 11
 e: 14
argumenti:
 a: 5
 b: 4
 c: 30
 d: 20
 e: 10
```

1.8.5 Ugnježdena definicija funkcije

Ugnježdeno definiranje funkcije (eng. *closure*) omogućuje definiciju funkcije u funkciji. Ugnježdena definicija funkcije je moguća u Pythonu jer su funkcije objekti te ih je moguće proslijeđivati kao argumente drugim funkcijama te vraćati kao rezultat funkcije. Ugnježdeno definirane funkcije mogu se pokazati korisne jer unutarnja funkcija može koristiti objekte definirane u vanjskoj:

```
>>> def vanjska(vanjski_argument):
...     def unutarnja(unutarnji_argument):
...             return vanjski_argument - unutarnji_argument
...     return unutarnja
...
>>> f = vanjska(10) # vanjski argument
>>> f(3)            # unutarnji argument
7
>>> f(15)
-5
```

1.8.6 Anonimne funkcije

Lambda funkcije su funkcije bez imena, tj. anonimne funkcije. Najčešće se koriste za definiranje vrlo jednostavnih i kratkih funkcija. Definiranje anonimne funkcije se vrši korištenjem ključne riječi `lambda`:

```
lambda argumenti: izraz
```

Upotrebu jednostavne funkcije koja vraća apsolutnu razliku dva broja

```
def diff(a, b):
    return abs(a - b)

print(diff(7, 11))
```

```
4
```

može se zamjeniti lambda funkcijom:

```
>>> diff = lambda a, b: abs(a - b)
>>> diff(7, 11)
4
```

Anonimne funkcije često se koriste kao argumenti drugih funkcija koje očekuju objekt tipa funkcija kao argument.

Primjerice, `sorted` funkcija očekuje argument `key` kao funkciju, koja je u većini slučajeva veoma jednostavna funkcija i obično se koristi anonimna funkcija da se izbjegne posebno definiranje funkcije. U slijedećem primjeru napravljeno je sortiranje liste tupleova po drugim članovima tupleova:

```
>>> sorted([(5, 5), (13, 2), (11, 6), (7, 4)], key=lambda a: a[1])
[(13, 2), (7, 4), (5, 5), (11, 6)]
```

U slijedećem primjeru prikazano je filtriranje članova niza djeljivih sa 7:

```
>>> parni = list(range(2, 100, 2))
>>> list(filter(lambda x: x%7 == 0, parni))
[14, 28, 42, 56, 70, 84, 98]
```

Još jedan primjer upotrebe anonimnih funkcija je u ugnježdenim definicijama funkcijama.

Python kod 1.16 Primjer anonimne funkcije u ugnježdenoj definiciji

```
def slozena_funkcija(ime):
    return lambda poruka: ime + ': ' + poruka

p = slozena_funkcija('Mario')
print(p('Bok!'))
print(p('Kako ste?'))
```

```
Mario: Bok!
Mario: Kako ste?
```

1.8.7 Rekurzivne funkcije

Do sada smo se već susreli s petljama i načinom kako funkcionira iteracijski postupak kao sukcesivno povećavanje indeksa, tj. brojčanika koji sudjeluje u operaciji koja u konačnici daje finalni rezultat. Primjer je suma uzastopnih brojeva od 1 do 100.

```
suma = 0
for index in range(1, 101):
    suma = suma + index

print(suma)
```

Znači, suma se postepeno povećava dok se ne iscrpi lista, tj. sekvenca indeksa definirana pozivom na `range` naredbu.

```
index = 1
    suma = 0 + 1
index = 2
    suma = 1 + 2
index = 3
    suma = 3 + 3
index = 4
    suma = 6 + 4
```

i tako dalje dok *index* ne postane 100. Drugim riječima postepeno se na varijablu nadodaju brojevi.

Također je moguće zapisati isti postupak prema koracima kako je izvršeno zbrajanje

```
suma = (0)
index = 1 : suma = ((0) + 1)
index = 2 : suma = (((0) + 1) + 2)
index = 3 : suma = ((((0) + 1) + 2) + 3)
itd.
```

Kada bismo indeksirali parcijalne sume tada bismo dobili izraz:

$$suma_{index} = suma_{index-1} + index\ .$$

Tako zapisani izraz daje naslutiti kako je slijedeća suma zapisana kao suma iz prethodnog koraka na koju se nadodaje vrijednost index iz trenutnog koraka iteracijske petlje. Ovakav zapis u biti je obrnuti zapis od prirodnog sukcesivnog povećavanja indeksa:

```
suma(100) = suma(99) + 100, gdje je
suma(99) = suma(98) + 99, gdje je
suma(98) = suma(97) + 98, i tako dalje sve do
.
.
.
suma(1) = suma(0) + 1
```

Mogućnost zapisa funkcije u obliku koji poziva istu funkciju nazivamo *rekurzija*, a takvu funkciju *rekurzivna funkcija*.

Tipična rekurzivna funkcija vrši poziv sama na sebe. Primjer je funkcija `rsuma`:

```
def rsuma(x):
    return rsuma(x)
```

No, kao što vidimo, ova funkcija vraća poziv na samu sebe koja vraća poziv na samu sebe koja vraća poziv na samu sebe i tako u beskonačnost. Ovako definirana rekurzivna funkcija nema kraja, a program ulazi u *beskonačnu petlju* (eng. *infinite loop*).

Kako bismo spriječili ovakvo ponašanje potrebno je uvesti *uvjet prekida*. To je granična vrijednost nakon koje funkcija više neće pozivati sebe već vratiti neku vrijednost. Ako se sada vratimo na sumu brojeva onda bi rekurzivna funkcija sume trebala izgledati ovako:

```
def rsuma(n):
    return rsuma(n - 1) + n
```

No oznaka prekida je `suma(0) = 0`, tj. kada u 100-tom pozivu funkcije `rsuma` bude primila za argument 0 potrebno je zaustaviti daljnje pozive `rsuma` funkcije. To ćemo postići testiranjem vrijednosti n:

```
def rsuma(n):
    if n == 0:
        return 0
    else:
        return rsuma(n - 1) + n
```

Na taj način omogućili smo oznaku kraja rekurzivnim pozivima funkcije.

Rekurzivni zapis unosi konciznost i eleganciju u programski kod. Slijedi nekoliko primjera.

Primjer 1.2 Kao primjer rekurzivnog postupka možemo uzeti proračun faktorijela broja n.

Faktorijel broja n se izračunava kao $n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot ... \cdot n$ odnosno rekurzivno kao $n! = (n-1)! \cdot n$.

Programska implementacija ovog postupka izgleda ovako:

```
def faktorijela(n):
    if n == 0:
        return 1
    else:
        return faktorijela(n - 1) * n
```

■

Primjer 1.3 Promotrimo potenciranje broja x eksponentom n.

Potenciranje je također iterativni proces koji može biti zapisan rekurzivno kao $x^n = x^{n-1} \cdot x$. Primjerice kub je jednak umnošku broja i kvadrata broja, tj. $x^3 = x^2 \cdot x$.

Opisani postupak može se implementirati na sljedeći način:

```
def potencija(x, n):
    if n == 0:
        return 1
    else:
        return potencija(x, n - 1) * x
```

■

Kao zaključak, potrebno je napomenuti da je rekurzivnu funkciju moguće napraviti za svaki poziv petlje, no je li nužno sve petlje pretvoriti u rekurzivne funkcije to ostaje na procijeni programeru koji mora odvagnuti količinu elegancije, hijerarhije, strukturiranosti te preglednosti koda.

1.9 Objektno-orijentirano programiranje

Marko Čavrak
Miran Tuhtan

Python je objektno orijentirani programski jezik. Pojam objekt je ključ za razumijevanje objektno-orijentiranog pristupa programiranju. Objekt je kompozicija *stanja* (atributi, varijable) i *ponašanja* (metode, funkcije). Stanja jednog objekta čine njegovi atributi (npr. objekt automobil može imati atribute promjer kotača, snaga motora, trenutni prijenosni omjer) koji su u programu utjelovljeni kao varijable koje poprimaju određene vrijednosti (stanja). Ponašanje ili aktivnosti određenog objekta utjelovljene su u programu u obliku funkcija koje izvršavaju određene radnje važne za objekt ili korisnike koji koriste objekte (npr. metode koje može izvršiti automobil su kočenje, ubrzavanje, mijenjanje brzine, skretanje). Tako definirani objekti omogućuju proširenje postojeće funkcionalnosti programskog jezika i time programeru olakšavaju pisanje programa. Često ih se može susresti prilikom izrade aplikacija s grafičkim sučeljem ili bilo kojoj drugoj aplikaciji čiji korisnici moraju stvarati objekte, mijenjati njihova stanja i koristiti metode kako bi definirali ponašanje ili uzrokovali neku aktivnost vezanu uz objekt. Najlakše će biti to shvatiti na slijedećem primjeru.

Primjer: "Aplikacija za pisanje teksta" omogućava korisniku slijedeće funkcionalnosti:

- izradu novog dokumenta - (u programskom kodu dokument je objekt)
- mijenjanje sadržaja dokumenta (sadržaj je jedan od atributa objekta)
- mijenjanje naziva dokumenta (naziv je jedan od atributa objekta)
- zapis sadržaja dokumenta u datoteku (*Save*, *Save As* funkcije kao metode objekta dokumenta)
- otvaranje dokumenta iz datoteke (*Open* funkcija kao metoda objekta dokumenta)
- lijevo, centralno, podjednako i desno poravnanje teksta u sadržaju (Funkcije *Left*, *Center*, *Right*, *Justify* kao metode objekta dokumenta)
- itd.

Pored ovih stanja i ponašanja jednog dokumenta kao objekta lako biste mogli poznavajući MS Word ili neku drugu aplikaciju za uređivanje i editiranje teksta osmisliti ili se sjetiti još neke od mnogih stanja i ponašanja koje MS Word posjeduje.

Aplikacija poput ove također mora omogućiti i izradu ili učitavanje nekoliko objekata kako bismo mogli uspoređivati ili kopirati njihov sadržaj ili recimo spojiti više dokumenata u jedan. Da bi to omogućila, aplikacija mora posjedovati način kako da omogući istovremenu pojavu više dokumenata. Dokument je opisan kao objekt u programskom kodu. Više dokumenata time čini više objekata, gdje svaki može imati sadržaj i naziv kao svoje atribute te metode Open, Save, Left, itd. kao ponašanja koja aplikacija omogućuje pojedinom dokumentu tj. objektu.

Time dolazimo do problema što svi izrađeni dokumenti imaju varijable kao stanja i funkcije kao metode koje su jednake za sve dokumente/objekte. Kako bi aplikacija svim dokumentima omogućila takvo nešto, mora u aplikaciji postojati zapis koji definira kako će se ponašati svaki pojedini objekt dokumenta te koje će funkcionalnosti svakom dokumentu aplikacija omogućiti.

Taj zapis u objektno-orijentiranim programskim jezicima nazivamo *klasom*, a svaki

poziv na izradu novog objekta *instanciranje*. Time je svaki objekt definiran kao jedna *instanca klase*.

1.9.1 Klase

Klase predstavljaju definiciju svih atributa i metoda koje će instance te klase poprimiti. Za definiranje klase, tj. opisa objekta, koristi se izraz

```
class ImeKlase:
```

nakon kojeg slijede definicije atributa i metoda.

Python kod 1.17 Definicija klase

```
class Dokument:
    """
    Klasa Dokument omogucuje instanciranje objekata Dokument
    """
```

```
>>> d1 = Dokument()
>>> d2 = Dokument()
>>> d1
<__main__.Dokument instance at 0x7f050f2c0098>
>>> d2
<__main__.Dokument instance at 0x7f05000287e8>
>>>
```

Prikazani izvorni kod i output iz Python konzole pokazuju način definicije klase te način instanciranja dva objekta `d1` i `d2`. Printanjem u konzoli objekata `d1` i `d2` moguće je vidjeti da su oba objekta instance te da je svaka zasebno referencirana u memoriji na drugo adresno mjesto.

1.9.2 Atributi

Atributi omogućuju dodjeljivanje stanja objekata. Njihova definicija nalazi se unutar klase i vrijedi za sve objekte instancirane iz klase.

Postoje dvije vrste atributa

- Klasni (globalni) atributi
- Objektni (lokalni) atributi

Klasni atributi

Atributi koji pripadaju svim objektima zovu se klasni atributi, i definiraju se odmah poslije definicije klase. To su globalne varijable koje vrijede za sve objekte instancirane iz te klase.

Python kod 1.18 Primjer definicije klasnih (globalnih) atributa

```
class Dokument:
    """
    Klasa Dokument omogućuje instanciranje objekata Dokument
    """

    ime = 'Dokument1'
    sadrzaj = 'Ovo je prva linija u dokumentu'

d1 = Dokument()
```

```
d2 = Dokument()
```

```
>>> d1.ime
'Dokument1'
>>> d2.ime
'Dokument1'
>>> Dokument.ime
'Dokument1'
```

Atribut `ime` je stoga dostupan svim objektima kao i samoj klasi `Dokument`. Potrebno je napomenuti da je moguće doći do pogreške ako objekt nastoji mijenjati vrijednost klasnom atributu. Naime, takvim činom nije izmijenjena vrijednost klasnom atributu već je izrađen novi atribut istoga imena, a koji pripada objektu.

```
>>> d2.ime
'Dokument1'
>>> d2.ime = 'Doc'
>>> Dokument.ime
'Dokument1'
```

Svrha postojanja globalnih atributa je u tome da omoguće identifikaciju objekata instanciranih iz jedne klase. Slijedeći primjeri primjene dati su na klasi Parkirališta:

- identifikacijski broj objekta (npr. Parking01, Parking02, ...)
- lista objekata (npr. [obj1, obj2, obj3])
- kumulativne sume (npr. ukupan broj automobila parkiranih na svim parkiralištima)

Objektni atributi

Objektni atributi čine atribute definirane u klasi unutar posebne metode koja se zove `__init__`. Ova metoda pripada u kategoriju *magičnih metoda klase* jer omogućuju napredne mogućnosti. Inicijalizacija je u drugim programskim jezicima prisutna pod nazivom *konstruktor*. Ona ima za cilj omogućiti definiciju atributa koji će poprimati stanja definirana prilikom stvaranja objekta.

Python kod 1.19 Primjer definicije objektnog atributa

```
class Dokument:
    """
    Klasa Dokument omogućuje instanciranje objekata Dokument
    """
    def __init__(self):
        self.ime = ''
        self.sadrzaj = ''

d1 = Dokument()
d2 = Dokument()

d1.ime = 'Prica01'
d1.sadrzaj = 'Bio jednom jedan val.'

d2.ime = 'ListaSerija'
d2.sadrzaj = 'Lud, zbunjen, normalan.\n'
d2.sadrzaj += 'Sila\n'
```

Metoda `__init__` omogućuje, također, prihvat argumenata predanih Klasi prilikom formiranja objekta. To činimo tako da u argument listu metode `__init__` predamo vrijednosti koje želimo da pojedini atributi tog objekta poprime.

Python kod 1.20 Inicijalizacija objekta

```
class Dokument:
    """
    Klasa Dokument omogućuje instanciranje objekata Dokument
    """
    def __init__(self, name='Dokument', content=''):
        self.ime = name
        self.sadrzaj = content


d1 = Dokument('Esej', 'Prva recenica')
d2 = Dokument('Blog01')
d3 = Dokument()
```

Do ovog koraka već ste uvidjeli da se pozivi atributa unutar metode `__init__` vrše pomoću specifične riječi *self*. Više riječi o tome bit će u sljedećem poglavlju.

1.9.3 Metode

Metode su funkcije koje koristimo da u klasi definiramo promjenu stanja atributa, izvršimo neke aktivnosti ili obavimo neke druge radnje vezane uz određeni objekt. Svaka metoda mora biti definirana kao funkcija i posjedovati minimalno jedan argument, a to je `self`, kao opis objekta za kojega se izvršava metoda. Slijedeći primjer definira metodu `info()` koja omogućuje printanje naziva i sadržaja objekta koji ju je pozvao.

Obratite pažnju na korišteni globalni atribut `dokument_list` kao listu objekata koja se puni prilikom inicijalizacije.

Python kod 1.21 Definicija metode

```
class Dokument:
    """
    Klasa Dokument omogucuje instanciranje objekata Dokument
    """

    dokument_list = []

    def __init__(self, name='Ime', content=''):
        self.ime = name
        self.sadrzaj = content
        Dokument.dokument_list.append(self)

    def info(self):
        print('Ime dokumenta:', self.ime)
        print('Sadrzaj:\n', self.sadrzaj)

d1 = Dokument('Esej', 'Prva rečenica')
d2 = Dokument('Blog01')
d3 = Dokument()

for d in Dokument.dokument_list:
    d.info()
    print('\n')
```

`self` prvi put dobiva značaj kao prvi član argument liste metode `__init__` (`def __init__(self, arg1, arg2, ...)`). Unutar same metode svaki od atributa koji se koriste mora biti zapisan uz pomoć `self.` prefiksa.

Možemo to gledati na drugačiji način. Zamislite da je klasa modul koji ima svoje pod-funkcije (metode). Svaka od tih pod-funkcija ima mogućnost mijenjati atribute nekom objektu. To je ostvarivo samo ako pod-funkcija može pristupiti objektu i njegovim atributima.

Upravo to činimo predajući `self` metodi `__init__`. Predajemo objekt u obliku riječi `self`, te nakon toga vrijednosti koje želimo dodijeliti atributima. No u stvarnosti Python dozvoljava pozivanje funkcija bez davanja sebe kao prvog člana argument liste.

Na slijedećem primjeru uspoređujemo dva istovjetna poziva metode gdje je prva na način da objekt poziva metodu, a druga da Klasnoj metodi predamo objekt kao prvi član argument liste te nakon njega sve ostale argumente.

```
>>> d1.info()
Ime dokumenta: Esej
Sadrzaj:
Prva recenica
>>> Dokument.info(d1)
Ime dokumenta: Esej
Sadrzaj:
Prva recenica
```

Znači `d1.info()` je isto što i `Dokument.info(d1)`. Prvi omogućava jednostavniji poziv metode dok drugi ukazuje način kako Python ustvari radi, tj. predaje metodi objekt koji u deklaraciji metode stoji kao `self`.

1.9.4 Preopterećivanje

Jedna od osobina objektno-orjentiranog programiranja je mogućnost preopterećivanja funkcija i operatora kako bismo i objektima omogućili istu funkcionalnost kao i klasičnim tipovima podataka poput *int*, *float*, *bool*, *string*, *list*, itd. Takva osobina često je i nužnost kako bismo operatore mogli dovoditi u određene odnose i operacije ili nad objektima izvršiti radnje poput printanja.

Često se za evaluaciju ili grananje služimo operatorima usporedbe (>, >=, <, <=, ==) te logičkim operatorima *and* i *or*. Istu funkcionalnost moguće je postići i usporedbom dvaju objekata gdje se objekti pritom uspoređuju na način da im se točno odredi što i kako se uspoređuje (npr. dvije košare voća *Kosara01* i *Kosara02* su objekti čiji sastav čini raznovrsno voće gdje svako ima svoju težinu i moguće je u svakom slučaju utvrditi ukupnu težinu košare te pristupiti usporedbi dvaju košarica).

Nadalje, objekti mogu biti tako osmišljeni da je od značaja primjena osnovnih numeričkih operatora (+, -, *, /, itd.) kako bi se kreirali novi objekti s izmjenjenim atributima na temelju numeričkih operatora (npr. ako su one dvije košare iz prethodnog zadatka bile pune i u njih više ne stane onda je moguće u novu košaru dodati sastojke iz prethodnih dviju punih košara).

Sva preopterećenja u Pythonu predefinirana su u obliku specijalnih metoda koje svaka klasa može implementirati. Specijalne metode imaju naziv omeđen sa dvije donje crte ispred i iza naziva (npr. `__str__` ili `__gt__`).

Preopterećivanje operatora

Preopterećivanje operatora omogućeno je u Pythonu korištenjem specijalnih metoda slijedećih imena

Operatori usporedbe

- `__gt__` - Veće od (>)
- `__ge__` - Veće ili jednako od (>=)
- `__lt__` - Manje od (<)
- `__le__` - Manje ili jednako od (<=)
- `__eq__` - Jednako (==)

Python kod 1.22 Preopterećivanje operatora usporedbe

```
from math import hypot

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def R(self):
        # vraća duljinu radij-vektora točke,
        # tj. udaljenost od ishodišta
        return hypot(self.x, self.y)

    # Preopterećeni operatori usporedbe točaka
    # s obzirom na udaljenost od ishodišta

    def __lt__(self, other): # less than
        return self.R() < other.R()

    def __le__(self, other): # less or equal
        return self.R() <= other.R()

    def __gt__(self, other): # greather than
        return self.R() > other.R()

    def __ge__(self, other): # greater or equal
        return self.R() >= other.R()

p1 = Point(2, 1)
p2 = Point(1, 2)
p3 = Point(3, 0)

print(p1 > p2)
print(p1 >= p2)
print(p1 <= p2)
print(p1 < p3)
print(p1 == p2)
```

```
False
True
True
False
True
```

Aritmetički operatori

- `__add__` - zbrajanje (+)
- `__sub__` - oduzimanje (-)

- `__mul__` - množenje (*)
- `__div__` - djeljenje (/)

Python kod 1.23 Preopterećivanje funkcije *print*

```
class Table:
    def __init__(self, a=1.0, b=1.0):
        self.a = a
        self.b = b
        self.area = self.a * self.b # atribut površine stola

    def __str__(self):
        s = 'Table = {:4.1f} x {:4.1f}\nArea = {:6.2f}\n'
        return s.format(self.a, self.b, self.area)

    def rotate(self): # rotira stranice a i b
        self.a, self.b = self.b, self.a
        print('Stol zarotiran!')

    # Preopterećeni aritmetički operatori
    def __add__(self, other): # zbraja dva stola
        self.a = self.a + other.a
        self.area = self.area + other.area
        self.b = self.area / self.a # izračunavamo novu prosječnu širinu
        print('Stolovi spojeni!')

    def __sub__(self, other): # oduzima dva stola
        if self.area > other.area:
            if self.a > other.a:
                if self.b > other.b:
                    self.area -= other.area
                    self.a -= other.a
                    self.b = self.area / self.a
                    print('Stolovi oduzeti!')
                else:
                    print('Drugi stol je širi od prvog.')
            else:
                print('Drugi stol je dulji od prvog.')
        else:
            print('Površina drugog je veća od prvog stola.')

t1 = Table(10, 20)
t2 = Table(20, 40)
print(t1)
print(t2)
t1 - t2
t2 - t1
t1 + t2
t1.rotate()
t1 - t2
print(t1)
print(t2)
```

```
Table = 10.0 x 20.0
Area = 200.00

Table = 20.0 x 40.0
Area = 800.00
```

```
Povrsina drugog je veca od prvog stola.
Stolovi oduzeti!
Stolovi spojeni!
Stol zarotiran!
Drugi stol je siri od prvog
Table = 40.0 x 20.0
Area = 800.00

Table = 10.0 x 60.0
Area = 600.00
```

Postoji još mnogo specijalnih metoda, ali njihovo izučavanje prepuštamo vama da ih istražite ako smatrate potrebnim.

1.9.5 Nasljeđivanje

Objekti koje je moguće susresti u stvarnom životu najčešće su kategorizirani u neke grupacije ili klastere, ili su dio neke hijerarhijske strukture. Rijetko da je neki objekt jedinstven u svoj svojoj unutrašnjosti. Čovjek jede, životinja jede, a i biljke jedu tj. izvršavaju jednu od svojih najvažnijih funkcija. Bicikl je kopneno vozilo, kao i automobil, dok je avion ili balon na vrući zrak zračno vozilo, a podmornica ili brod pomorsko vozilo. Sva ova prometala su vozila. Na taj način slijedi hijerarhija koja definira pripadnost pojedinih objekata. Postoje i rekurzivne hijerarhijske veze poput obiteljskih stabala ili svake strukture koja za svoj rezultat ima potomstvo novih objekata.

Nasljeđivanje je u pravilu proces kojim od viših hijerarhijskih razina ili grupacija poprimamo neke osobine ili funkcije.

Objektno-orjentirano programiranje omogućuje naslijeđivanje atributa i metoda iz nadređenih klasa. Deklaracija klase koja posjeduje nadklasu data je u slijedećoj liniji koda:

```
class Sudoku(Igra):
```

Klasa `Sudoku` naslijeđuje metode od svoje nadklase `Igra`, koja se često zove i bazna (osnovna) klasa. Kao što je `Igra` *nadklasa* klasi `Sudoku` tako je `Sudoku` klasi `Igra` *podklasa*.

U slijedećem primjeru dana je nadklasa `Road` koja inicijalizira naziv ceste `name` i duljinu ceste `length` te definira sadržaj metode `info()` koja služi za ispis vrijednosti atributa svakog instanciranog objekta bilo podklase `Highway`, `MainRoad`, `LocalRoad` ili osnovne nadklase `Road`.

Naslijeđivanje ima još jednu prednost koja se očituje u sistematizaciji izvornog koda. Kada ne bismo imali nadklasu, svaka od podklasa trebala bi sama inicijalizirati svoje atribute kao i definirati svoju funkciju `info()`. Na taj način izvorni kod bi bio predugačak, nejasan i nečitljiv.

Python kod 1.24 Naslijeđivanje

```
class Road:
    def __init__(self, name, length):
        self.name = name
        self.length = length

    def info(self):
        print("Road definition")
        print("Ime:\t %15s" % self.name)
```

```
        print("Length:\t %15s" % self.length)
        print()

class Highway(Road):
    pass # Bez lokalnih definicija

class MainRoad(Road):
    pass # Bez lokalnih definicija

class LocalRoad(Road):
    pass # Bez lokalnih definicija

h1 = Highway('A1', 1000)
m1 = MainRoad('E54', 540)
l1 = LocalRoad('S16', 125)
h1.info()
m1.info()
l1.info()
```

```
Road definition
Ime:                  A1
Length:             1000

Road definition
Ime:                 E54
Length:              540

Road definition
Ime:                 S16
Length:              125
```

U situaciji kada pojedina podklasa ipak zahtjeva prilikom inicijalizacije definirati neki vlastiti atribut tada definicija `__init__` unutar podklase *predefinira* (eng. *override*) `__init__` metodu nadklase. U tom slučaju je potrebno pozvati inicijalizacijsku metodu nadklase, koristeći funkciju `super()`, unutar inicijalizacijske metode podklase.

Python kod 1.25 Naslijeđivanje override __init__metode

```
class Road:
    def __init__(self, name, length):
        self.name = name
        self.length = length

    def info(self):
        print("Road definition")
        print("Ime:\t %15s" % self.name)
        print("Length:\t %15s" % self.length)
        print()


class Highway(Road):
    def __init__(self, name, length): # override __init__ nadklase
        super().__init__(name, length) # invokacija __init__-a nadklase
        self.type = 'highway'
        self.toll = 50 # Kn/100km


class MainRoad(Road):
    def __init__(self, name, length):
        super().__init__(name, length)
```

```
        self.type = 'mainroad'


class LocalRoad(Road):
    def __init__(self, name, length):
        super().__init__(name, length)
        self.type = 'localroad'


h1 = Highway('A1', 1000)
m1 = MainRoad('E54', 540)
l1 = LocalRoad('S16', 125)

print(h1.toll)
```

```
50
```

Pored jednostavnih naslijeđivanja moguća su i višestruka naslijeđivanja iz više nadklasa i definiraju se na slijedeći način:

```
class Smartphone(MobileDevice, Computer):
```

Takva naslijeđivanja mogu dovesti do problema ako obje nadklase imaju zajedničku nadklasu (tzv. *diamond problem* pa se preporuča dodatni oprez prilikom korištenja takvog naslijeđivanja ili pokušati izbjeći višestruka naslijeđivanja.

1.10 Oblikovanje stringova

Stefan Ivić
Miran Tuhtan

U sljedećim potpoglavljima objašnjeni su napredniji načini oblikovanja stringova operatorom `%` i metodom `format`.

1.10.1 Oblikovanje operatorom %

String objekti imaju ugrađeni operator (`%`) koji omogućava ubacivanje vrijednosti unutar stringa na posebno označena mjesta. Mjesta na koja treba ubaciti određenu vrijednost u stringu označavaju se znakom `%` i posebnim oznakama koje detaljnije definiraju formatiranje ubačene vrijednosti. Nakon stringa i operatora `%` (nije isto što i oznaka u samom stringu), u tupleu ili dictionaryju se zadaju vrijednosti ili varijable koje treba umetnuti u string.

Izraz koji definira format ubačene vrijednosti sastoji se od:

- znak `%` koji označava početak izraza za definiranje formata,
- ključ mapiranja (neobavezno) koji definira ime (ključ) vrijednosti koja se ubacuje u string,
- konverzijske opcije (neobavezno),
- minimalni broj rezerviranih znakova (neobavezno),
- preciznost (neobavezno), označava broj decimalnih mjesta koje se prikazuju i piše se nakon znaka `.`,
- tip konverzije.

Upotrebom ključeva mapiranja zadavanje argumenata u dictionaryju ne mora biti u istom redoslijedu kao i pozicije ubacivanja u samom stringu. Na ovaj se način argumenti mogu upotrijebiti više puta, bez potrebe da ih se višestruko puta zapisuje unutar dictionaryja. Ako se koristi ključ mapiranja onda vrijednosti treba zadati u dictionaryju, dok se inače zadaju u tupleu.

Konverzijske opcije se definiraju pomoću znakova opisanih u tablici 1.5.

Tablica 1.5
Opcije konverzije

Simbol	Opis
`'0'`	Prazne znakove popunjava s nulama
`'-'`	Poravnanje ulijevo
`' '`	Nezauzete znakove popunjava razmacima
`'+'`	Uvijek postavlja predznak (`+` ili `-`)

Pomoću operatora `%` u string je moguće ubaciti razne numeričke (i string) formate koji se definiraju pomoću simobla danih u tablici 1.6.

Primjer u izvornom kodu 1.26 prikazuje ubacivanje i formatiranje decimalnog i cijelog broja u string. Primjer obuhvaća primjenu tupleova bez ključa mapiranja te na istom primjeru primjenu dictionaryja sa ključevima mapiranja.

Python kod 1.26 Primjer formatiranja numeričkih podataka sa tupleom i dictionaryjem

```
>>> s = 'a = %.2f, b = %d' % (12.365, 18)
```

Tablica 1.6
Definiranje tipova podataka

Simbol	Tip
'd'	Cijeli broj
'o'	Oktalni broj
'x'	Heksadecimalni broj
'e'	Decimalni broj u eksponencijanom obliku
'f'	Decimalni broj
'c'	Tekstualni znak
's'	String
'%'	Nema konverzije argumenta, rezultat sadrži znak '%'

```
>>> print(s)
a = 12.37, b = 18
>>> s = 'a = %(a).2f, b = %(b)d' % {'a': 12.365, 'b': 18}
>>> print(s)
a = 12.37, b = 18
>>> s = 'a = %(a).2f, a = %(a)+.1f' % {'a': 12.365}
>>> print(s)
a = 12.37, a = +12.4
```

Različite primjene konverzijskih opcija prikazane su u izvornom kodu 1.27.

Python kod 1.27 Primjer formatiranja numeričkih podataka sa tupleom i dictionaryjem

```
>>> print('x = [%5.2f, %5.2f]' % (11.14, 7.88))
x = [11.14,  7.88]
>>> print('x = [%f, %f]' % (11.14, 7.88))
x = [11.140000, 7.880000]
>>> print('x = [%10f, %10f]' % (11.14, 7.88))
x = [ 11.140000,   7.880000]
>>> print('x = [%010f, %010f]' % (11.14, 7.88))
x = [011.140000, 007.880000]
>>> print('x = [%+10f, %+10f]' % (11.14, 7.88))
x = [+11.140000,  +7.880000]
>>> print('x = [%-+10f, %-+10f]' % (11.14, 7.88))
x = [+11.140000, +7.880000 ]
>>> print('x = [%-+10.2f, %-+10.2f]' % (11.14, 7.88))
x = [+11.14    , +7.88     ]
>>> print('x = [%+10.2f, %+10.2f]' % (11.14, 7.88))
x = [    +11.14,      +7.88]
```

Python kod 1.28 Primjer formatiranja numeričkih podataka za ispis u stupcima

```
from math import sqrt, sin

l = list(range(1, 11))

for i in range(len(l)):
    s = '%03d %4.1f %6.1f %5.2f %+6.3f' % \
        (i+1, l[i], l[i]**2.0, sqrt(l[i]), sin(l[i]))
    print(s)
```

```
001  1.0    1.0  1.00 +0.841
```

```
002  2.0    4.0  1.41 +0.909
003  3.0    9.0  1.73 +0.141
004  4.0   16.0  2.00 -0.757
005  5.0   25.0  2.24 -0.959
006  6.0   36.0  2.45 -0.279
007  7.0   49.0  2.65 +0.657
008  8.0   64.0  2.83 +0.989
009  9.0   81.0  3.00 +0.412
010 10.0  100.0  3.16 -0.544
```

1.10.2 Oblikovanje metodom format

Svi string objekti sadrže metodu `format` koja omogućava napredno formatiranje i kontrolu stringova.

Analogno oblikovanju operatorom `%`, mjesta na koja treba ubaciti određenu vrijednost označavaju se se vitičastim zagradama (`{ }`).

```
>>> '{} nije {}'.format('prvi', 'drugi')
'prvi nije drugi'
>>> '{} nije {}'.format(11, 2)
'11 nije 2'
```

Moguće je i mijenjati redoslijed argumenata, zadavanjem rednog broja unutar zagrada.

Argumenti se mogu formatirati nakon dvotočke. Primjerice, lijevo poravnanje je `<`, centralno je `^`, a desno je `>`, nakon kojih slijedi broj znakova koji želimo rezervirati za dani argument.

```
>>> '{1} nije {0}'.format('prvi', 'drugi')
'drugi nije prvi'
>>> '{1:^10} nije {0:>15}'.format('prvi', 'drugi')
'  drugi    nije            prvi'
```

Brojevima je također moguće odrediti dopuštenu duljinu. Cijelim brojevima određuje se njihova ukupna duljina, a decimalnim brojevima moguće je odrediti i preciznost nakon decimalne točke.

```
>>> 'odgovor je: {:5d}'.format(42)
'odgovor je:    42'
>>> 'pi je: {:7.2f}'.format(3.141592653589793)
'pi je:    3.14'
>>> 'pi je: {:+09.3f}'.format(3.141592653589793)
'pi je: +0003.142'
```

U gornjem primjeru, `+` postavlja predznak, `0` označava da želimo neiskorištena mjesta ispuniti nulama, `9` je ukupna duljina broja, `.3` određuje preciznost nakon decimalne točke (primijetite: iako je originalno treća znamenka `1`, broj je pravilno zaokružen).

Mjesta na koje se ubacuju stringovi mogu biti i imenovana te je moguće koristiti i keyword-argumente u proizvoljnom redoslijedu ili dictionaryje.

```
>>> '{ime} {ime}-{prezime}'.format(prezime='Ghali', ime='Boutros')
'Boutros Boutros-Ghali'
>>> osoba = {'ime': 'Janko', 'prezime': 'Polic', 'nadimak': 'Kamov'}
>>> '{ime} {prezime} {nadimak}'.format(**osoba)
'Janko Polic Kamov'
```

Više o formatiranju stringova, uz ilustrativne primjere, moguće je pronaći na: https://pyformat.info/.

1.10.3 F-strings oblikovanje

Ovaj način oblikovanja je logičan nastavak `format` metode oblikovanja. Umjesto nastavka `format`, upisuje se znak **`f`** ispred navodnika. Odnosno,

```
>>> a, b = 10, 20
>>> f'a={a}, b={b}'
a=10, b=20
```

Oblikovanje je sada preglednije te se zadržava ista kontrola oblikovanja:

```
>>> import numpy as np
>>> f'Pi je približno (np.pi:.2f}'
Pi je približno 3.14
```

Osim kontrole ispisa unaprijed definiranog broja decimala, jednostavno je poravnavanje: lijevo, centralno i desno. U sljedećem primjeru se poruka poravnava u polju od dvadeset znakova lijevo, centralno i desno. Znak `\n` označava prelazak u novi red.

```
>>> poruka = 'Tekst'
>>> print(f'{poruka:<20}\n{poruka:^20}\n{poruka:>20}')
Tekst
       Tekst
               Tekst
```

Formatiranje ispisa tablice je sada jednostavnije i preglednije:

```
>>> table = {1: 1.97, 2: 100.75, 3: 18400}
>>> for no, cijena in table.items():
...     item= f'item:{no}'
...     print(f'{item:^10} ==> {cijena:>10.2f}')
item:1    ==>       1.97
item:2    ==>     100.75
item:3    ==>   18400.00
```

U gornjem primjeru je string `item:{no}` poravnat centralno u polju od deset znakova. Varijabla `cijena` je desno poravnata u polju od deset znakova, od kojih su zadnja dva znaka decimalna mjesta (`>10.2f`).

Osim varijabli, moguće je ispisati, vrijednost izraza, kao i pozvati funkciju:

```
>>> print(f'{21*2}')
42
>>> name='Rene'
>>> print(f'Zovem se {name.upper()}.')
Zovem se RENE.
>>> print(f'{[2**n for n in range(3, 9)]}')
[8, 16, 32, 64, 128, 256]
```

Od verzije 3.8, omogućen je još jednostavniji ispis varijabli, što se može vidjeti u sljedećem primjeru:

```
>>> import numpy as np
>>> A = 2**2*np.pi
>>> print(f'{A=}')
A=12.566370614359172
```

```
>>> print(f'{A=:.2f}')
A=12.57
```

1.11 Rad s datotekama

Stefan Ivić
Bojan Crnković
Miran Tuhtan

1.11.1 Otvaranje i zatvaranje datoteka

Osnovna manipulacija sa sadržajem datoteka u Pythonu je omogućena korištenjem `file` objekta.

Python ima ugrađenu funkciju `open` za otvaranje postojećih ili stvaranje novih datoteka na disku. Funkcija `open` vraća `file` objekt, koji sadrži metode i atribute za pristup i manipulaciju informacija i sadržaja otvorene datoteke.

Sintaksa otvaranja datoteka je:

```
f = open(ime_datoteke, vrsta_pristupa, buffering)
```

gdje je `ime_datoteke` sting koji sadrži relativnu ili apsolutnu putanju datoteke, `vrsta_pristupa` je string koji sadrži mod pristupa, `buffering` je integer koji označava veličinu međupohrane. Argumenti `vrsta_pristupa` i `buffering` nisu obavezni. `f` je novonastali `file` objekt kreiran pozivom funkcije `open`.

Mod pristupa se definira kao string koji sadrži znakove:

- `'r'`, `'w'` ili `'a'` kojima se definira da li će se vršiti čitanje (*read*), pisanje (*write*) ili dodavanje (*append*)
- `'b'` označava binarni oblik datoteke. Ako nije specificiran onda je oblik datoteke takstualni (`'t'`).
- `'+'` omogućuje istodobno čitanje i pisanje (ili dodavanje) datoteke.

Početna pozicija unutar datoteke razlikuje se ovisno o opciji `'r'`, `'w'` (početak datoteke) ili `'a'` (kraj datoteke).

Nakon obavljanja željenih manipulacija, koje su objašnjene u narednim poglavljima, datoteku je potrebno zatvoriti. Za zatvaranje datoteke koristi se `close` metoda koju ima svaki `file` objekt:

```
f.close()
```

Nakon zatvaranja nisu moguće nikakve operacije na datoteci.

1.11.2 Pisanje u datoteke

Za zapisivanje informacija u datoteku koriste se metode `write` i `writelines` klase `file`.

`write` metoda prima string varijablu koju zapisuje u prethodno otvorenu datoteku u modu za pisanje (`'w'`) ili dodavanje (`'a'`). Ovu funkciju može se ponavljati kako bi se dodalo još teksta u datoteku.

```
f.write(tekst)
```

Python kod 1.29 Primjer kreiranja tekstualne datoteke

```
datoteka = open('info.txt', 'w')
```

```
datoteka.write('Tehnički fakultet')
datoteka.write(', Rijeka')
datoteka.write('\nVukovarska 58')

datoteka.close()
```

```
Tehnički fakultet, Rijeka
Vukovarska 58
```

Funkcija `writelines` zapisuje listu stringova u datoteku. Postignuti efekt je isti kao da se pozove `write` za svaki element liste. Iako njeno ime na to upućuje, `writelines` ne zapisuje svaki element liste u novi redak, već je za novi redak potrebno umetnuti `\n`.

```
f.writelines(lista_stringova)
```

```
l = ['malo', ' ', 'teksta']

f.writelines(l)

# Isto što i
for s in l:
    f.write(s)
```

1.11.3 Čitanje iz datoteka

Za čitanje datoteke potrebno je željenu datoteku otvoriti u modu za čitanje (`'r'`). Osnovni način za pročitati sadržaj određene datoteke je preko funkcija:

- `read` koja vraća sadržaj datoteke (od trenutne pozicije) željene veličine u obliku stringa,
- `readline` koja vraća sadržaj trenutne linije u obliku stringa,
- `readlines` koja vraća listu stringova koji predstavljaju pojedine linije datoteke

Sintaksa metode `read` je:

```
f.read(size)
```

gdje je `size` veličina sadržaja kojeg treba pročitati. Za binarnu datoteku `size` označava broj bitova, a za tekstualnu broj znakova. Ukoliko je size negativan ili nije zadan, čita se cijeli sadržaj datoteke. Kod čitanja datoteka vrlo često je bitna trenutna pozicija u samoj datoteci što je objašnjeno u poglavlju 1.11.4.

```
f.readline(size)
```

Metoda `readline` čita jednu liniju u datoteci i to od trenutne pozicije do kraja linije (do znaka `'\n'`). Argument `size` definira maksimalnu veličinu pročitanih podataka. Za prazne linije metoda vraća string koji sadrži znak za novu liniju `'\n'`. Kada je trenutna pozicija na kraju datoteke, metoda vraća prazan string.

```
f.readlines(size)
```

`readlines` vraća sadržaj datoteke kao listu linija. `size` ima jednak učinak kao i na dvije prethodno objašnjene metode.

Za potrebe primjera, zadana je datoteka *test.txt* sljedećeg sadržaja:

```
123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
3. linija
4. linija
zadnja!
```

U izvornom kodu 1.30 dani su primjeri upotrebe navedenih funkcija za čitanje sadržaja datoteke.

Python kod 1.30 Čitanje sadržaja tekstualne datoteke

```
data = open('test.txt', 'r')

print(data.read(5))
print(data.readline())
print(data.readline(6))
print(data.readlines())

data.close()
```

```
12345
6789

ABCDEF
['GHIJKLMNOPQRSTUVWXYZ\n', '3. linija\n', '4. linija\n', 'zadnja!']
```

Osim pomoću navedenih funkcija, postoji još jedan način pristupanju linijama neke tekstualne datotetke, a to je iterirajući kroz file objekt kao što je prikazano u izvornom kodu 1.31.

Python kod 1.31 Pristupanje sadržaju datoteke preko iteriranja po `file` objektu

```
data = open('test.txt', 'r')
s = ''
for line in data:
    s += line
print(s)
data.close()
```

```
123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
3. linija
4. linija
zadnja!
```

1.11.4 Pozicioniranje u datoteci

Trenutnu poziciju u datoteci može se saznati preko metode `tell`:

```
f.tell()
```

Metoda `tell` daje broj bitova (znakova) trenutne pozicije od početka datoteke (Izvorni kod 1.32).

`seek` omogućuje micanje trenutne pozicije u datoteci. Sintaksa metode je:

```
f.seek(offset, from)
```

`offset` je broj kojim definiramo pomak pozicije. Pozitivan je u naprijed, a negativan u nazad. Argumentom `from` se definira od kuda će se pomak računati (0 za početak datoteke, 1 za trenutni položaj i 2 za kraj datoteke).

Python kod 1.32 Dohvaćanje trenutne pozicije u datoteci

```
data = open('test.txt', 'r')

print(data.readline())
print('pozicija:', data.tell())
print(data.read(3))
print('pozicija:', data.tell())
data.seek(6, 1)
print(data.read(3))
data.seek(-3, 1)
print(data.read(5))
data.seek(0, 0)
print(data.read(5))
data.seek(-5, 2)
print(data.read(5))
data.close()
```

```
123456789

pozicija: 10
ABC
pozicija: 13
JKL
JKLMN
12345
dnja!
```

1.11.5 Preimenovanje, kopiranje i brisanje datoteka

Python kod 1.33 Kopiranje, micanje i preimenovanje datoteka

```
# simply
import os
os.rename('a.txt','b.kml')

# or
import shutil
shutil.move('a.txt', 'b.kml')

# or if you want to copy..
import shutil
shutil.copy('a.txt', 'b.kml')
```

Paket `os` omogućuje manipulaciju datotekama na disku. Neke od korisnih naredbi za stvaranje direktorija i brisanje su:

- `makedirs()` omogućuje stvaranje direktorija
- `getcwd()` vraća tekući direktorij u obliku stringa

- `chdir()` mijenja tekući direktorij
- `rename()` omogućuje preimenovanje datoteke
- `removedirs()` briše prazne direktorije
- `remove()` briše datoteku

Python kod 1.34 Stvaranje datoteka i direktorija

```
import os

direktorij = os.getcwd()
os.makedirs(direktorij + '/novidir')
os.chdir(direktorij + '/novidir')
f = open('novi.txt', 'w')
f.close()
```

S brisanjem direktorija i datoteka moramo biti malo više oprezni pa `removedirs()` funkcija vraća pogrešku ako pokušamo obrisati direktorij koji nije prazan.

Python kod 1.35 Preimenovanje i brisanje datoteka i direktorija

```
import os

os.chdir(os.getcwd() + '/novidir')
direktorij = os.getcwd()
print(direktorij)
os.rename('novi.txt', 'novinovi.txt')
os.remove('novinovi.txt')
os.removedirs(direktorij)
```

Modul `shutil` omogućava naprednije operacije s datotekama tj. s grupom datoteka.

1.11.6 Arhiviranje

Modul `shutil` sadrži funkciju `make_archive` koja omogućava kreiranje arhive datoteka. Funkcija se poziva:

```
shutil.make_archive(archive_name, format, root_dir)
```

Gdje `archive_name` predstavlja naziv arhiva koji će se kreirati, argument `format` definira format arhiva.

```
from shutil import make_archive
import os
archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

1.12 Python Standard Library

Bojan Crnković
Stefan Ivić

Instalacija Pythona dolazi sa standardnom knjižnicom koja sadrži razne module i funkcionalnosti koju nije potrebno posebno instalirati (vidi poglavlje 1.1.3). Primjeri najčešće korištenih modula iz standardne knjižnice navedeni su u sljedećim potpoglavljima.

1.12.1 Matematičke funkcije

Python modul `math` sadrži standardne matematičke funkcije za realne brojeve koje su definirane C standardom, s kojima se najčešće susrećemo prilikom pisanja nekog programa.

Tablica 1.7
Opis nekih funkcija `math` modula

Funkcija	Opis
`math.fabs(x)`	Funkcija vraća apsolutnu vrijednost realnog broja x.
`math.fsum(niz)`	Funkcija vraća sumu elemenata nekog niza brojeva.
`math.isnan(x)`	Funkcija vraća `True` ako je x tipa `NaN`.
`math.exp(x)`	Exponencijalna funkcija e^x.
`math.log(x, b)`	Funkcija $\log_b(x)$ odnosno $\ln(x)$ ako se ne zada b.
`math.cos(x)`	Funkcija vraća kosinus kuta x zadanog u radijanima.
`math.acos(x)`	Inverz funkcije kosinus.
`math.e`	Matematička konstanta e=2.718281828459045...

Primjer upotrebe nekih funkcija `math` modula:

```
from math import log, e, fsum
print(log(e))
print(fsum([1,2,3]))
```

```
1.
6.
```

Python modul `cmath` sadrži standardne matematičke funkcija nad poljem kompleksnih brojeva. Argumenti funkcija mogu biti tipa `float` ili `complex`.

Tablica 1.8
Opis nekih funkcija `cmath` modula

Funkcija	Opis
`cmath.polar(x)`	Funkcija vraća polarnu (r,ϕ) reprezentaciju kompleksnog broja x.
`cmath.sqrt(x)`	Funkcija vraća korijen kompleksnog broja

Primjer upotrebe nekih funkcija `cmath` modula:

```
from cmath import sqrt
sqrt(1.+2.j)
```

```
(1.272019649514069+0.7861513777574233j)
```

Python modul `decimal` omogućuje računanje i pravilno zaokruživanje realnih brojeva u aritmetici s proizvoljnim brojem decimalnih mjesta.

Primjer upotrebe nekih funkcija `decimal` modula:

```
from decimal import *
getcontext().prec = 6
print Decimal(1) / Decimal(3)
getcontext().prec =30
print Decimal(1) / Decimal(3)
```

```
0.333333
0.333333333333333333333333333333
```

1.12.2 Vrijeme izvršavanja

Učinkovitost određenog algoritma, djela izvornog koda ili pak cijelog izvornog koda vrlo je bitan faktor ukupne kvalitete računalnog programa. Učinkovitost određenog izvornog koda se, prije svega, može mjeriti vremenom izvođenja tog koda.

Treba imati na umu da vrijeme izvođenja ovisi o hardverskim resursima računala pa su stoga rezultati različiti ovisn o računalu na kojem se testiranje vrši. Vrijeme izvršavanja ovisi i o trenutnom opterećenju hardverskih resursa (operativni sustav, sistemski servisi i pokrenute aplikacije), pa će, u manjoj mjeri, testiranja na istom računalu biti različita.

Vrlo elegantan način za mjerenje vremena izvršavanja određenog djela koda je upotrebom funkcija `clock` i `time` iz modula `time`. Funkcija `time` daje trenutno vrijeme u sekundama od 00:00 1. siječnja 1970. godine. Funkcija `clock` je preciznija i primjerenija mjerenju vremena izvršavanja, ali različito funkcionira na Linux (Unix) i MS Windows platformama:

- na Linux platofrmi, `time.clock()` daje utrošeno procesorsko vrijeme, tj. koliko je sekundi procesor bio opterećen,
- na MS Windows platformi, `time.clock()` daje "ukupno" utrošeno vrijeme koje obuhvaća vrijeme računanja (procesor) ali i utrošeno vrijeme na čitanje datoteka (disk) ili npr. utrošeno vrijeme na uspostavljanje konekcije preko nekog mrežnog protokola.

`time.clock()` ili `time.time()` se mogu pozivati više puta te na taj način mjeriti vrijeme izvršavanja više djelova izvornog koda. Kod mjerenja izvršavanja koda koji radi sa čitanje ili pisanjem datoteka ili koji vrši komunikaciju preko mrežnih protokola, rezultati će biti ovisni i o platformi na kojoj se kod izvršava.

Python kod 1.36 Primjer mjerenja vremena izvršavanje algoritma

```
import time

# dohvati vrijeme početka
start_c = time.clock()
start_t = time.time()
print('Start clock: %.5f' % start_c)
print('Start time: %.5f' % start_t)
```

```
# algoritam
a = 0.0
for i in range(1000):
    for j in range(i):
        a = a + (i - j)**0.3

# dohvati vrijeme završetka
stop_c = time.clock()
stop_t = time.time()
print('Stop clock: %.5f' % stop_c)
print('Stop time: %.5f' % stop_t)

print('Rezultat: %.2f' % a)
print('Vrijeme izvrsavanja (clock): %.3f ms' % (stop_c - start_c))
print('Vrijeme izvrsavanja (time): %.3f ms' % (stop_t - start_t))
```

```
Start clock: 0.01000
Start time: 1373797428.12780
Stop clock: 0.12000
Stop time: 1373797428.23620
Rezultat: 2656321.74
Vrijeme izvrsavanja (clock): 0.110 ms
Vrijeme izvrsavanja (time): 0.109 ms
```

Python Standard Library sadrži klasu `Timer` koja omogućuje mjerenje vremena izvršavanja Python izraza.

Python kod 1.37 Primjer upotrebe `Timer` klase za mjerenje vremena izvođenja funkcije

```
import timeit
from math import sqrt

# algoritam
def f(x):
    s = 0.0
    for i in range(x):
        s = s * sqrt(s+x)

t = timeit.Timer(stmt='f(500)', setup='from __main__ import f')

r1 = t.timeit(number=1000)
print('Potrebno vrijeme za 1000 poziva funkcije f(500): %.5f' % r1)

r5 = t.repeat(number=1000, repeat=5)
print('\nPonavljanje testa (5 puta):')
for r in r5:
    print('Potrebno vrijeme za 1000 poziva funkcije f(500): %.5f' % r)
```

```
Potrebno vrijeme za 1000 poziva funkcije f(500): 0.05833

Ponavljanje testa (5 puta):
Potrebno vrijeme za 1000 poziva funkcije f(500): 0.05900
Potrebno vrijeme za 1000 poziva funkcije f(500): 0.05927
Potrebno vrijeme za 1000 poziva funkcije f(500): 0.05888
Potrebno vrijeme za 1000 poziva funkcije f(500): 0.05953
Potrebno vrijeme za 1000 poziva funkcije f(500): 0.05925
```

1.12.3 Datum i vrijeme

Pristupanje trenutnom datumu i vremenu moguće je na više načina, ovisno o formatu koji želimo. Primjerice, izvršavanjem koda gdje se funkciji `localtime` kao argument daje trenutno vrijeme u sekundama dobiveno naredbom `time`:

```
import time
vrijeme=time.localtime (time.time())
print (vrijeme)
```

dobiva se trenutni datum i vrijeme u sljedećem formatu.

```
time.struct_time(tm_year=2017, tm_mon=11, tm_mday=3, tm_hour=9, tm_min=49,
                                 tm_sec=33, tm_wday=4, tm_yday=307, tm_isdst=0)
```

Formatiranja teksta moguće je napraviti pozivanjem `asctime`:

```
import time
vrijeme=time.asctime(time.localtime (time.time()))
print (vrijeme)
```

gdje se sada trenutni datum i vrijeme dobivaju u sljedećem formatu.

```
Fri Nov  3 10:00:36 2017
```

1.13 Greške u Python kodu

Marko Čavrak

S obzirom na to da je Python interpreterski jezik, nailaskom na prvu grešku u kodu program se zaustavlja s izbacivanjem tekstualne informacije o istoj. Dio programskog koda koji se nalazi prije greške se izvršava u cijelosti, dok se dio koda nakon greške ne izvršava. Tipovi grešaka kao i najčešće greške objašnjeni su u sljedećim potpoglavljima.

1.13.1 Tipovi grešaka

Većina programskih jezika posjeduje određene usluge prema korisniku koje omogućuju identificiranje i manipulaciju grešaka u pisanom izvornom kodu te davaju korisna objašnjenja vezana uz izvor greške ili način kako ih ukloniti. Python slijedi takvu logiku i omogućuje prijavljivanje dva tipa grešaka:

- Sintaksne greške *Syntax Error* i
- Iznimke (eng. *Exceptions*)

Obje greške utvrđuju se isključivo tijekom izvođenja tj. interpretiranja koda: sintaksne greške na početku izvođenja koda, a iznimke za svaki izvršeni red koda. Nakon što Python identificira prvu grešku na koju naiđe program stane sa izvršavanjem i objavi poruku slijedećeg izgleda

```
>>> print(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'c' is not defined
```

Traceback obavještava korisnika o tome gdje je greška pronađena i daje opis greške pomoću sintaksne greške ili iznimke (eng. *exception*) koja objašnjava razlog prekida izvršavanja programa. U navedenom primjeru **NameError** je iznimka koja opisuje grešku pri pozivu *imena* `'c'`.

Sintaksne greške testiraju svaku liniju koda netom prije izvođenja cijelog programa i utvrđuju svaku nepravilnost u sintaksi programskog jezika. Npr.

```
j = 1

if j < 2:
    print('Help')
else:
    if True
        print('Help2')
```

```
  File "/home/mcavrak/.spyder2/.temp.py", line 7
    if True
           ^
SyntaxError: invalid syntax
```

prilikom izvođenja ovog koda program nikada niti neće ući u *else* jer je uvjet *if* naredbe takav da vraća *True*, međutim program svejedno vraća grešku prilikom provere sintakse (`'nedostaje dvotočka iza True'`)

No, iznimke će program objaviti samo ako naiđe na njih.

Primjer kada naiđe na grešku (pogrešno upisana varijabla 'i', trebalo je 'j'):

```
j = 1

if j < 2:
    print('Help')
    print(i)
```

```
Help
Traceback (most recent call last):
  File "/home/mcavrak/.spyder2/.temp.py", line 6, in <module>
    print i
NameError: name 'i' is not defined
```

Primjer kada ne naiđe na grešku:

```
j = 1

if j < 2:
    print('Help')
else:
    print(i)
```

```
Help
```

Najčešće iznimke koje susrećemo su:

- **ZeroDivisionError**

  pokušaj operacije djeljenja sa nulom

  ```
  >>> 10 * (1/0)
  Traceback (most recent call last):
    File "<stdin>", line 1, in ?
  ZeroDivisionError: division by zero
  ```

- **NameError**

  pokušaj operacije sa varijablom čije ime (*name*) nije u popisu imena (*namespace*)

  ```
  >>> 4 + i*3
  Traceback (most recent call last):
    File "<stdin>", line 1, in ?
  NameError: name 'i' is not defined
  ```

- **TypeError**

  Pokušaj izvršavanja operacije na krivom tipu podataka

  ```
  >>> '2' + 2
  Traceback (most recent call last):
    File "<stdin>", line 1, in ?
  TypeError: must be str, not int
  ```

Postoji još mnogo iznimaka te ih je moguće pronaći na http://docs.python.org/3/library/exceptions.html#bltin-exceptions.

No, iznimkama možemo manipulirati, tj. moguće je ugraditi u kod identifikaciju greške te djelovanje sukladno tome. O tome više u slijedećem poglavlju.

1.13.2 Manipulacija greškama

Iznimkama možemo manipulirati (eng. *exception handling*), tj. možemo prihvatiti informaciju o greški te djelovati u skladu sa njom ili granati kod tako da nastavi sa izvršavanjem unatoč greški. Razmotrimo primjer kada korisnik unosi brojeve na konzoli pomoću *input* funkcije.

```
var = input('Unesi broj:')
a = int(var)
print(a)
```

Python vraća iznimku **ValueError** jer nije u stanju pretvoriti preuzeti string u cjelobrojnu vrijednost.

```
Unesi broj:z
Traceback (most recent call last):
  File "/home/flood/untitled0.py", line 9, in <module>
    a = int(var)
ValueError: invalid literal for int() with base 10: 'z'
```

Kako bismo izbjegli takvu grešku i prihvatili iznimku poslužiti ćemo se sa *try - except* formulacijom:

```
var = input('Unesi broj:')

try:
    a = int(var)
    print('SUCCESS')
except ValueError:
    print('Nije unešen broj!')
    print('Broj postavljen na defaultnu vrijednost 0.')
    a = 0

print(a)
```

U ovom kodu izvršen je prihvat iznimke na način da je:

- unešena ključna riječ *try* čije se tijelo prvo izvršava
- ako program ne naiđe na iznimku tada program preskače dio tijela koji pripada *except* sekciji i nastavlja sa kodom (`'print a'`).
- ako program naiđe na iznimku u *try* sekciji koda tada se zaustavlja na mjestu gdje je našao grešku, ne nastavlja dalje sa preostalim djelom koda (`'print SUCCESS'`) već skače na *except* sekciju koja odgovara iznimki koju je uhvatio (u ovom slučaju **ValueError**) te potom nastavlja sa kodom (`'print a'`)
- ukoliko program ne naiđe na vrstu iznimke opisanu u except sekciji tada izbaci grešku u obliku *neuhvaćene iznimke* (eng. *unhandled exception*)

Pored ovog osnovnog koncepta primjene prihvata iznimaka, moguće je podići iznimku prema korisničkoj želji te izraditi korisnički definirane iznimke. Ove funkcionalnosti nadilaze osnove i čitatelj je slobodan potražiti više informacija na https://docs.python.org/3.6/tutorial/errors.html.

1.13.3 Ispravljanje grešaka

Python ima modul `pdb` („Python DeBugger") koji pomaže kod otkrivanja i ispravljanja grešaka u kodu skripte. Ako ne koristite neki specijalizirani editor unutar nekog

razvojnog okruženja tada je potrebno pozvati pdb modul na početku skripte:

```
import pdb
```

Na nekom ključnom mjestu u kodu na kojem se želite zaustaviti morate staviti funkciju:

```
pdb.set_trace()
```

na primjeru 1.38 može se vidjeti upotreba pdb modula.

Ako koristite editor Spyder tada nije nužno pozvati pdb modul. Dovoljno je postaviti prekide u kodu dvostrukim lijevim klikom na lijevoj strani dokumenta i pokrenuti skriptu u Debug načinu rada.

```
(Pdb)
```

Unutar pdb modula moguće je kontrolirati izvršavanje koda, a najčešće se koriste naredbe:

- n (*next*) izvršava se sljedeća naredba u skripti,
- c (*continue*) izvrešava se ostatak skripte do sljedećeg prekida,
- s (*step into*) pdb ulazi u podprogram ili funkciju,
- r (*return*) slično kao naredba c, ali pdb izvršava ostatak podprograma ili funkcije do izlaza iz podprograma,
- p var (*print*) pdb istpis stanja varijable var
- ENTER pdb izvršava zadnju naredbu koja mu je poslana,
- q (*quit*) zaustavlja izvršavanje skripte.

Python kod 1.38 Upotreba dbg modula

```
import pdb

pdb.set_trace()
a = 0
b = 1
c = 2
pdb.set_trace()
a = b + c
pdb.set_trace()
a = b - c
print('kraj')
```

2. NumPy

Zadavanje polja u NumPy-u
Informacije o polju i indeksiranje
Manipulacije s NumPy poljima
Učitavanje i spremanje NumPy polja
Matrični račun
Rad s polinomima

2.1 Zadavanje polja u NumPy-u

Stefan Ivić
Miran Tuhtan

Bazni objekt u NumPy-u, `ndarray`, je homogeno multidimenzionalno polje. Homogenost polja podrazumijeva da su svi podaci u polju istog tipa. U NumPy-u, dimenzije se nazivaju `axes`, a broj dimenzija `rank`. Elementi polja indeksirani su nenegativnim cijelim brojevima, počevši od nule.

NumPy polja mogu se zadavati na više načina. Bazično, zadavanje polja može se podjeliti na ručni unos pomoću funkcije `array` te na automatsko generiranje polja. NumPy polja mogu se automatski generirati pomoću neke od naredbi za generiranje polja.

2.1.1 Naredba array

Funkcija `array` omogućava kreiranje NumPy polja iz običnih Python listi ili tuple-ova. Prilikom kreiranja polja pomoću `array` funkcije, u argument funkcije upisuju se elementi matrice i to na način da svaka dimenzija započinje otvorenom uglatom zagradom (`[`) i završava zatvorenom uglatom zagradom (`]`) dok su elementi razdvojeni zarezom.

Najjednostavnija sintaksa funkcije `array` je:

```
numpy.array(object)
```

Ako je argument funkcije `array` tuple, onda pojedina dimenzija započinje i završava sa zagradama. Pri tome treba paziti da je argumnet funkcije `array` samo jedan tuple (cijeli tuple mora biti u jednim zagradama). Moguće je kombinirati Python liste i tuple-ove prilikom zadavanja matrice.

```
>>> import numpy as np
>>> M1 = np.array([[1, 2, 3],[4, 5, 6]])
>>> M1
array([[1, 2, 3],
[4, 5, 6]])
>>> M2 = np.array(((6, 7, 8), (9, 10, 11)))
>>> M2
array([[ 6,  7,  8],
[ 9, 10, 11]])
>>> M3 = np.array(([6, 7, 8], (9, 10, 11)))
>>> M3
array([[ 6,  7,  8],
[ 9, 10, 11]])
```

Dodatni, neobavezni argumenti funkcije `array` su:

- `dtype` - željeni tip podataka za elemente polja. Ako argument nije zadan, elementi polja će preuzeti minimalni tip potreban za pohranu podataka iz objekta `object`. Ovaj argument se može koristiti samo za 'povećati' (eng. *upcast*) tip zadanog objekta. Za 'smanjiti' (eng. *downcast*) tip treba koristiti `.astype(t)` naredbu.
- `copy` - bool varijabla (`True` ili `False`) koja određuje kopira li se objekt.
- `order` - određuje način na koji se polje pohranjuje u memoriji. Polje može biti pohranjeno na tri načina:

- `'C'` - elementi polja će biti poredani u memoriji na način da zadnji indeks varira najbrže (C-povezani poredak)
- `'F'` - elementi polja će biti poredani u memoriji na način da prvi indeks varira najbrže (Fortran-povezani poredak)
- `'A'` - elementi polja neće imati definiran poredak u memoriji (C-povezani poredak, Fortran-povezani poredak ili čak nepovezani poredak)

`subok` - bool varijabla (`True` ili `False`) kojom se definira hoće li kreirano polje poprimiti tip `numpy` podklase (`True`) iz koje se uzimaju vrijednosti ili će poprimiti tip bazne `numpy.array` klase. Zadana vrijednost `subok` varijable je `False`.

`ndmin` - minimalni broj dimenzija generiranog polja. Prema potrebi, broj dimenzija će biti povećan na način da će jedinice biti dodane u `shape` polja.

2.1.2 Naredba arange

Funkcija `arange` omogućava kreiranje jednodimenzionalnog polja koje sadrži aritmetički niz definiran prvim članom, zadnjim članom te korakom.

Sintaksa funkcije je

```
numpy.arange(start, stop, step, type)
```

gdje je `start` početak niza, `stop` kraj niza, `step` korak niza, a `type` je tip podataka sadržanih u nizu.

> **Napomena**
> Oprez! Prilikom korištenja ne-cjelobrojnih koraka niza rezultat funkcije `arange` može biti nekonzistentan. U takvim slučajevima bolje je koristiti funkciju `linspace` (2.1.3).

Svi argumennti funkcije `arange`, osim `stop`, su neobavezni i ukoliko nisu uneseni imaju zadane vrijednosti:

- `start` ima zadanu vrijednost `0`
- `step` ima zadanu vrijednost `1` te zahtijeva i zadavanje argumenta `start`
- `type` ako nije zadan, postavlja se tipa na temelju ostalih argumenata.

```
>>> import numpy as np
>>> M1 = np.arange(10, 30, 3)
>>> M1
array([10, 13, 16, 19, 22, 25, 28])
>>> M2 = np.arange(-2, 3)
>>> M2
array([-2, -1,  0,  1,  2])
>>> M3 = np.arange(5)
>>> M3
array([0, 1, 2, 3, 4])
```

2.1.3 Naredba linspace

Funkcija `linspace` ima sličnu namjenu kao i funkcija `arange` (kreiranje aritmetičkog niza) samo što se umjesto koraka niza zadaje broj članova niza.

I sintaksa funkcije je vrlo slična funkciji `arange`:

```
numpy.linspace(start, stop, num=50, endpoint=True, retstep=False)
```

gdje je `start` početak niza, `stop` kraj niza, `num` broj elemenata niza, `endpoint` je opcija uključivanja zadnjeg elementa a `retstep` je opcija vraćanja korištenog koraka niza.

```
>>> import numpy as np
>>> M1 = np.linspace(0, 10, 21)
>>> M1
array([  0. ,   0.5,   1. ,   1.5,   2. ,   2.5,   3. ,   3.5,   4. ,
4.5,   5. ,   5.5,   6. ,   6.5,   7. ,   7.5,   8. ,   8.5,
9. ,   9.5,  10. ])
>>> M2 = np.linspace(0, 5, 10, endpoint=False)
>>> M2
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5])
>>> M3, step = np.linspace(0, 1, retstep=True)
>>> M3
array([ 0.        ,  0.02040816,  0.04081633,  0.06122449,  0.08163265,
0.10204082,  0.12244898,  0.14285714,  0.16326531,  0.18367347,
0.20408163,  0.2244898 ,  0.24489796,  0.26530612,  0.28571429,
0.30612245,  0.32653061,  0.34693878,  0.36734694,  0.3877551 ,
0.40816327,  0.42857143,  0.44897959,  0.46938776,  0.48979592,
0.51020408,  0.53061224,  0.55102041,  0.57142857,  0.59183673,
0.6122449 ,  0.63265306,  0.65306122,  0.67346939,  0.69387755,
0.71428571,  0.73469388,  0.75510204,  0.7755102 ,  0.79591837,
0.81632653,  0.83673469,  0.85714286,  0.87755102,  0.89795918,
0.91836735,  0.93877551,  0.95918367,  0.97959184,  1.        ])
>>> step
0.02040816326530612
```

2.1.4 Naredba zeros

Funkcija `zeros` omogućava generiranje polja koje sadrži nule. Osnovna sintaksa funkcije `zeros` je

```
numpy.zeros(shape)
```

gdje je `shape` lista koja sadrži veličine dimenzija.

Osim veličine dimnezija, postoje i dodatni neobavezni argumenti za definiranje tipa podataka te načina spremanja polja u memoriju:

```
numpy.zeros(shape, dtype=float, order='C')
```

Za generiranje polja nula može poslužiti i funkcija `zeros_like` koja generira polje nula istih dimenzija kao neko postojeće polje. Sintaksa funkcije je:

```
numpy.zeros_like(a)
```

gdje je `a` matrica čije dimenzije treba preuzeti.

> **Napomena**
> Funkcija `zeros_like` ima dodatne argumente `dtype` i `order`. `dtype` omogućuje nametanje tipa podataka za generirano polje, a ako nije specificiran onda generirano polje nasljeđuje tip podataka od argumenta `a`. `order` omogućuje mijenjanje rasporeda spremanja polja u memoriju.

Isti se rezultat može postići i na druge načine:

```
>>> import numpy as np
>>> a = np.array([0, 2, 5, 1])
>>> b = np.zeros(a.shape)
>>> print(a)
[0 2 5 1]
>>> print(b)
[ 0.  0.  0.  0.]
```

```
>>> import numpy as np
>>> a = np.zeros([3, 5])
>>> print(a)
[[ 0.  0.  0.  0.  0.]
[ 0.  0.  0.  0.  0.]
[ 0.  0.  0.  0.  0.]]
>>> b = np.array([2, 5, 6, 0])
>>> print(b)
[2 5 6 0]
>>> c = np.zeros_like(b)
>>> print(c)
[0 0 0 0]
```

2.1.5 Naredba ones

Za generiranje matrice zadanih dimenzija koja sadrži jedinice koristi se funkcija `ones`. Sintaksa funkcije je ista kao i kod funkcije `zeros`:

```
numpy.ones(shape, dtype=float, order='C')
```

Za kreiranje matrice, koja sadrži jedinice istih dimenzija kao neka postojeća matrice, koristi se `ones_like` funkcija:

```
numpy.ones_like(a)
```

gdje je `a` matrica čije dimenzije treba preuzeti.

Napomena
Funkcija `ones_like` ima dodatne argumente `dtype` i `order` koji funkcioniraju jednako kao što je opisano za funkciju `zeros_like`.

```
>>> import numpy as np
>>> a = np.ones([2, 4])
>>> print(a)
[[ 1.  1.  1.  1.]
[ 1.  1.  1.  1.]]
>>> b = np.ones([3, 2], dtype=int)
>>> print(b)
[[1 1]
[1 1]
[1 1]]
>>> c = np.array([5, 8])
>>> d = np.ones_like(c)
>>> print(d)
[1 1]
```

2.1.6 Naredba eye

Generiranje dvodimenzionalne jedinične matrice (sadrži jedinice na dijagonali, a ostalo su nule) moguće je s funkcijom `eye`.

> **Napomena**
> Funkcija `eye` omogućuje generiranje isključivo dvodimenzionalnih jediničnih matrica.

Potpuna sintaksa funkcije `eye` je:

```
numpy.eye(N, M=None, k=0, dtype=<class 'float'>)
```

gdje je `N` broj redaka generiranog polja, `M` broj stupaca generiranog polja (ako nije zadano, broj stupaca jednak je broju redaka), `k` indeks dijagonale (0 je glavna dijagonala, pozitivna vrijednost se odnosi na dijagonale iznad, a negativna na dijagonale ispod glavne dijagonale) dok je `dtype` tip podataka generiranog polja.

```
>>> import numpy as np
>>> a = np.eye(4)
>>> print(a)
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]
 [ 0.  0.  0.  1.]]
>>> b = np.eye(5, 3)
>>> print(b)
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
>>> c = np.eye(3, 5, 1)
>>> print(c)
[[ 0.  1.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.]
 [ 0.  0.  0.  1.  0.]]
```

2.1.7 Naredba diag

Često su brojevi na dijagonali matrice važni u proračunu i javlja se potreba da ih se spremi u zasebnu varijablu. Funkcija `diag` omogućuje upravo to: generira vektor koji sadrži elemenate dijagonale zadane matrice.

Sintaksa funkcije vrlo je jednostavna. Funkcija `diag` prima kao argument 2D matricu iz koje se želi preuzeti dijagonala. Dodatni argument, `k`, specificira položaj (indeks) dijagonale: 0 je glavna dijagonala, pozitivna vrijednost se odnosi na dijagonale iznad a negativna na dijagonale ispod glavne dijagonale.

```
numpy.diag(v, k=0)
```

```
>>> import numpy as np
>>> M1 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> print(M1)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

```
>>> print(np.diag(M1))
[1 5 9]
>>> print(np.diag(M1, 1))
[2 6]
```

Ako se funkciji `diag` kao argument zada 1D vektor, tada funkcija vraća 2D matricu kojoj je zadani vektor dijagonala.

```
>>> import numpy as np
>>> a = np.array([7, 2, 3, 11])
>>> np.diag(a)
array([[ 7,  0,  0,  0],
       [ 0,  2,  0,  0],
       [ 0,  0,  3,  0],
       [ 0,  0,  0, 11]])
```

2.1.8 Naredba meshgrid

Funkcija `meshgrid` služi za generiranje matrice koordinata na temelju vektora koordinata. Implementacija funkcije `meshgrid` se vrši na način:

```
numpy.meshgrid(x, y, sparse, indexing)
```

gdje su argumenti `x` i `y` jednodimenzionalna NumPy polja iz kojih će se kreirati dvije NumPy matrice jednakih dimenzija. Elementi s jednakim indeksima u dvije kreirane NumPy matrice zajedno formiraju jednu koordinatnu točku. Argument `sparse` poprima vrijednost boolean tipa (`True` ili `False`). U slučaju da je `sparse` zadan kao `True` kreiraju se samo prvi red odnosno prvi stupac za `x` i `y` polja. Argument `indexing` poprima string vrijednost 'xy' ili 'ij' te određuje način indeksiranja kreirane matrice.

```
>>> import numpy as np
>>> x = np.linspace(0, 5, 6)
>>> y = np.linspace(10, 50, 5)
>>> X, Y = np.meshgrid(x, y)
>>> print(x)
[ 0.  1.  2.  3.  4.  5.]
>>> print(y)
[ 10.  20.  30.  40.  50.]
>>> print(X)
[[ 0.  1.  2.  3.  4.  5.]
 [ 0.  1.  2.  3.  4.  5.]
 [ 0.  1.  2.  3.  4.  5.]
 [ 0.  1.  2.  3.  4.  5.]
 [ 0.  1.  2.  3.  4.  5.]]
>>> print(Y)
[[ 10.  10.  10.  10.  10.  10.]
 [ 20.  20.  20.  20.  20.  20.]
 [ 30.  30.  30.  30.  30.  30.]
 [ 40.  40.  40.  40.  40.  40.]
 [ 50.  50.  50.  50.  50.  50.]]
```

`meshgrid` je vrlo koristan kod evaluacije funkcija dviju varijabli na mreži točaka, što se najčešće koristi pri vizualizaciji:

```
>>> import numpy as np
>>> x = np.linspace(0, 3, 4)
```

```
>>> y = np.linspace(0, 5, 6)
>>> xx, yy = np.meshgrid(x, y, sparse=True)
>>> z = np.cos(xx + yy**2) - np.sin(xx**2 - yy)
>>> z
array([[ 1.        , -0.30116868,  0.34065566, -1.40211098],
[ 1.38177329, -0.41614684, -1.1311125 , -1.64300187],
[ 0.25565381,  1.12513317,  0.05087286,  0.09691566],
[-0.77001025,  0.0702259 , -0.83704529,  1.12326946],
[-1.71446198, -0.13404333,  0.66031671,  1.94762889],
[ 0.03227854, -0.10988317,  0.54933218, -0.20580337]])
```

2.2 Informacije o polju i indeksiranje

Stefan Ivić
Miran Tuhtan

Pristupanje elementima NumPy polja je vrlo česta radnja u računarskom inženjerstvu te je to fokus u ovom potpoglavlju. Za pristupanje pojedinim elementima NumPy polja potrebno je znati indekse polja, a za indeksiranje vrlo je bitno znati informacije poput broja elemenata i/ili oblika NumPy polja. Modul NumPy nudi mnogo alata kojima se može dobiti informacija o NumPy poljima.

2.2.1 Informacije o polju

Svako NumPy polje ima definiran broj dimenzija, koji se može provjeriti pomoću `ndim` atributa polja.

```
numpy.ndarray.ndim
```

ili pomoću funkcije `ndim`

```
numpy.ndim(ndarray)
```

gdje je `ndarray` numpy polje.

Za svaku dimenziju, polje ima rezerviran određen broj elemenata polja. Broj elemenata po dimenzijama polja čini niz koji se može dobiti preko atributa `shape`

```
numpy.ndarray.shape
```

ili preko funkcije `shape`

```
numpy.shape(ndarray)
```

gdje je `ndarray` numpy polje.

Ukupni broj elemenata u polju određuje se funkcijom `size`

```
numpy.size(ndarray)
```

ili atributom numpy polja

```
numpy.ndarray.size
```

gdje je `ndarray` numpy polje.

```
>>> import numpy as np
>>> M1 = np.array([[1, 2, 3], [4, 5, 6]])
>>> M1.ndim
2
>>> np.ndim(M1)
2
>>> M1.shape
(2, 3)
>>> np.shape(M1)
(2, 3)
>>> M1.size
6
```

```
>>> np.size(M1)
6
```

2.2.2 Indeksiranje polja

Pristupanje elementima NumPy polja vrši se pomoću indeksa na sličan način kao i s listama. Indeks označava poziciju elementa polja pomoću cijelog broja, počevši od nule. Kod jednodimenzionalnih polja (vektori), vrijednostima se pristupa preko jednog indeksa upisanog u uglate zagrade. Svaki sljedeći element vektora ima indeks za jedan veći od prethodnog, što znači da je indeks zadnjeg elementa jednak ukupnom broju elemenata vektora umanjenom za jedan. Svaki indeks koji je jednak ili veći broju elemenata vektora je indeks nepostojećeg elementa vektora i korištenje tog indeksa će prouzročiti grešku.

```
>>> import numpy as np
>>> V = np.array([6.0, 2.2, 3.0, 5.8, 5.4, 0.8])
>>> n = np.size(V)
>>> print(n)
6
>>> indeksi = np.arange(n)
>>> print(indeksi)
[0, 1, 2, 3, 4, 5]
>>> for i in indeksi:
...     print(V[i])
...
6.0
2.2
3.0
5.8
5.4
0.8
>>> print(V[6])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index out of bounds
```

Napomena
Indeksi u jednom paru uglatih zagrada (npr. `M[2, 4]`) su zapravo tuple, pa se pristupanju određenom elementa može vršiti i preko tuplea indeksa (`poz = (3, 1); print M[poz]`).

Indeksiranje višedimenzionalnih polja je analogno jednodimenzionalnim poljima, za svaku dimenziju se koristi po jedan indeks. Elementima polja se može pristupati s indeksima svake dimenzije upisanim u uglate zagrade ili s indeksima upisanim u jedne uglate zagrade, a razmaknutim zarezima.

Python kod 2.1 Manipulacije poljima pomoću indeksa

```
import numpy as np

M = np.array([[ 1,  2,  3,  4,  5],
              [ 6,  7,  8,  9, 10],
              [11, 12, 13, 14, 15],
              [16, 17, 18, 19, 20],
              [21, 22, 23, 24, 25]])
```

```
print(M[2][4])        # ispiši element na poziciji 2,4
poz = (3, 2)          # definiraj poziciju 3,2
print(M[poz])         # ispiši element na poziciji poz

print(M[1:-1, 1:-1])  # ispiši dio polja bez prvih i zadnjih redaka i stupaca
print(M[:, ::-1])     # ispiši polje s obrnutim redoslijedom stupaca
print(M[::2])         # Ispiši svaki drugi redak polja
```

```
15
18
[[ 7  8  9]
 [12 13 14]
 [17 18 19]]
[[ 5  4  3  2  1]
 [10  9  8  7  6]
 [15 14 13 12 11]
 [20 19 18 17 16]
 [25 24 23 22 21]]
[[ 1  2  3  4  5]
 [11 12 13 14 15]
 [21 22 23 24 25]]
```

2.3 Manipulacije s NumPy poljima

Stefan Ivić
Siniša Družeta

U ovom potpoglavlju predstavljene su funkcije kojima se može manipulirati NumPy poljima. Manipulacija poljima ovdje uključuje promjenu dimenzija polja, transponiranje, dodavanje i/ili brisanje elemenata polja te kopiranje polja. NumPy modul ima veliku količinu različitih funkcija kojima se navedene radnje mogu vrlo lagano postići.

2.3.1 Manipulacije s dimenzijama polja

Elementi NumPy polja određenih dimenzija, mogu se presložiti u NumPy polje drukčijih dimenzija. Preslagivanje polja na druge dimenzije vrši se pomoću funkcije `reshape`:

```
numpy.reshape(a, newshape, order='C')
```

gdje je `a` izvorno polje, `newshape` vektor novih dimenzija, a `order` je način spremanja polja u matricu.

Uvijet za reshape je da broj elemenata polja ostane isti, odnosno umnožak veličina dimenzija mora biti konstantan.

> **Napomena**
> Ukoliko se, kod poziva funkcije `reshape`, veličina jedna od dimenzija ne specificira, tada se ona automatski računa na temelju ostalih veličina dimenzija i ukupnog broja elemenata inicijalne matrice.

Specifični slučaj mijenjanja dimenzija polja je kada inicijalno polje želimo transformirati u jednodimenzionalno polje tj. vektor. Iako je to moguće napraviti s funkcijom `reshape`, postoji specijalizirana funkcija `ravel` za transformaciju n-dimnezionalnih polja u jednodimenzionalna polja. Sintaksa funkcija je vrlo slična kao i za `reshape` osim što nije potrebno specificirati vektor veličina dimenzija jer je rezultat uvijek jednodimenzionalno polje.

```
numpy.ravel(a, order='C')
```

Za pretvorbu matrice u vektor koristi se i funkcija `flatten`. Razlika je što je ovo funkcija `ndarray` objekta i uvijek kreira kopiju polja.

```
numpy.flatten(order='C')
```

Python kod 2.2 Mijenjanje dimenzija polja

```
import numpy as np

M1 = np.arange(0, 20)
print(M1)

M2 = np.reshape(M1, [4, 5])
print(M2)
M3 = np.reshape(M2, [2, -1])
print(M3)
```

```
print(np.ravel(M2))
print(M3.flatten())
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
[[ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
```

Brisanje jediničnih dimenzija matrice, koje vrlo često znaju biti suvišne, može se napraviti funkcijom `squeeze`.

Python kod 2.3 Mijenjanje dimenzija polja

```
import numpy as np

M1 = np.arange(0, 20)

M2 = np.reshape(M1, [4, 1, 1, 5])
print(M2.shape)

M3 = np.squeeze(M2)
print(M3.shape)
```

```
(4, 1, 1, 5)
(4, 5)
```

2.3.2 Transponiranje

Transponiranje matrica moguće je izvesti na dva načina:

- funkcijom `transpose`
- atributom `T`

Implementacija se vrši na način:

```
numpy.transpose(T)
```

gdje je argument `T` NumPy polje koje će se transponirati.

Python kod 2.4 Transponiranje matrica

```
import numpy as np

M1 = np.array([
    [1, 0, 3, 4],
    [2, 2, 7 ,8],
    [6, 8, 3, 0],
    [9, 8, 2, 1]])

M2 = M1.T
print(M2)
```

```
M3 = np.transpose(M1)
print(M3)
```

```
[[1 2 6 9]
 [0 2 8 8]
 [3 7 3 2]
 [4 8 0 1]]
[[1 2 6 9]
 [0 2 8 8]
 [3 7 3 2]
 [4 8 0 1]]
```

2.3.3 Dodavanje i brisanje elemenata polja

Dodavanje i brisanje elemenata polja je radnja koja se vrlo često koristi u programiranju. Postoji mnoštvo NumPy funkcija koje se koriste za dodavanje, brisanje ili modificiranje elemenata NumPy polja.

Funkcijom `numpy.append` se mogu vrijednosti pridodati kao zadnji elementi u postojeće NumPy polje. Koristi se na način:

```
numpy.append(arr, v)
```

gdje je argument `arr` originalno NumPy polje u koje će se dodati vrijednosti, a `v` vrijednost koja se dodaje kao zadnji element tog NumPy polja.

Python kod 2.5 Dodavanje elemenata na kraj NumPy polja

```
import numpy as np

A = np.array([1, 2, 3])
A = np.append(A, 10)

B = np.array([4, 5, 6])
B = np.append(B, [7, 8, 9])

print("A =", A)
print("B =", B)
```

```
A = [ 1  2  3 10]
B = [4 5 6 7 8 9]
```

Umjesto da se vrijednost dodaje na kraj NumPy polja funkcijom `append` može se dodati i na neko specifično mjesto u polju funkcijom `insert`. Ona se implementira na način:

```
numpy.insert(arr, x, v, axis)
```

gdje su argumenti `arr` i `v` jednaki kao kod funkcije `append`, a argument `x` se odnosi na indeks elementa u NumPy polju na koji se postavlja vrijednost `v`. Argument `axis` definira način na koji će se dodati vrijednost u NumPy polje. Zadana vrijednost argumenta `axis` je 0, što znači da je polje jednodimenzionalno te će se vrijednost `v` nadodati u nizu na poziciji `x`. U slučaju da je vrijednost argumenta 1, to znači da je NumPy polje u koje se nadodaje vrijednost matričnog oblika te se vrijednost `v` nadodaje kao stupac ili redak u NumPy polje.

Python kod 2.6 Dodavanje elemenata na specifični element NumPy polja

```
import numpy as np

C = np.array([1, 2, 3])
C = np.insert(C, 2, 10)

D = np.array([[4, 5, 6], [7, 8 ,9]])
D = np.insert(D, 1, 5, axis = 0)

E = np.array([[4, 5, 6], [7, 8 ,9]])
E = np.insert(E, 1, 5, axis = 1)

print("C =", C)
print("D =", D)
print("E =", E)
```

```
C = [ 1  2 10  3]
D = [[4 5 6]
     [5 5 5]
     [7 8 9]]
E = [[4 5 5 6]
     [7 5 8 9]]
```

U programskom kodu 2.6 može se vidjeti razlika između vrijednosti 0 i 1 za argument `axis`. Kada je vrijednost 0, `numpy.insert` je na zadanoj poziciji dodao cijeli redak vrijednosti 5, a kada je vrijednost 1, na zadanoj poziciji je dodan stupac vrijednosti 5.

Kada je potrebno izbrisati elemente iz NumPy polja koristi se funkcija `numpy.delete` te se implementira na način:

```
numpy.delete(arr, x, axis)
```

gdje su argumenti `arr`, `x` i `axis` već definirani kod `numpy.append` i `numpy.insert`, ali se u ovom slučaju ne dodaju nove vrijednosti nego brišu elementi na poziciji `x`.

Python kod 2.7 Brisanje elementa u NumPy polju

```
import numpy as np

F = np.array([1, 2, 3])
F = np.delete(F, 2)

G = np.array([[4, 5, 6],[8, 9, 10]])
G = np.delete(G, 2, axis = 1)

print("F =", F)
print("G =", G)
```

```
F = [1 2]
G = [[4 5]
     [8 9]]
```

U slučaju da se žele promijeniti dimenzije NumPy polja može se koristiti funkcija `numpy.resize` kojom se definira broj stupaca i/ili redaka koje može poprimiti novo NumPy polje. Definira se na način:

```
numpy.resize(arr,  ns)
```

gdje je argument `arr` NumPy polje koje se želi promijeniti, a `ns` su dimenzije redaka i/ili stupaca koje će promijeniti ulazni polje `arr`.

Python kod 2.8 Promjena dimenzija NumPy polja

```
import numpy as np

H = np.array([[0,1], [2,3]])
H = np.resize(H, (2, 3))

I = np.array([[4,5], [6,7]])
I = np.resize(I, (2,4))

print("H =", H)
print("I =", I)
```

```
H = [[0 1 2]
     [3 0 1]]
I = [[4 5 6 7]
     [4 5 6 7]]
```

U programskom kodu 2.8 definirana su dva NumPy polja dimenzija 2 retka i 2 stupca za koje je primijenjena funkcija `numpy.resize`. NumPy polje H je promijenilo oblik u 2 retka i 3 stupca te je funkcija `numpy.resize` automatski popunila elemente da bi se formirao takav oblik. Kada ponestane vrijednosti za formiranje oblika vrijednosti se ponovno ispočetka originalnog NumPy polja nadodaju.

S obzirom na to da se vrlo često u računarskom inženjerstvu formiraju matrice s mnogo nula, vrlo je korisna funkcija `numpy.trim_zeros` koja iz jednog polja briše vrijednosti nula s prvog i/ili zadnjeg elementa NumPy polja. Implementira se na način:

```
numpy.trim_zeros(arr,  trim)
```

gdje je `arr` NumPy polje iz kojeg se želi maknuti nula s početka i/ili kraja, a argument `trim` definira s koje strane se želi maknuti nula. Zadana vrijednost argumenta `trim` je string 'fb' koji predstavlja brisanje nula s obje strane NumPy polja. U slučaju da se želi maknuti nula samo s početka polja potrebno je staviti vrijednost 'f' (*front*), a ako se želi maknuti zadnja vrijednost stavlja se vrijednost 'b' (*back*).

Python kod 2.9 Brisanje nula s prvog ili zadnjeg elementa NumPy polja

```
import numpy as np

J = np.array([0,1,2,0,5,0,2,0])
J = np.trim_zeros(J)

K = np.array([0, 1, 0])
K = np.trim_zeros(K, trim = 'f')

print("J =", J)
print("K =", K)
```

```
J = [1 2 0 5 0 2]
K = [1 0]
```

Funkcijom `numpy.unique` može se iz NumPy polja u kojemu postoje više jednakih vrijednosti kreirati NumPy polje u kojemu će biti jedinstvene vrijednosti bez ponavljanja. Funkcija se definira na način:

```
numpy.unique(arr, return_index, return_inverse, return_counts, axis)
```

gdje je argument `arr` NumPy polje koje sadrži ponavljajuće vrijednosti, `axis` argument prethodno definiran kod funkcije `insert`. Argumenti `return_index`, `return_inverse` i `return_counts` poprimaju vrijednosti boolean tipa te vraćaju polje indeksa na kojemu se nalaze ponavljajuće vrijednosti, polje indeksa ponavljanja novog NumPy polja (u slučaju da se želi rekonstruirati originalnog polja) i polje koje predstavlja koliko puta se ponavljaju vrijednosti iz novog polja.

Python kod 2.10 Formiranje NumPy polja bez ponavljanja vrijednosti

```
import numpy as np

L = np.array([1, 1, 2, 2, 3, 3, 5])
L = np.unique(L, return_index=True, return_inverse=True, return_counts=True)

M = np.array([[1, 2, 2], [1, 2, 2], [2, 3, 3]])
M = np.unique(M, axis = 1)

N = np.array(['a', 'b', 'a', 'd', 'c', 'b'])
N = np.unique(N)

print ("L =", L)
print ("M =", M)
print ("N =", N)
```

```
L = (array([1, 2, 3, 5]),
     array([0, 2, 4, 6]),
     array([0, 0, 1, 1, 2, 2, 3]),
     array([2, 2, 2, 1]))
M = [[1 2]
     [1 2]
     [2 3]]
N = ['a' 'b' 'c' 'd']
```

U programskom kodu 2.10 i rezultatu tog koda vidi se primjena funkcije `numpy.unique`. Izlazna vrijednost polja L sadrži novo polje bez ponavljajućih vrijednosti, indekse vrijednosti koje su se maknuli iz originalnog polja L, indekse koji se ponavljaju da bi novo polje poprimilo izgled originalnog polja L i koliko puta se koja vrijednost ponavlja u novom polju da bi ono poprimilo izgled originalnog polja L. Također je moguće vidjeti po NumPy polju N da se funkcija `numpy.unique` može primijeniti i na vrijednosti tipa string.

2.3.4 Kopiranje polja

Ponekad je izrazito važno razumjeti da se NumPy polja prilikom pridruživanja postojećeg polja novoj varijabli **ne kopiraju**, već se samo kreira dodatno "ime" (*pointer*) za postojeće polje. U tom slučaju promjena izvornog polja automatski podrazumijeva i promjenu "novog", jer zapravo nikakvo kopiranje nije provedeno.

Ako želimo osigurati da zaista dobijemo kopiju polja sa kojom dalje želimo nešto raditi nezavisno od stanja izvornog polja, možemo koristiti funkciju `numpy.copy`. Funkcija se definira kao:

```
numpy.copy(arr, order)
```

gdje argument `arr` predstavlja NumPy polje koje želimo kopirati, dok je argument `order` opcionalan i služi za definiranje načina mapiranja polja u memoriji računala.

Programski kod 2.11 demonstrira kreiranje kopije polja.

Python kod 2.11 Kopiranje NumPy polja

```
import numpy as np

X = np.array([1, 2, 3])
Y = X
Xcopy = np.copy(X)

X[0] = 10

print('X = ', X)
print('Y = ', Y)
print('Xcopy = ', Xcopy)
```

```
X =  [10  2  3]
Y =  [10  2  3]
Xcopy =  [1 2 3]
```

2.4 Učitavanje i spremanje NumPy polja

Stefan Ivić

Vrlo se često događa da se rezultati raznih proračuna ili mjerenja spremaju u tekstualne datoteke. Takve datoteke, svaka zapisana u svom formatu često se učitavaju u Python za daljnju obradu, vizualizaciju ili provjeru. NumPy pruža gotove funkcije za automatizirano učitavanja tabličnih podataka u NumPy polja.

Jednako bitna je i mogućnost spremanja podataka iz NumPy polja u tekstualne datoteke.

Osim tekstualnih datoteka, učitavanje i spremanje NumPy polja moguće je i s binarnim datotekama.

2.4.1 Učitavanje iz tekstualnih datoteka

Učitavanje numeričkih vrijednosti iz tekstualnih datoteka u NumPy polja vrši se sa funkcijom `loadtxt`. Podrazumijeva se da su brojevi u tekstualnoj datoteci zapisani u retcima koji svi imaju jednak broj zapisanih vrijednosti. Arumenti funkcije `loadtxt` omogućavaju detaljnije određivanje učitavanja tekstualne datoteke pomoću razdvojnika, komentara, određivanja pasivnih početnih redaka te odabir stupaca za učitavanje i dr.

Sintaksa funkcije s najčešće korištenim argumentima je:

```
numpy.loadtxt(fname, comments, delimiter, skiprows, usecols)
```

gdje je `fname` ime datoteke iz koje se učitavaju podaci, `comments` je simbol kojim započinju pasivni retci (komentari), `delimiter` je simbol(i) kojim su razdvojene vrijednosti u retcima, `skiprows` je broj početnih redaka koji se preskaču prilikom učitavanja a `usecols` je lista indeksa stupaca koji se učitavaju.

> **Napomena**
> Zadana vrijednost argumenta `delimiter` je `None` što znači da se redak dijeli na temelju razmaka (podrazumijeva i tabulatore) i više uzastopnih razmaka smatra kao jedan razmak.

Sadržaj datoteke `data.txt`:

```
Tehnički fakultet
Zavod za mehaniku fluida i računarsko inženjerstvo


1   8.14    28.72   -0.12   126.04  5
2   8.37    26.79   -0.47   130.96  9
3   4.41    26.55   -0.44   125.82  2
4   6.74    22.48   -0.55   133.24  0
5   5.30    24.30   -0.70   137.91  9
* ovo je komentar
6   6.11    22.14   -0.87   102.43  7
7   2.10    25.96   -0.90   137.88  2
*8   7.40    24.40   -0.03   122.03  1
9   0.07    22.04   -0.73   143.81  2
10  8.45    23.07   -0.47   137.48  0
11  2.91    22.07   -0.07   116.26  6
12  9.14    20.70   -0.10   132.36  9
13  7.62    23.83   -0.51   111.39  1
```

Python kod 2.12 Učitavanje iz tekstualne datoteke

```
import numpy as np

M = np.loadtxt('data.txt', skiprows=4, comments='*', usecols=(1, 3, 5))
print(M)
```

```
[[ 8.14 -0.12  5.  ]
 [ 8.37 -0.47  9.  ]
 [ 4.41 -0.44  2.  ]
 [ 6.74 -0.55  0.  ]
 [ 5.3  -0.7   9.  ]
 [ 6.11 -0.87  7.  ]
 [ 2.1  -0.9   2.  ]
 [ 0.07 -0.73  2.  ]
 [ 8.45 -0.47  0.  ]
 [ 2.91 -0.07  6.  ]
 [ 9.14 -0.1   9.  ]
 [ 7.62 -0.51  1.  ]]
```

2.4.2 Naredba genfromtxt

Naprednije mogućnosti pri učitavanju NumPy polja iz tekstualnih datoteka pruža funkcija `genfromtxt`. Funkcija `genfromtxt` podatke tipa *string* automatski pretvara u decimalan broj vrijednosti *nan*.

> **Napomena**
> U datoteci se mogu pojaviti vrijednosti *none* ili *error* koji mogu za funkciju `loadtxt` uzrokovati probleme s učitavanjem datoteke zbog kombiniranja različitih tipova varijabi u jednoj datoteci (*int*/*float* i *string*). Funkcija `genfromtxt` nema problema s tim jer vrši konverziju takvih vrijednosti.

Nedostajuće podatke može se zamijeniti nekim drugim vrijednostima, primjerice:

```
numpy.genfromtxt('data.txt', filling_values=-999)
```

Gdje argument `filling_values` predstavlja vrijednost kojom će biti zamijenjene sve varijable tipa *str* koje se pojave u datoteci. Ukoliko se argumentu definira lista više vrijednosti, vrijednosti za zamjenu varijabli tipa "*string* će biti pridružene pripadnom stupcu. Sadržaj datoteke `data1.txt`:

```
8.14    26.72   -0.12
8.37    none    -0.47
N/A     26.55   -0.44
none    22.48   error
7.42    24.3    none
```

Python kod 2.13 Učitavanje iz tekstualne datoteke

```
import numpy as np

M = np.genfromtxt('data1.txt', filling_values=(-100,-200,-300))
print(M)
```

```
[[ 8.14 26.72  -0.12  ]
```

```
 [ 8.37 -200   -0.47  ]
 [ -100 26.55  -0.44  ]
 [ -100 22.48  -300   ]
 [ 7.42 24.3   -300   ]]
```

2.4.3 Spremanje u tekstualne datoteke

Funkcija `savetxt` omogućava jednostavno spremanje NumPy polja u tekstualne datoteke.

```
numpy.savetxt(fname, X, fmt, delimiter, newline, header, footer, comments)
```

gdje su argumenti `fname`, `comments` i `delimiter` isti kao i kod funkcije `loadtxt`, opisani u poglavlju 2.4.1. U argumentu `X` su stavljeni podaci u obliku polja koji će se spremiti u datoteku. Argument `fmt` definira način na koji će se zapisati podaci, moguće je skratiti decimalni zapis, koristiti znanstveni zapis brojeva itd. Argumentom `newline` se definira koja je oznaka kojom se dijele redovi podataka, a funkcije `header` i `footer` služe za zadavanje zapisa (string) na početku odnosno na kraj datoteke u koju se sprema.

Python kod 2.14 Pisanje u tekstualnu datoteku

```
import numpy as np

X = np.eye(4) * np.pi

naslov = "Naslov datoteke"
kraj = "Kraj datoteke"

M = np.savetxt("datoteka.txt", X, fmt='%.5e', newline='\n', header=naslov, footer=
    kraj)
```

Sadržaj datoteke `datoteka.txt` koja je generirana programskim kodom 2.14:

```
# Naslov datoteke
3.14159e+00 0.00000e+00 0.00000e+00 0.00000e+00
0.00000e+00 3.14159e+00 0.00000e+00 0.00000e+00
0.00000e+00 0.00000e+00 3.14159e+00 0.00000e+00
0.00000e+00 0.00000e+00 0.00000e+00 3.14159e+00
# Kraj datoteke
```

2.4.4 Naredba save

Funkcija `save` omogućava da se NumPy polje spremi kao NumPy binarna datoteka ekstenzije .npy. Implementira se na način:

```
numpy.save(file, arr)
```

gdje je argument `file` ime binarne datoteke u koju će se spremiti, a `arr` je NumPy polje koje se želi spremiti.

Python kod 2.15 Spremanje polja u NumPy datoteku

```
import numpy as np

X = np.array([1, 2, 3, 4])
```

```
M = np.save("numpy_polje", X)
```

2.4.5 Naredbe savez i savez_compressed

Funkcije `savez` i `savez_compressed` koriste se za spremanje nekoliko NumPy polja u jednu NumPy binarnu datoteku ekstenzije .npz. Razlika između dvije navedene funkcije je ta da `savez` polje ne koristi kompresiju podataka pri spremanju dok `savez_compressed` koristi. Isti argumenti se implementiraju u funkcije `savez` i `savez_compressed` te se vrši na način:

```
numpy.savez(fname, X, Y)
numpy.savez_compressed(fname, X, Y)
```

gdje je `fname` ime datoteke u koje će se spremiti polja (ekstenzija .npz), a `X` i `Y` su polja koja se žele spremiti u tu datoteku.

Python kod 2.16 Pisanje više NumPy polja u datoteku

```
import numpy as np

X = np.arange(1, 10, 1)
Y = X**2
Z = X*Y

M = np.savez("tri_polja", X, Y, Z)
M_komprimirano = np.savez_compressed("tri_polja_komprimirano", X, Y, Z)
```

2.4.6 Naredba load

Da bi se datoteke koje su kreirane funkcijama `save`, `savez` i `savez_compressed` učitale natrag u program potrebno je iskoristiti funkciju `load`. Funkcija `load` se definira na način:

```
numpy.load(file)
```

gdje je `file` argument naziva datoteke u kojoj je spremljeno binarno NumPy polje ili polja. Može učitati ekstenzije .npy i .npz te se one moraju dodati uz ime datoteke da bi funkcija radila.

Python kod 2.17 Čitanje NumPy datoteke u program

```
import numpy as np

A = np.load("numpy_polje.npy") #NumPy lista
B = np.load("tri_polja.npz") #NumPy dictionary
C = np.load("tri_polja_komprimirano.npz") #NumPy dictionary

print(B.files)
print(C.files)
```

U programskom kodu 2.17 može se vidjeti implementacija funkcije `load` te je uz svaku varijablu komentar. Kada se čita jedno NumPy polje ekstenzije .npy kreirano funkcijom `save`, onda `load` sprema podatke iz datoteke u listu, dok se za `savez` i `savez_compressed` podaci spremaju u dictionary tip podatka (opisan u poglavlju

1.3.7). Da bi se pristupilo spremljenim poljima u tom tipu podatka potrebno je znati vrijednost key elementa, a te vrijednosti se mogu dobiti atributom `files` kao što je napravljeno u programskom kodu.

2.5 Matrični račun

Stefan Ivić

NumPy modul `numpy.linalg` sadrži funkciju kojom se može rješavati sustav linearnih jednadžbi. Za rješavanje sustava linearnih jednadžbi potrebno je kreirati NumPy polje matričnog te vektorskog oblika.

2.5.1 Rješavanje sustava linearnih jednadžbi

Za rješavanje sustava linearnih jednadžbi

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{2.1}$$

koristi se NumPy funkcija `solve` (koja se nalazi u modulu `linalg`):

```
numpy.linalg.solve(a, b)
```

gdje je `a` matrica sustava a `b` vektor slobodnih članova.

Python kod 2.18 Rješavanje sustava linearnih jednadžbi

```
import numpy as np

A = np.array([
    [ 1.0,  3.0,  6.0],
    [-2.0,  0.0,  3.0],
    [ 8.0, -2.0,  0.0]
    ])
b = np.array([2.0, 6.0, 0.0])

x = np.linalg.solve(A, b)

print(x)
```

```
[-0.58823529 -2.35294118  1.60784314]
```

Detaljni postupak rješavanja sustava linearnih jednadžbi i korištene metode opisane su u poglavlju 4.5.

2.6 Rad s polinomima

Siniša Družeta

NumPy sadrži mnogo prikladnih funkcija za rad sa polinomima, što uključuje kreiranje polinoma, manipulacije s polinomima te prilagodbu na podatke (eng. *fitting*). Sve te funkcionalnosti sadržane su u podmodulu `numpy.polynomial`.

Za rad s polinomima potrebno je importirati `Polynomial` klasu:

```
from numpy.polynomial import Polynomial
```

Sada možemo definirati polinom pomoću koeficijenata:

```
p = Polynomial(koeficijenti)
```

ili pomoću korijena (nul-točaka):

```
p = Polynomial.fromroots(korijeni)
```

a možemo i kreirati polinom m-tog stupnja regresijom na skupu točaka (X, Y):

```
p = Polynomial.fit(X, Y, m)
```

Na definiranom polinomu možemo očitati koeficijente

```
p.convert().coef
```

dok se lista njegovih korijena može dobiti s

```
p.roots()
```

Evaluacija polinoma `p` u danoj točki x može se vršiti pozivom `Polynomial` objekta kao funkcije:

```
y = p(x)
```

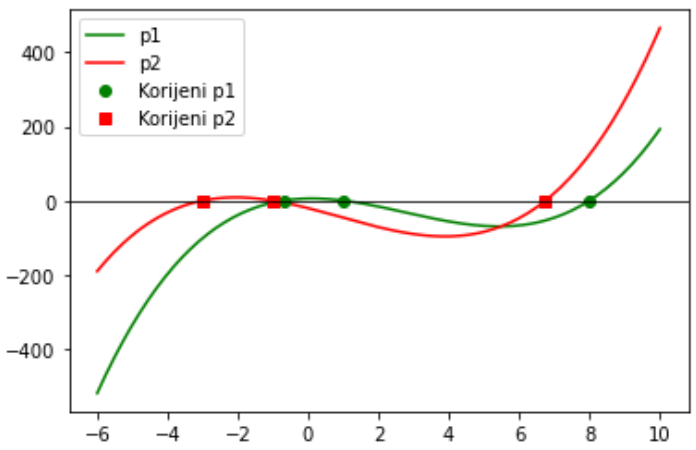
gdje `x` može biti broj ili `ndarray`.

Python kod 2.19 Zadavanje polinoma kao `Polynomial` objekta

```
import numpy as np
from numpy.polynomial import Polynomial
import matplotlib.pyplot as plt

# Zadavanje polinoma preko koeficijenata
p1 = Polynomial([ 5.6, 1.7, -8.3, 1.])
print('p1:')
print(p1)

# Zadavanje polinoma preko korijena
p2 = Polynomial.fromroots([-3, -1.02, 6.75])
print('\np2:')
print(p2)

# Koeficijenti polinoma
print('\nKoeficijenti:')
print(p1.convert().coef)
```

```
print(p2.convert().coef)

# Korijeni polinoma
x01 = p1.roots()
x02 = p2.roots()
print('\nKorijeni:')
print(x01)
print(x02)

x = np.linspace(-6, 10, 200)
y1 = p1(x)
y2 = p2(x)

plt.plot(x, y1, 'g', label='p1')
plt.plot(x, y2, 'r', label='p2')
plt.plot(x01, np.zeros_like(x01), 'go', label='Korijeni p1')
plt.plot(x02, np.zeros_like(x02), 'rs', label='Korijeni p2')
plt.axhline(c='k', lw=0.8)
plt.legend(loc='best')
plt.show()
```

```
p1:
5.6 + 1.7 x - 8.3 x**2 + 1.0 x**3

p2:
-20.655 - 24.075 x - 2.73 x**2 + 1.0 x**3

Koeficijenti:
[ 5.6   1.7 -8.3   1. ]
[-20.655 -24.075  -2.73    1.   ]

Korijeni:
[-0.7  1.   8. ]
[-3.   -1.02  6.75]
```



Slika 2.1
Graf polinoma iz primjera 2.19

3. Vizualizacija podataka

Matplotlib
Linijski grafovi
Dodatne postavke linijskih grafova
Svojstva prikaza grafa
Spremanje grafova
Podgrafovi
Grafovi funkcija dvije varijable
Animacije

3.1 Matplotlib

Luka Grbčić
Ivana Lučin

Matplotlib je knjižnica za vizualizaciju podataka u programskom jeziku Python. Sadrži mnoge funkcionalnosti koje olakšavaju korisniku izradu različitih vrsta grafova s mnogim mogućnostima prilagodbe njihova izgleda. Omogućava jednostavnu izradu linijskih 2D grafova i animacija kao i njihovo pohranjivanje.

3.1.1 Podmodul PyPlot

Modul za izradu grafova unutar knjižnice matplotlib je PyPlot. Funkcije koje su dio modula omogućavaju brzo i jednostavno kreiranje grafova, pri čemu korisnik u potpunosti može regulirati svojstva linija, teksta, koordinatnih osi, dimenzija grafa itd. PyPlot podržava izradu grafova i korištenjem NumPy polja. Dodavanje modula u programski kod se izvršava pomoću naredbe:

```
import matplotlib.pyplot
```

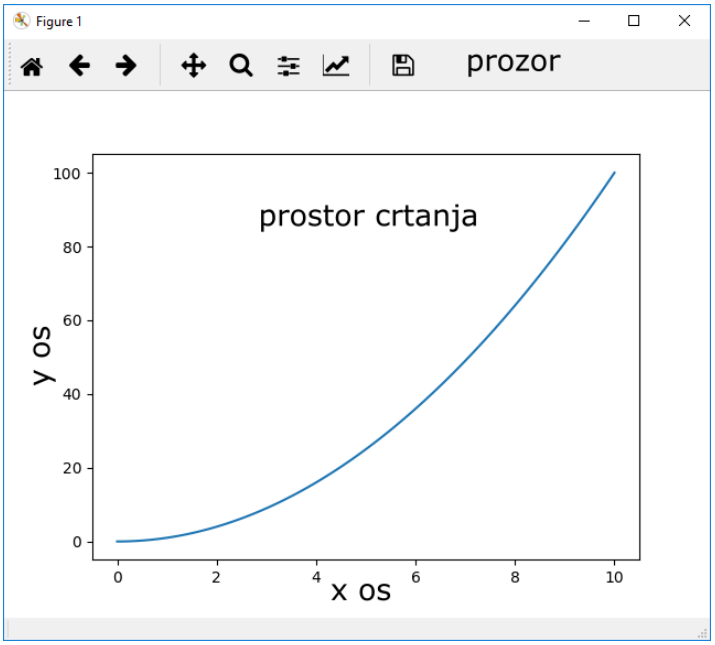
Terminologija koja se koristi pri izradi grafova u matplotlib knjižnici i PyPlot modulu je prozor (eng. *figure*), prostor crtanja (eng. *axes*) te osi (eng. *axis*) prikazano na slici 3.1.

Osim prikaza grafova u posebnom prozoru u Spyder 3 okruženju moguće je prikazati grafove i direktno u IPython konzoli, slika 3.2.

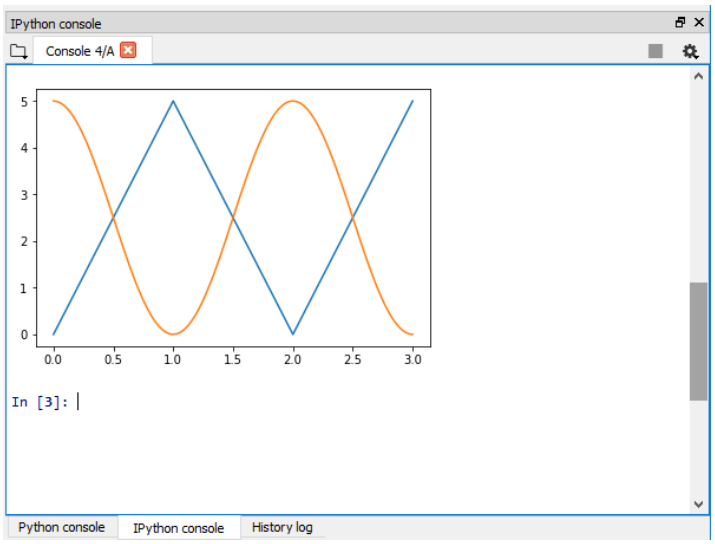
Ako se u kod postavi više uzastopnih funkcija za prikaz grafa, tada će se grafovi iscrtati na jednom prozoru, odnosno na istom prostoru crtanja. Da bi se grafovi prikazali u posebnom prozoru, potrebno je nakon naredbe za prikaz grafa napisati funkciju `show`:

```
matplotlib.pyplot.plot(args)
matplotlib.pyplot.show()
```

U slučaju prikaza grafa u IPython konzoli nije potrebno napisati funkciju `show` nakon pozivanja funkcije `plot`.



Slika 3.1
Terminologija matplotliba



Slika 3.2
Prikaz grafa u konzoli

3.2 Linijski grafovi

Luka Grbčić
Ivana Lučin

Za crtanje linijskih grafova kao argument u funkciji plot potrebno je staviti vektor koordinata točaka za svaku os.

```
matplotlib.pyplot.plot(x, y)
```

Za generiranje grafa funkcija `plot` pristupa pojedinom elementu vektora x te ga pridružuje odgovarajućem elementu vektora y čime oni formiraju x i y koordinatu za jednu točku grafa. Elementi vektora x i y moraju imati jednaki broj elemenata da bi funkcija plot radila.

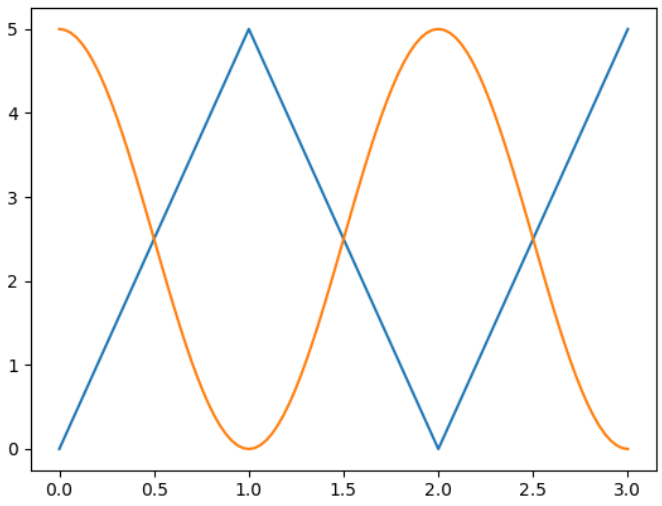
```
matplotlib.pyplot.plot(y)
```

U slučaju da se postavi samo vektor y osi, funkcija plot će sama kreirati uniformni vektor jednake duljine za os x [0, 1, 2, 3, ..., n].

Python kod 3.1 Dva jednostavna grafa

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

A = [0, 5, 0, 5]
plt.plot(A)

x = np.linspace(0, 3, 100)
y = 2.5 + 2.5*np.cos(np.pi * x)

plt.plot(x,y)
plt.show()
```

Programski kod 3.1 kreira dva grafa na jednom prostoru crtanja funkcijom `plot` te ih prikazuje funkcijom `show`.



Slika 3.3
Grafovi generirani iz primjera 3.1

Promatrajući sliku 3.3 i odgovarajući programski kod može se vidjeti da je graf plave boje definiran samo jednim argumentom, vektorom od četiri elemenata. Funkcija

plot je za taj vektor sama kreirala još jedan vektor koji odgovara x osi te ima jednak broj elemenata i definiran je kao [0, 1, 2, 3]. Graf zelene boje poprima dva vektora x i y kao argument. Vektor x je definiran sa 100 elemenata, a vektor y zbog svoje ovisnosti o vektoru x poprima jednak broj elemenata. Funkcija plot točke u vektorima spaja linijama te se tako dobije izgled krivulje. Manjim brojem točaka u vektoru dobije se izgled manje zaglađene krivulje.

3.3 Dodatne postavke linijskih grafova

Luka Grbčić
Ivana Lučin

U funkciji `plot` moguće je definirati boju grafa, tip oznake (eng. *marker*) točke koordinata i tip linije kojom će funkcija `plot` spajati koordinate. Da bi se postavka primijenila, potrebno je boju grafa i tip oznake točke napisati kao argument u funkciji `plot`. Argument je string i definira se nakon vektora grafa.

Jednostavni primjer funkcije s argumentom za crtanje grafa crvene boje s punom linijom:

```
matplotlib.pyplot.plot(x,y,'r-')
```

U slučaju da argument za postavku boje nije definiran u funkciji `plot`, postavlja se zadana boja, a ako oznaka točke nije definirana, funkcija `plot` spaja točke punom linijom.

Tablica 3.1
Tipovi linija

Simbol	Tip linije
`'-'`	Puna linija
`'--'`	Iscrtkana linija
`'-.'`	Crta-točka
`':'`	Točkasta linija

Tablica 3.2
Boje

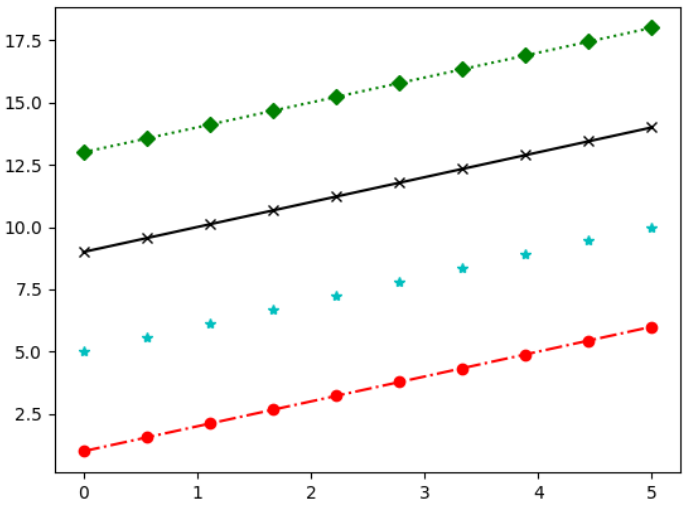
Simbol	Boja
`'b'`	Plava
`'r'`	Crvena
`'g'`	Zelena
`'c'`	Cijan
`'m'`	Magenta
`'y'`	Žuta
`'k'`	Crna
`'w'`	Bijela

Tablica 3.3 Oznake

Simbol	Oznaka
'.'	Točka
','	Piksel
'o'	Krug
'v'	Trokut prema dolje
'^'	Trokut prema gore
'>'	Trokut prema desno
'<'	Trokut prema lijevo
's'	Kvadrat
'p'	Peterokut
'*'	Zvijezda
'h'	Šesterokut 1
'H'	Šesterokut 2
'+'	Plus
'x'	Križ
'D'	Romb
'd'	Tanki romb
'|'	Vertikalna linija
'_'	Horizontalna linija

Python kod 3.2 Primjena različitih oznaka i boja

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

x = np.linspace(0, 5, 10)
f1 = x + 1
f2 = x + 5
f3 = x + 9
f4 = x + 13

plt.plot(x, f1, 'r-.o')
plt.plot(x, f2, 'c*')
plt.plot(x, f3, 'k-x')
plt.plot(x, f4, 'g:D')
plt.show()
```



Slika 3.4
Grafovi različitih boja i oznaka iz programskog koda 3.2

Na slici 3.4 vidi se prikaz grafova koji su definirani u programskom kodu 3.2. Zadana su četiri različita grafa s istim vektorom osi x, a različitim vektorima osi y. Svaki graf je izgledom drugačiji jer su argumenti za boje, oznake i linije drugačije definirani u funkciji `plot` nakon x i y vektora.

3.3.1 Napredno definiranje boja

U funkciji `plot` moguće je naprednije definirati boju grafa koja će se koristiti, a to se radi dodavanjem argumenta `color` (ili samo `c`) nakon x i y vektora:

```
matplotlib.pyplot.plot(x, y, color=0.5)
matplotlib.pyplot.plot(x, y, color='#0000FF')
matplotlib.pyplot.plot(x, y, c=(0.2, 0.7, 0.1))
matplotlib.pyplot.plot(x, y, c ='Blue')
matplotlib.pyplot.plot(x, y, c ='C1')
```

Općenito, vrijednosti argumenta color su definirane kao:

- `gray_string` - string koji definira nijansu sive boje, gdje je vrijednost 0 crna boja, a 1 bijela boja
- `hex_string` - string heksadecimalne vrijednosti boje (npr. '#0000FF')
- `rgb_tuple` - tuple u kojem su definirana tri broja koja odgovaraju udjelima crvene, zelene i plave boje
- `html_string` - string koji sadrži HTML naziv boje
- `C_string` - string koji sadrži oznaku standardnih matplotlib boja ('C0', 'C1', 'C2', ..., 'C9')

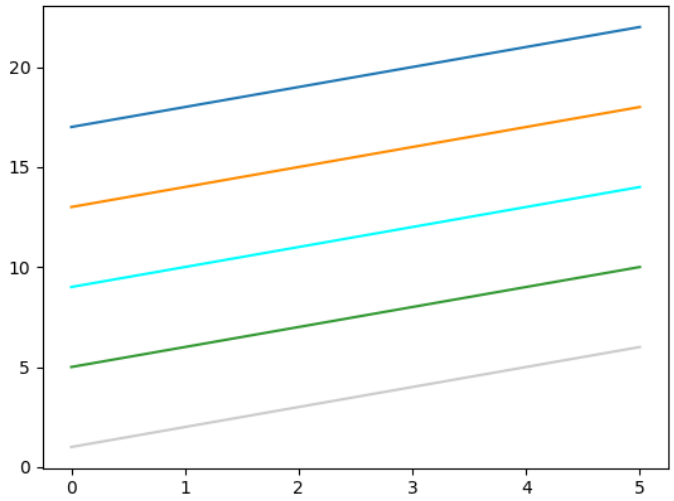
Boje je moguće i definirati punim nazivom:

```
matplotlib.pyplot.plot(x, y, color='grey')
matplotlib.pyplot.plot(x, y, c='darkorange')
matplotlib.pyplot.plot(x, y, color='black')
```

U dokumentaciji knjižnice matplotliba (`http://matplotlib.org/examples/color/named_colors.html`) moguće je vidjeti punu listu svih boja koje se mogu koristiti (s i bez argumenta `color`).

Python kod 3.3 Napredne postavke boja

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

x = np.linspace(0, 5, 10)
g1 = x + 1
g2 = x + 5
g3 = x + 9
g4 = x + 13
g5 = x + 17

plt.plot(x, g1, color='0.8')        # gray_string
plt.plot(x, g2, c=(0.2, 0.6, 0.2))  # rgb_tuple
plt.plot(x, g3, color='Cyan')       # html_string
plt.plot(x, g4, c='#FF8C00')        # hex_string
plt.plot(x, g5, c='C0')             # C[0-9]
plt.show()
```



Slika 3.5
Grafovi različitih boja iz programskog koda 3.3

3.3.2 Napredno definiranje oznaka i linija

Kao što je moguće naprednije definirati karakteristiku boje grafa, tako je moguće oznaku i linije grafa detaljnije definirati. Oznake se kao argument u funkciji `plot` mogu definirati koristeći sljedeće naredbe:

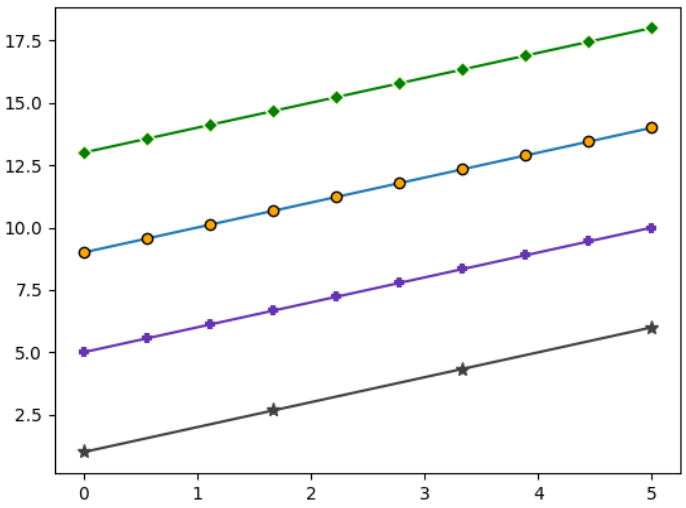
- `marker` - naredba kojom se definira tip oznake, simboli definirani u tablici 3.3
- `markersize` (`ms`) - broj kojim se definira veličine oznake
- `markeredgecolor` (`mec`) - string kojim se definira boja ruba oznake
- `markerfacecolor` (`mfc`) - string kojim se definira boja lica oznake
- `markeredgewidth` (`mew`) - broj kojim se definira debljina ruba oznake
- `markevery` - broj kojim se definira koliko točaka će biti prikazano (npr. svaka druga)

Navedeni argumenti postavljaju se na način:

```
matplotlib.pyplot.plot(x, y, markersize=5)
matplotlib.pyplot.plot(x, y, markeredgecolor='r', markerfacecolor='b')
matplotlib.pyplot.plot(x, y, markevery=2, markeredgewidth=1)
```

Python kod 3.4 Napredne postavke oznaka

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

x = np.linspace(0, 5, 10)
h1 = x + 1
h2 = x + 5
h3 = x + 9
h4 = x + 13

plt.plot(x, h1, color='0.25', marker='*', markersize=8, markevery=3)
plt.plot(x, h2, c=(0.4, 0.2, 0.7), marker='+', markeredgewidth=3)
plt.plot(x, h3, marker='o', markerfacecolor='orange', markeredgecolor='k')
plt.plot(x, h4, c='g', marker='D', markeredgecolor='lightyellow')
plt.show()
```



Slika 3.6
Grafovi različitih oznaka i boja iz programskog koda 3.4

Linije se podešavaju sljedećim argumentima u funkciji `plot`:

- `linestyle` - string kojim se definira tip linije, stringovi definirani u tablici 3.1
- `linewidth` - broj kojim se definira debljina linije

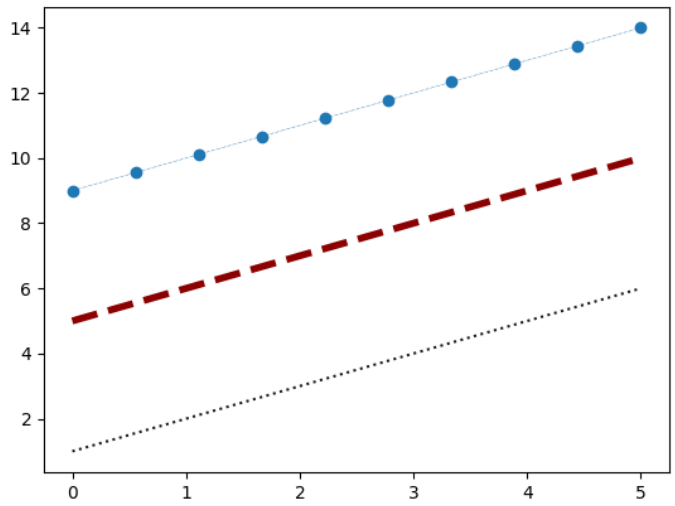
Argumenti se implementiraju na način:

```
matplotlib.pyplot.plot(x, y, linestyle='--')
matplotlib.pyplot.plot(x, y, linewidth=4)
matplotlib.pyplot.plot(x, y, linestyle=':', linewidth=1.8)
```

Python kod 3.5 Napredne postavke linija

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

x = np.linspace(0, 5, 10)
h1 = x + 1
h2 = x + 5
h3 = x + 9

plt.plot(x, h1, color='0.1', linestyle=':')
plt.plot(x, h2, c='darkred', linestyle='--', linewidth=4)
plt.plot(x, h3, marker='o', linestyle='-.', linewidth=0.3)
```

```
plt.show()
```



Slika 3.7
Grafovi različitih tipova linija definiranih u programskom kodu 3.5

3.3.3 Transparentnost grafova

Uz sve navedene mogućnosti naprednijeg podešavanja izgleda grafova, moguće je i podesiti transparentnost samog grafa na prostoru crtanja. Transparentnost grafa se u funkciji `plot` definira argumentom `alpha` na način:

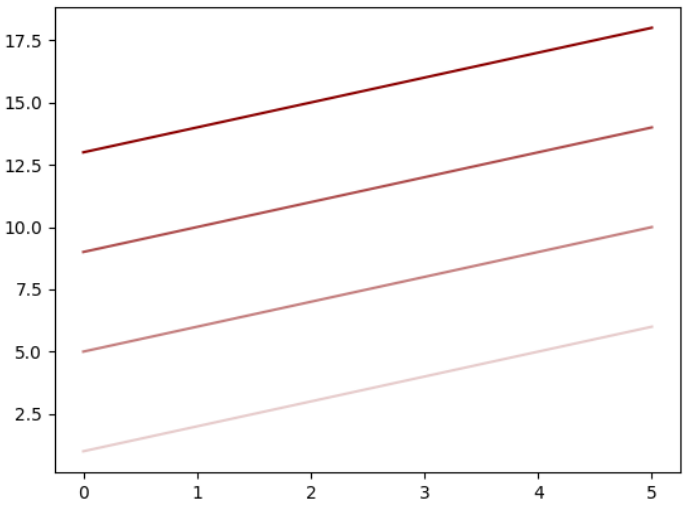
```
matplotlib.pyplot.plot(x, y, alpha=0.5)
```

gdje `alpha` mora biti broj od 0 do 1. Niže vrijednosti čine graf više transparentnim dok vrijednost 1 formira potpuno netransparentan graf.

Python kod 3.6 Transparentnost linija

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

x = np.linspace(0, 5, 10)
h1 = x + 1
h2 = x + 5
h3 = x + 9
h4 = x + 13

plt.plot(x, h1, color='darkred', alpha=0.2)
plt.plot(x, h2, color='darkred', alpha=0.5)
plt.plot(x, h3, color='darkred', alpha=0.7)
plt.plot(x, h4, color='darkred', alpha=1)
plt.show()
```

U programskom kodu 3.6 je definirana samo jedna boju za svaku krivulju, ali je uz argument boje dodan i argument transparentnosti `alpha` te zato na slici 3.8 svaka boja poprima drugu nijansu.



Slika 3.8
Grafovi različitih transparentnosti, definirano u programskom kodu 3.6

3.3.4 Bojanje područja između dvije krivulje

Naredba `fill_between` omogućava popunjavanje bojom prostora ispod krivulje. Argumenti koje je potrebno zadati naredbi su x i y koordinate točaka od kojih se popunjava površina prema x-osi.

```
matplotlib.pyplot.fill_between(x, y)
```

Dodatni opcionalni argument funkcije je `y2`, koji određuje y-koordinate točaka dodatne krivulje koja predstavlja drugu granicu područja koje se popunjava bojom. Ukoliko je `y2` broj onda je druga granica horizontalni pravac definiran konstantom.

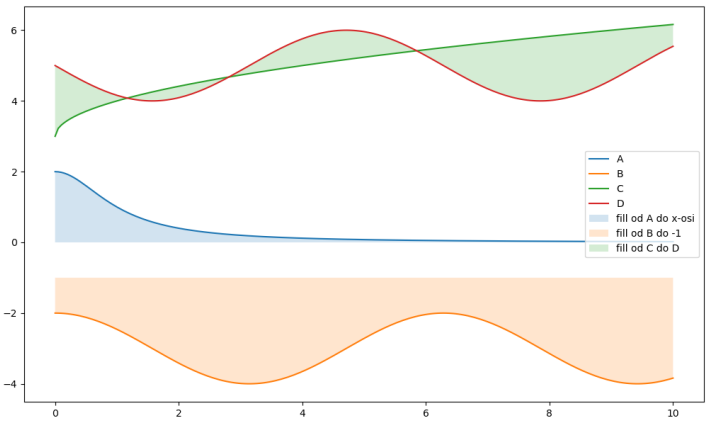
```
matplotlib.pyplot.fill_between(x, y, y2)
```

Dodatno, moguća je promjena boje (`color`), prozirnosti (`alpha`) i sl.

Python kod 3.7 Primjer upotrebe naredbe `fill_between`

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

x = np.linspace (0, 10, 200)
ya = 2 / (1 + x**2)
yb = np.cos (x) - 3
yc = 2 + np.sqrt(x) + 1
yd = 5 - np.sin(x)

plt.plot(x, ya, label='A')
plt.plot(x, yb, label='B')
plt.plot(x, yc, label='C')
plt.plot(x, yd, label='D')

# Popunjavanje do x osi
plt.fill_between(x, ya, alpha=0.2, label='fill od A do x-osi')

# Popunjavanje do zadanog pravca za y=-1
plt.fill_between(x, yb, y2=-1, alpha=0.2, label='fill od B do -1')

# Popunjavanje između 2 krivulje (C i D)
plt.fill_between(x, yc, yd, alpha=0.2, label='fill od C do D')

plt.legend(loc='right')
```

```
plt.subplots_adjust(left=0.03, right=0.99, bottom=0.05, top=0.98)
```



Slika 3.9
Primjeri za popunjavanje područja grafa bojom korištenjem naredbe `fill_between`

3.4 Svojstva prikaza grafa

Ivana Lučin
Luka Grbčić

Novi prozor stvara se pomoću funkcije `figure`. Pozivanjem funkcije na odgovarajućim mjestima u kodu omogućava se prikaz grafova u odvojenim prozorima.

```
matplotlib.pyplot.figure()
```

Funkcija `figure` po potrebi može primati dodatne argumente kojima se može regulirati izgled grafa:

```
matplotlib.pyplot.figure(figsize = (4, 3), dpi = 100)
```

Argument `figsize` prima tuple koji specificira širinu i visinu prozora u inčima. Argument `dpi` (*dots per inch*) regulira rezoluciju koja je bitna kod spremanja grafova u raster format ili pri spremanju animacija u video format.

```
matplotlib.pyplot.figure(edgecolor ='blue', linewidth=1, facecolor='lightblue')
```

Argument `facecolor` definira boju oko prostora crtanja grafa. Argumenti `edgecolor` i `linewidth` reguliraju boju i debljinu okvira grafa.

Na primjeru programskog koda 3.1 napravljen je programski kod 3.8 gdje je na odgovarajuća mjesta pozvana funkcija `figure` pri čemu se postiže prikaz grafova u odvojenim prozorima s različitim postavkama prikaza grafa.

Python kod 3.8 Odvojeni prikaz grafova

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

A = [0, 5, 0, 5]
plt.figure(figsize=(4,3), dpi=100)
plt.plot(A)

x = np.linspace(0, 3, 100)
y = 2.5 + 2.5*np.cos(np.pi * x)
plt.figure(edgecolor='blue', linewidth=1, facecolor='lightblue')
plt.plot(x, y)

plt.show()
```

3.4.1 Prostor crtanja

Dodavanje naslova na prostor crtanja se vrši pomoću funkcije `title`:

```
matplotlib.pyplot.title('Tekst naslova')
```

Uključivanje i isključivanje mreže pomoćnih linija na prostoru crtanja vrši se pozivom funkcije `grid` koja se može pozvati i bez dodatnih argumenata koji služe za preciznije definiranje postavki.

```
matplotlib.pyplot.grid(b='on', axis='x', which='major')
```

Argumentom `b` kontrolira se isključivanje ili uključivanje mreže pomoćnih linija. Može poprimiti vrijednosti *true* ili *false* kao i *on* ili *off*. Argumentom `axis` kontrolira se za koje će osi biti postavljena mreža pomoćnih linija. Vrijednosti za `axis` mogu biti *both*, *x* i *y*. Argument `which` može biti *major*, *minor* ili *both* i služi za kontrolu prikaza glavne ili sporedne mreže pomoćnih linija. Ukoliko se želi prikazati samo sporedna mreža pomoćnih linija potrebno je pozvati dodatnu funkciju za njen prikaz.

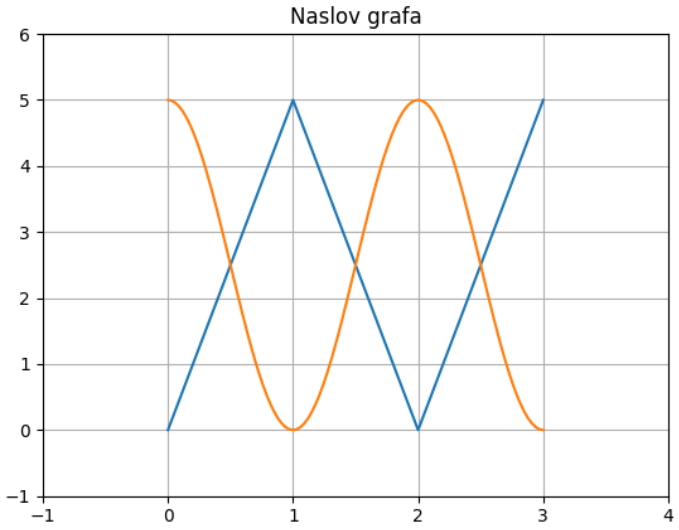
```
matplotlib.pyplot.grid(which='minor')
minorticks_on()
```

Funkcija `grid` može primati dodatne argumente za detaljniju kontrolu izgleda linija mreže pomoćnih linija. Primjerice:

```
matplotlib.pyplot.grid(color='k', linestyle=':', linewidth='0.2')
```

Funkcijama `xlim` i `ylim` određuju se granice prostora crtanja. Sintaksa funkcije je:

```
matplotlib.pyplot.xlim((xmin, xmax))
matplotlib.pyplot.ylim((ymin, ymax))
```



Slika 3.10
Izmijenjenje granice grafa

Dodaju li se izvornom programskom kodu 3.1 sljedeće linije prije poziva funkcije `show`:

```
matplotlib.pyplot.title('Naslov grafa')
matplotlib.pyplot.grid()
matplotlib.pyplot.xlim((-1, 4))
matplotlib.pyplot.ylim((-1, 6))
```

moguće je dobiti graf na slici 3.10.

3.4.2 Koordinatne osi

Funkcijama `xlabel` i `ylabel` moguće je dodati nazive koordinatnim osima:

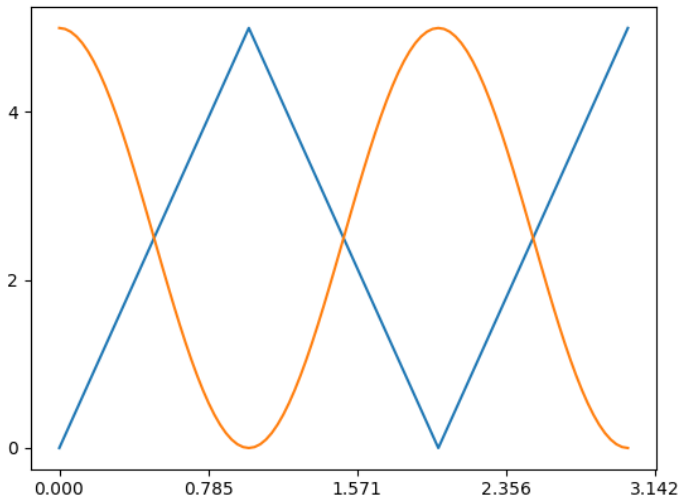
```
matplotlib.pyplot.xlabel('Opis x osi')
matplotlib.pyplot.ylabel('Opis y osi')
```

Ukoliko se žele promijeniti oznake vrijednosti na koordinatnim osima to je moguće učiniti pozivom funkcije `ticks`. Funkcija kao argumente može primiti vektor lokacija na kojima se žele postaviti oznake te vektor koji sadrži njihove nazive.

```
matplotlib.pyplot.xticks(location, labels)
```

Na taj način može se regulirati broj podjela na koordinatnim osima te se također mogu definirati nove tekstualne oznake na koordinatnim osima. Primjerice ukoliko se graf želi prikazati na osi x na području od 0 do π s korakom $\pi/4$ i na osi y na području od 0 do 6 s korakom 2 potrebno je dodati sljedeće linije:

```
matplotlib.pyplot.xticks([0, np.pi/4, np.pi/2, np.pi*3/4, np.pi])
matplotlib.pyplot.yticks([0, 2, 4, 6])
```



Slika 3.11
Prilagođene vrijednosti na koordinatnim osima

Navedene linije produciraju graf na slici 3.11. Kako bi se promijenile i oznake na osima potrebno je funkciji `xticks` dati vektor s pripadnim tekstualnim vrijednostima koje će na grafu zamijeniti postojeće:

```
matplotlib.pyplot.xticks([0, np.pi/4, np.pi/2, np.pi*3/4, np.pi],
       ['0', '$\pi$/4', '$\pi$/2', '$\pi$*3/4', '$\pi$'])
```

Na taj se način producira konačan izgled grafa prikazan na slici 3.12.

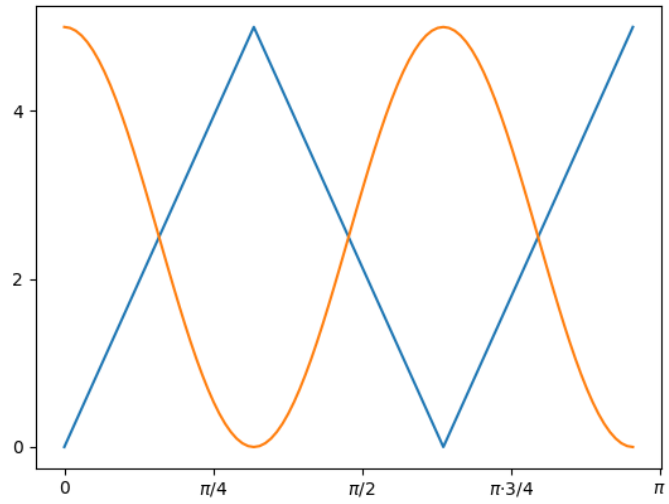
Kako bi se prikazale koordinatne osi mogu se koristiti funkcije `axhline` i `axvline`. Pozivanje navedenih funkcija bez argumenata kreiraju se osi x i y na grafu:

```
matplotlib.pyplot.axhline()
matplotlib.pyplot.axvline()
```

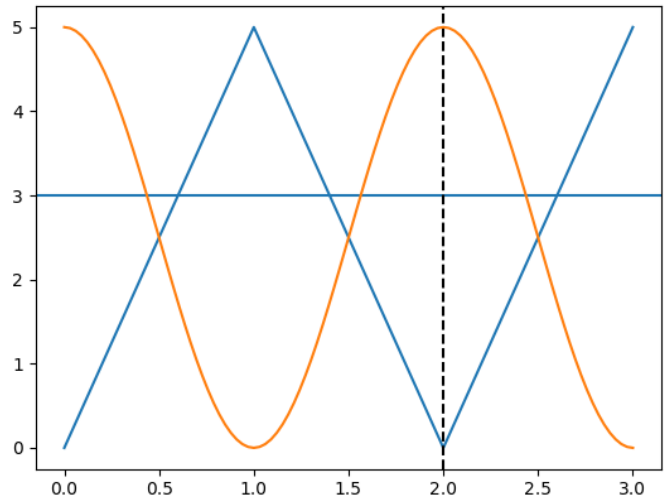
Dodavanjem sljedećih linija izvornom kodu 3.1:

```
matplotlib.pyplot.axhline(3)
matplotlib.pyplot.axvline(2, color='k', linestyle='--')
```

dobiva se graf (slika 3.13) na kojem je kreirana horizontalna linija koja prolazi kroz vrijednost 3 i vertikalna linija koja prolazi kroz vrijednost 2. Vertikalnoj liniji je dodatnim argumentima definiran prikaz crnom bojom i isprekidanom linijom. Korištenjem ostalih argumenata za manipulaciju linijama može se dodatno prilagoditi izgled grafa.



Slika 3.12
Prilagođene oznake vrijednosti na koordinatnim osima



Slika 3.13
Dodana vertikalna i horizontalna linija

Ukoliko se graf želi prikazati s jednakim granicama za obje osi potrebno je pozvati funkciju:

```
matplotlib.pyplot.axis('equal')
```

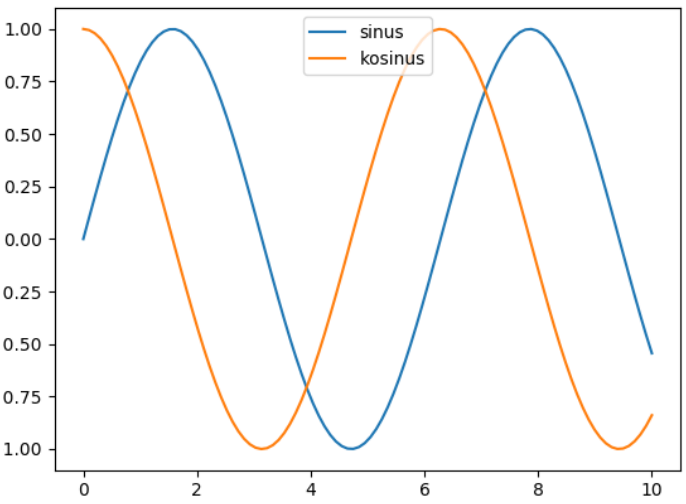
3.4.3 Legenda

Za svaku krivulju prikazanu na grafu može se pozivanjem argumenta `label` unutar funkcije `plot` definirati njen opis. Opisi se mogu prikazati u legendi na prostoru crtanja pomoću funkcije `legend`. Argumentom `loc` određuje se pozicija legende na prozoru. Vrijednosti koje se daju argumentu mogu biti cijeli brojevi ili stringovi koji specificiraju poziciju legende na prostoru crtanja.

Python kod 3.9 Prikaz legende na grafu

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

x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)
```

```
plt.plot(x, y1, label='sinus')
plt.plot(x, y2, label='kosinus')
plt.legend(loc='upper center')
plt.show()
```



Slika 3.14
Prikaz legende na grafu

Pozicija	Broj	String
Gore desno	1	`'upper right'`
Gore lijevo	2	`'upper left'`
Dolje lijevo	3	`'lower left'`
Dolje desno	4	`'lower right'`
Desno	5	`'right'`
Na sredini lijevo	6	`'center left'`
Na sredini desno	7	`'center right'`
Dolje u sredini	8	`'lower center'`
Gore u sredini	9	`'upper center'`
U sredini	10	`'center'`

Tablica 3.4
Pozicija legende

3.4.4 Anotacije

Na grafu se mogu dodati tekstualne oznake pomoću funkcije `annotate`. Funkciji je potrebno definirati tekst koji će se prikazati na grafu i koordinate koje definiraju gdje želimo da se tekst prikaže. Lokaciju je potrebno definirati u argumentu `xy`:

```
matplotlib.pyplot.annotate('tekst oznake', xy=(1, 1))
```

Tekst je također moguće prikazati s pokaznim strelicama navođenjem dodatnih argumenata u funkciji:

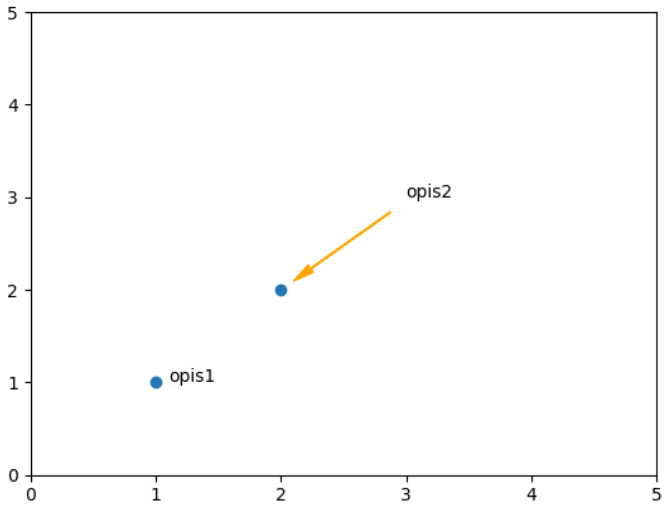
```
matplotlib.pyplot.annotate('tekst oznake', xy=(1,1), xytext=(3,3),
          arrowprops=dict(facecolor='black', width=0.5, shrink=0.1)))
```

Argument `xytext` definira završnu točku pokazne strelice, argument `arrowprops` omogućava detaljnije definiranje izgleda strelice gdje se može regulirati boja, veličina

linije i vrha strelice kao i udaljenost vrha strelice od točke na koju pokazuje (argument `shrink`).

Python kod 3.10 Prikaz anotacija na grafu

```
import matplotlib.pyplot as plt

plt.plot([1, 2], [1, 2], 'o')
plt.annotate('opis1', xy=(1.1, 1))
plt.annotate('opis2', xy=(2, 2), xytext=(3, 3),
             arrowprops=dict(color='orange', width=0.5,
                             headwidth=5, shrink=0.1))
plt.xlim(0, 5)
plt.ylim(0, 5)
plt.show()
```



Slika 3.15
Prikaz anotacija na grafu

Izvršavanjem programskog koda 3.10 dobiva se izgled na Slici 3.15, gdje su prikazana oba načina kreiranja anotacija na grafu.

3.5 Spremanje grafova

Luka Grbčić
Ivana Lučin

Spremanje grafova radi se funkcijom `savefig` koju sadrži PyPlot modul. Funkcija `savefig` postavlja se nakon što je napisana i definirana funkcija `plot`:

```
matplotlib.pyplot.plot(x, y)
matplotlib.pyplot.savefig('imedatoteke')
```

gdje `savefig` kao argument poprima ime datoteke u obliku stringa. Ako format slike nije definiran kao ekstenzija na ime datoteke, funkcija`savefig` sprema sliku u zadanom formatu .png. Formati u koje funkcija `savefig` može spremiti slike su eps, pdf, pgf, png, ps, raw, rgba, svg i svgz. Spremanje slika u veći broj formata (npr. jpeg ili jpg) moguće je instalacijom (u slučaju da nije predinstaliran) modula *Python Imaging Library* konzolnom naredbom:

```
pip install pillow
```

Python kod 3.11 Primjer korištenja funkcije `savefig`

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

A = [0, 5, 0, 5]
plt.plot(A, c='lightgreen', linestyle='-.', linewidth=4)

x = np.linspace(0, 3, 100)
y = 2.5 + 2.5*np.cos(np.pi * x)
plt.plot(x, y, c='k', linestyle='--')

plt.savefig('graf')
plt.savefig('graf.pdf')
```

U programskom kodu 3.11 vidi se da je funkcija `savefig` napisana nakon definicije vektora x i y te njihove implementacije u funkciji `plot`. Funkcija `savefig` navedena je dva puta, jednom bez ekstenzije formata u imenu, a jednom s ekstenzijom (pdf). To znači da se slika grafova spremila u dva različita formata (png i pdf), ali pod istim imenom 'graf'. Važno je napomenuti da je zadana lokacija spremanja slike ona u kojoj se nalazi spremljen python kod. U slučaju da je potrebno spremiti sliku na neku specifičnu lokaciju, potrebno je dodati željenu lokaciju uz ime slike:

```
matplotlib.pyplot.savefig('/lokacija1/lokacija2/imeslike')    #linux
matplotlib.pyplot.savefig(r'C:\lokacija1\lokacija2\imeslike') #windows
```

3.5.1 Dodatne postavke pri spremanju grafova

Funkciju `savefig` moguće je pozvati uz još dodatnih argumenata koji omogućuju dodatno prilagođavanja slike koja se želi spremiti. Umjesto postavljanja ekstenzije uz ime slike moguće je i eksplicitno definirati format kao argument u funkciji `savefig` na način:

```
matplotlib.pyplot.savefig('imedatoteke', format='pdf')
```

Uz ime datoteke i format, funkcija `savefig` prima i argumente koje se mogu primijeniti i na svojstva prikaza grafa (dpi, edgecolor, facecolor) opisane u poglavlju 3.4.

3.6 Podgrafovi

Luka Grbčić
Ivana Lučin

Podgrafovi (eng. *subplots*) u modulu PyPlot omogućavaju stvaranje više prostora crtanja na jednom prozoru te je funkcija kojom se kreiraju `subplot`. `subplot` se s argumentima poziva na način:

```
matplotlib.pyplot.subplot(1, 1, 1)
matplotlib.pyplot.plot(x, y)
matplotlib.pyplot.subplot(1, 1, 2)
matplotlib.pyplot.plot(a, b)
```

gdje su argumenti tri broja koja predstavljaju postavke pozicioniranja i selekcije podgrafova. Za svaku funkciju `plot` mora se pridodati funkcija `subplot` ako se krivulja želi prikazati kao podgraf.

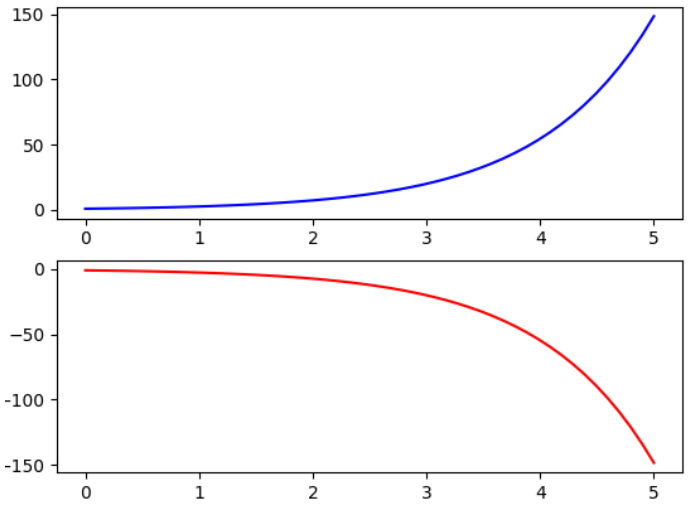
Prvi argument je broj koji na prozoru rezervira broj redova za svaki podgraf dok drugi argument radi istu stvar samo za stupce u prozoru. Na primjer, ako se za prvi argument stavi broj 5, funkcija `subplot` stvara 5 redova na prozoru za podgrafove te ako se za drugi argument stavi broj 3, funkcija stvara 3 stupca za podgrafove. Treći argument je poveznica s funkcijom `plot` i on sekvencijalno prikazuje definirane krivulje kao podgrafove. Svaka krivulja mora imati svoj broj definiran u `subplot` funkciji.

Ako je svaki od brojeva redaka, stupaca i grafova manji od 10, moguće je kao argument postaviti samo kao troznamenkasti broj:

```
matplotlib.pyplot.subplot(311)
```

Python kod 3.12 Implementacija funkcije `subplot`

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

x = np.linspace(0, 5, 50)

y1 = np.exp(x)
y2 = -np.exp(x)

plt.subplot(2, 1, 1)
plt.plot(x, y1, color='b')
plt.subplot(2, 1, 2)
plt.plot(x, y2, color='r')
plt.show()
```



Slika 3.16
Izgled podgrafa funkcije `subplot` iz koda 3.12

U programskom kodu 3.12 su definirane dvije `plot` funkcije te se svaka referira na svoju `subplot` funkciju. S obzirom na to da su potrebna dva podgrafa jer su u kodu definirane dvije krivulje, prvi argument je broj 2 te on dijeli prozor na dva retka, a drugi argument je broj 1 koji rezervira prostor samo za jedan stupac na prozoru. Treći argument je različit za svaku funkciju `subplot` te se on direktno veže na funkciju `plot`. Ako je jednak broju 1, to znači da se na podgrafu crta prva definirana krivulja.

3.6.1 Dodatne postavke podgrafova

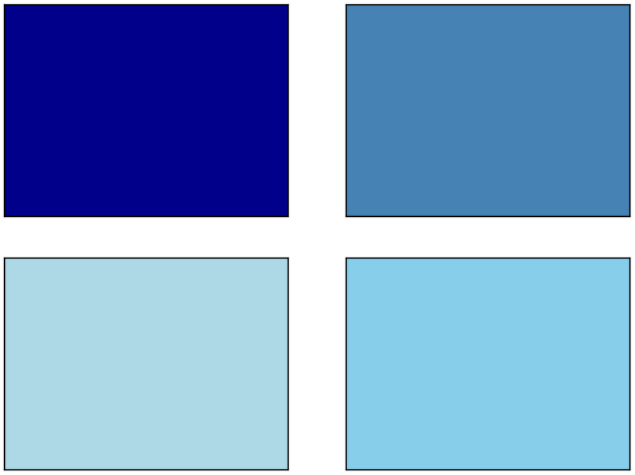
Postoji mogućnost dodavanja dodatnog argumenta u funkciju `subplot`:

```
matplotlib.pyplot.subplot(4, 2, 1, facecolor='b')
```

gdje je `facecolor` argument za boju prostora crtanja opisan u poglavlju 3.4.

Python kod 3.13 Implementacija argumenta `facecolor`

```
import matplotlib.pyplot as plt

def ticks():
    plt.xticks([])
    plt.yticks([])

plt.subplot(2, 2, 1, facecolor='darkblue')
ticks()
plt.subplot(2, 2, 2, facecolor='steelblue')
ticks()
plt.subplot(2, 2, 3, facecolor='lightblue')
ticks()
plt.subplot(2, 2, 4, facecolor='skyblue')
ticks()
plt.show()
```



Slika 3.17
Podgrafovi s različitim bojama prostora crtanja generirani iz programskog koda 3.13

3.6.2 Polarni grafovi

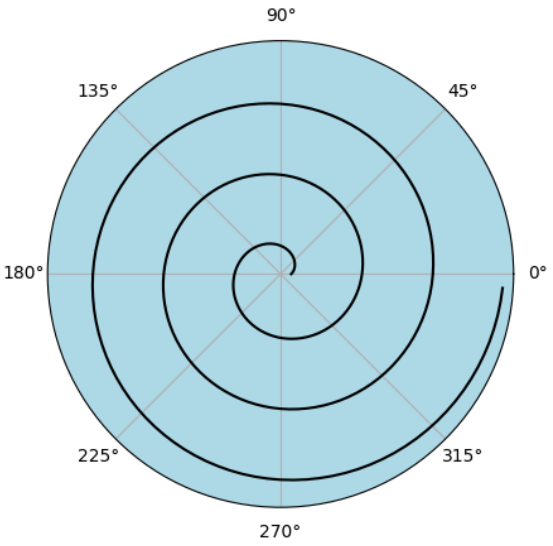
Funkcija `subplot` omogućava crtanje grafa u polarnim koordinatama. Da bi se stvorio polarni graf potrebno je staviti argument u `subplot` funkciju na način:

```
matplotlib.pyplot.subplot(1, 1, 1, projection='polar')
```

gdje su prva tri argumenta brojevi definirani u poglavlju 3.6, a argument `projection` definira način projekcije grafa. Za polarni graf vrijednost argumenta mora biti string `polar`.

Python kod 3.14 Kod za izradu polarnog grafa

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

r = np.arange(0, 3, 0.01)
theta = 2 * np.pi * r

def ticks():
    plt.yticks([])

plt.subplot(111, projection='polar', facecolor='lightblue')
plt.plot(theta, r, color='k')
ticks()

plt.show()
```



Slika 3.18
Polarni graf generiran programskim kodom 3.14

U programskom kodu 3.14 je vidljivo da je logika izrade grafa jednaka onoj kada se ne radi o polarnom grafu, ali je projekcija promijenjena te su osi dodatnim argumentom `polar` funkcije `subplot` prilagođene polarnim koordinatama (vidljivo na Slici 3.18).

3.6.3 Tabularno organizirani grafovi (GridSpec)

Ukoliko prozor (`figure`) zamislimo kao dvodimenzionalnu mrežu ili matricu s jasno definiranim brojem redaka i stupaca, odabirom odgovarajuće konfiguracije redak/stupac moguće je jednoznačno pozicionirati prostor crtanja (`axes`). Funkcija `add_gridspec` omogućava inicijalizaciju takve temeljne mreže pozivom:

```
fig = plt.figure(constrained_layout=True)
grid = fig.add_gridspec(2, 3)
```

Argument `constrained_layout` ključan je za održavanje razmaka između podgrafova. Razmak će biti automatski reguliran u slučaju da vrijedi `True`, u protivnom je poželjno zadati relativan položaj s obzirom na okvir prozora i susjedne grafove. Funkcija `add_gridspec` u prozor upisuje mrežu (matricu) s zadanim brojem redaka i stupaca. Dodavanje podgrafova vrši se funkcijom `add_subplot` koja može primiti proizvoljni raspon redaka i stupaca iz mreže definirane s `add_gridspec`. Indeksiranje se vrši analogno NumPy poljima (2.2), a moguće je koristiti i raspone:

```
ax1 = fig.add_subplot(grid[1:, :-2])
```

Dimenzije podgrafova moguće je regulirati s `width_ratios` i `height_ratios`, parametrima koji primaju liste relativnih dimenzija pojedinih podgrafova u mreži:

```
fig.add_gridspec(4, 3, width_ratios=[1, 4, 2], height_ratios=[2, 5, 3, 6])
```

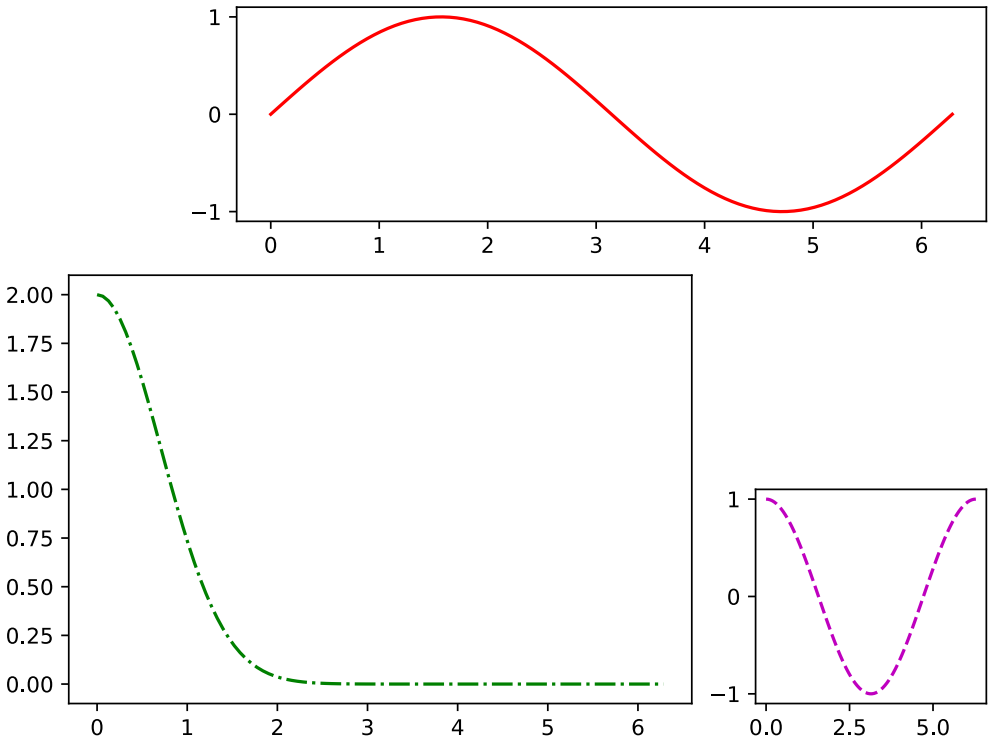
Međusobni odnos te položaj mreža podgrafova u prozoru definira se grupom parametara. Osnovni preduvjet negacija je argumenta `constrained_layout`. Parametri `wspace` i `hspace` uvjetuju vertikalni odnosno horizontalni razmak između podgrafova. Analogno, `left`, `right`, `bottom` i `top` definiraju margine mreže s obzirom na okvir prozora. Svi parametri izražavaju se relativnim dimenzijama u intervalu od $[0,1]$ pri čemu mora vrijediti `left` $<$ `right` i `bottom` $<$ `top`. Primjerice:

```
fig.add_gridspec(nrows, ncols, wspace=.05, hspace=.05,
                                left=.1, right=.9, bottom=.1, top=.9)
```

Primjena `GridSpec` modula prikzana je u programskom kodu 3.15:

Python kod 3.15 Temeljna primjena `GridSpec`-a

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

figure = plt.figure(constrained_layout=True)

# kreiraj gridspec mrežu s 3 retka i 5 stupaca
grid = figure.add_gridspec(3, 5)

# dodaj podgrafove s add_subplot
# za svaki podgraf definiraj raspon redaka/stupaca iz mreže grid
# podgraf ax1: prvi redak, svi stupci počevši s drugim iz mreže grid
ax1 = figure.add_subplot(grid[0, 1:])
# podgraf ax2: drugi redak, svi stupci izuzev posljednja 2 iz mreže grid
ax2 = figure.add_subplot(grid[1:, :-2])
# podgraf ax3: treci redak, svi stupci izuzev posljednja 2 iz mreže grid
ax3 = figure.add_subplot(grid[2, -2:])

x = np.linspace(0, 2*np.pi, 100)

# u definirane podgrafove ax1, ax2, ax3 ucrtaj krivulje
ax1.plot(x, np.sin(x), 'r-')
ax2.plot(x, 2*np.exp(-x**2), 'g-.')
ax3.plot(x, np.cos(x), 'm--')
```

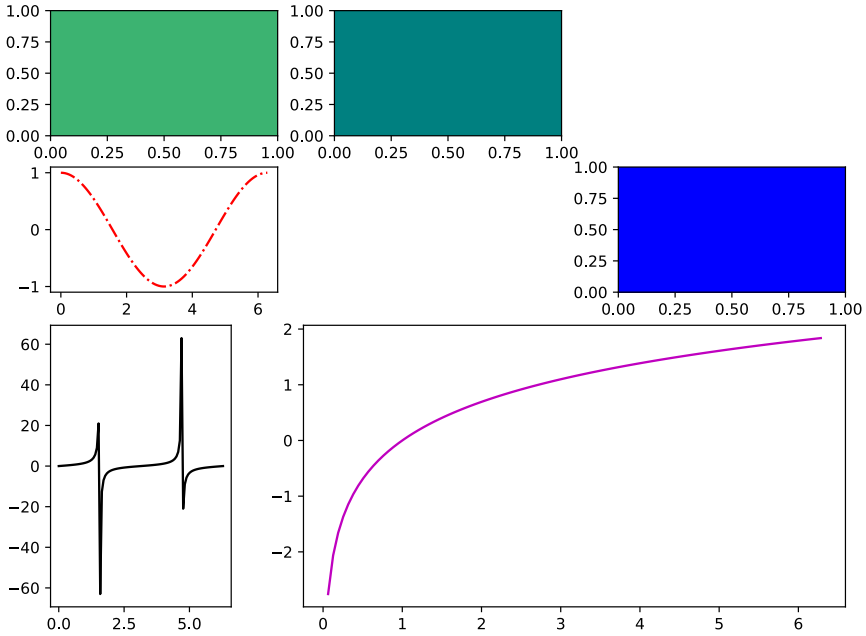


Slika 3.19 Podgrafovi i funkcije kreirane programskim kodom 3.15

Naprednije implementacije GridSpec-a mogu uključuivati više mreža u okviru jednog prozora. Ovaj pristup primjenjen je u kodu 3.16:

Python kod 3.16 Viešestruke GridSpec mreže s podgrafovima

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

# nužno constrained_layout=False
figure = plt.figure(figsize=(8, 6), constrained_layout=False)

# parametri relativnih dimenzija podgrafova
width = [1, 3]
height = [2, ]

# gridspec 1 mreža s relativnim dimenzijama i marginama
grid1 = figure.add_gridspec(1, 2, width_ratios=width, height_ratios=height,
                                    left=.05, right=.95, top=.475, bottom=.05)
# gridspec 2 mreža s propisanim marginama i razmacima
grid2 = figure.add_gridspec(2, 3, wspace=.25, hspace=.25,
                                    left=.05, right=.95, top=.95, bottom=.525)

x = np.linspace(0, 2*np.pi, 100)

# dodavanje podgrafova u "grid1"; odabir retka i stupca iz mreže "grid1"
ax1 = figure.add_subplot(grid1[0,0])
ax1.plot(x, np.tan(x), 'k')
ax2 = figure.add_subplot(grid1[0,1])
ax2.plot(x[1:], np.log(x[1:]), 'm')

# dodavanje podgrafova u "grid2"; odabir retka i stupca iz mreže "grid2"
ax3 = figure.add_subplot(grid2[0,0], facecolor='mediumseagreen')
ax4 = figure.add_subplot(grid2[0,1], facecolor='teal')
ax5 = figure.add_subplot(grid2[1,2], facecolor='blue')
ax6 = figure.add_subplot(grid2[1,0])
ax6.plot(x, np.cos(x), 'r-.')
```



Slika 3.20 Podgrafovi kreirani programskim kodom 3.16

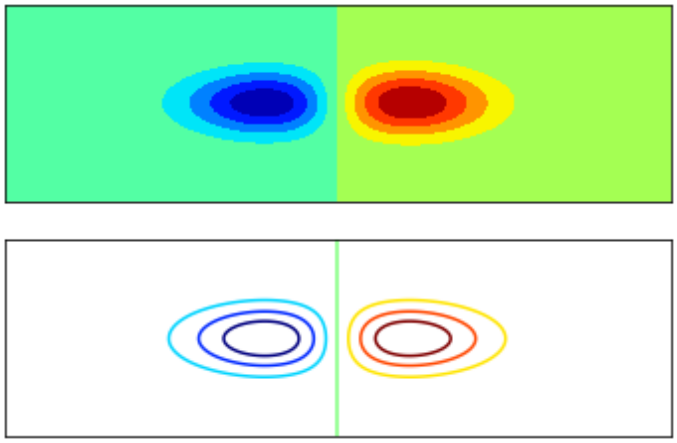
3.7 Grafovi funkcija dvije varijable

Luka Grbčić
Ivana Lučin

Do sada, u prethodnim poglavljima vizualizacije podataka, detaljno su predstavljene mogućnosti prikaza podataka koji su definirani s dvije koordinate (x,y) te formiraju 2D linijske grafove.

U slučaju da je potrebno prikazati podatke koji su definirani s tri koordinate (x,y,z), odnosno funkciju koja ovisi o dvije nezavisne varijable $f(x,y)$, PyPlot modul može pomoću izolinija kreirati dvodimenzionalni graf iz trodimenzionalnih podataka. Korištenjem boja i nijansi boja moguće je razlikovati vrijednosti podataka tlocrtnim pogledom na graf.

Funkcije kojima se izrađuju izolinije u PyPlot modulu su `contour` i `contourf` te je razlika između dvije navedene funkcije ta da `contour` samo linijama pridodaje odgovarajuću boju te ostavlja bijelu pozadinu prostora crtanja, a `contourf` bojom ispunjava i površinu između linija za odgovarajuću vrijednost točke.



Slika 3.21
Razlika između `contourf` (gore) i `contour` (dolje) funkcije

Obje funkcije na isti način primaju podatke kao argumente. Jedine razlike u argumentima koje postoje se odnose na postavke prikaza grafa (boje, linije). Podaci se definiraju na način:

```
matplotlib.pyplot.contour(M)
matplotlib.pyplot.contourf(M)
```

gdje je argument matrica u kojoj se nalaze podaci za prikaz. Različite vrijednosti elemenata matrice poprimaju različitu boju te se tako može prikazati promjena po trećoj, vertikalnoj osi.

Vrijednost podataka po nivou se automatski raspodijeli, ali se taj parametar može detaljnije specificirati dodavanjem argumenta u funkciju. Argument koji definira raspodjelu nivoa može biti cijeli broj kojim se odredi koliko će funkcija `contour` ili `contourf` različitih nivoa generirati:

```
matplotlib.pyplot.contour(M, 3)
matplotlib.pyplot.contourf(M, 3)
```

Na primjer, ako se postavi cijeli broj 3 nakon matrice podataka, funkcija automatski stvara tri izolinije koje imaju različiti nivo. Kod funkcije `contourf` veći broj stvara glađi prijelaz između izolinija.

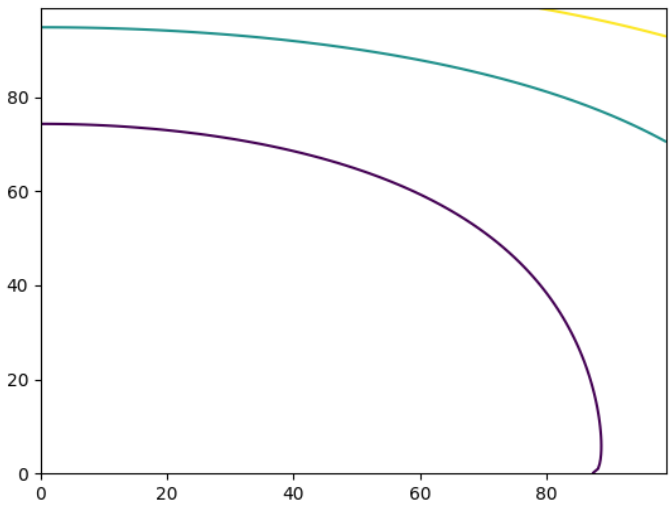
Drugi način regulacije raspodjele nivoa vrijednosti podataka može biti argumentom:

```
matplotlib.pyplot.contour(M, v)
matplotlib.pyplot.contourf(M, v)
```

gdje je drugi argument nakon matrice vektor svih vrijednosti nivoa za koje će se kreirati izolinije. Vrijednosti u tom vektoru moraju rastu od najmanje do najveće jer u suprotnom funkcije `contour` i `contourf` ne kreiraju izolinije te izbacuju grešku.

Python kod 3.17 Implementacija funkcije `contour` s argumentom broja izolinija

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

def f(x, y):
    return np.exp(y) + x**2 - y**0.5

x = np.linspace(0, 3, 100)
y = np.linspace(0, 3, 100)
X, Y = np.meshgrid(x, y)

M = f(X, Y)

plt.contour(M, 3)
plt.show()
```



Slika 3.22
Prikaz izolinija generiranih programskim kodom 3.17

U programskom kodu 3.17 se vidi da je kao argument zadana matrica M te broj tri koji kreira izolinije za tri različita nivoa kao što je prikazano na Slici 3.22. Na svakoj od tri izolinije su sve vrijednosti jednake što je vidljivo jer je boja jednaka.

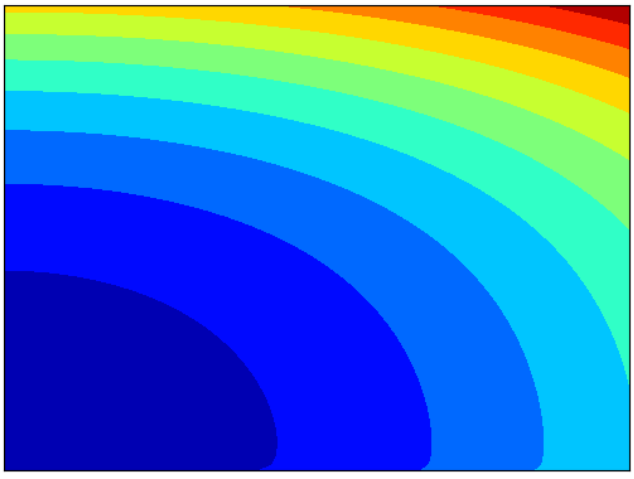
U funkcijama `contour` i `contourf` moguće je definirati i točne koordinatu za sve vrijednosti koje se žele prikazati. Koordinate se definiraju kao dva argumenta koji se moraju postaviti prije matrice vrijednosti vertikalne osi:

```
matplotlib.pyplot.contour(X, Y, M)
matplotlib.pyplot.contourf(X, Y, M)
```

gdje su prva dva argumenta matrice čiji elementi zajedno formiraju x i y koordinatu, odnosno jednu točku u koordinatnom sustavu. Može se gledati na sva tri argumenta kao matrice koje formiraju x, y i z koordinate, gdje se vrijednosti osi z razlikuju bojama. Da bi funkcija kreirala graf, sve tri matrice moraju biti iste duljine. Za kreiranje matrica za x i y koordinate iz vektora, vrlo je dobro koristiti funkciju `meshgrid` iz modula NumPy opisanu u poglavlju 2.1.8.

Python kod 3.18 Implementacija funkcije `contourf` s argumentom x i y koordinata

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

def f(x, y):
    return np.exp(y) + x**2 - y**0.5

def ticks():
    return plt.xticks([]), plt.yticks([])

x = np.linspace(0, 3, 100)
y = np.linspace(0, 3, 100)
X, Y = np.meshgrid(x, y)
M = f(X, Y)

plt.contourf(X, Y, M, 10, cmap = 'jet')
ticks()
plt.show()
```

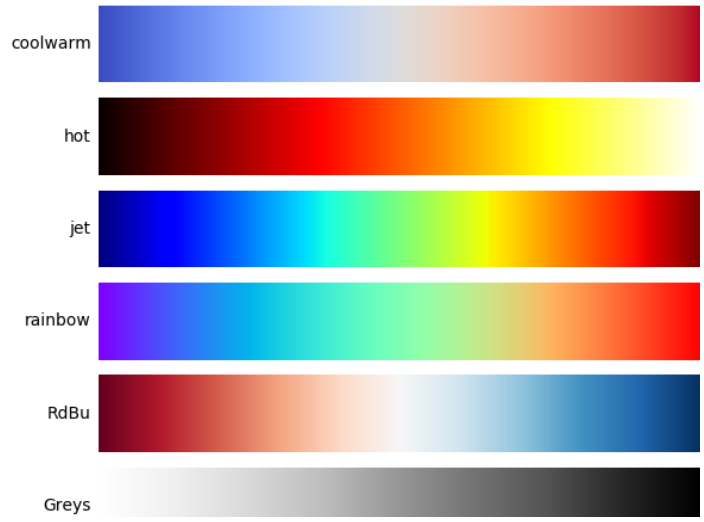


Slika 3.23
Prikaz izolinija i popunjenih površina generirane programskim kodom 3.18

3.7.1 Mapa boja

S obzirom na to da su boje glavni alat kojim se mogu 3D podaci prikazati na 2D grafovima, mapa boja igra vrlo važnu ulogu. Mapa boja (eng. *colormap*) je način na koji se numerička vrijednost iz intervala $[0,1]$ pretvara u boju. Stvara se interval boja koje imaju svoju RGBA (*Red, Green, Blue, Alpha*) specifikaciju, točnije udio crvene, zelene i plave boje te prozirnosti.

Kada se podaci žele vizualizirati u `contourf` ili `contour` funkciji, automatski se generira njihova raspodjela u interval raspona $[0,1]$ te s obzirom na to da je mapa boja u jednakom intervalu, svaka vrijednost dobije odgovarajuću boju. Knjižnica `matplotlib` sadrži predefinirane mape boja koje se nalaze u modulu `matplotlib.pyplot.cm`.



Slika 3.24
Neke od preddefinarnih mapa boja u knjižnici `matplotlib`

Osim boja navedenih na Slici 3.24 postoji mnoštvo drugih mapa koje se mogu pronaći u dokumentaciji knjižnice `matplotlib` (`https://matplotlib.org/users/colormaps.html`).

Implementacija mape boja se vrši dodavanjem argumenta `cmap` u funkcijama `contourf` i `contour` na način:

```
matplotlib.pyplot.contourf(X, Y, M, cmap = 'coolwarm')
```

gdje argument `cmap` prima string preddefinirane mape boja. Korisno je znati da se mapa boja može obrnuti dodatkom "_r" na vrijednost string argumenta `cmap` na način:

```
matplotlib.pyplot.contourf(X, Y, M, cmap = 'coolwarm_r')
```

gdje string *coolwarm_r* prikazuje obrnutu mapu boja *coolwarm*.

3.7.2 Dodatne postavke izolinjskih grafova

Umjesto korištenja mapa boja, moguće je boje izolinija i površina eksplicitno definirati argumentom `colors`:

```
matplotlib.pyplot.contour(X, Y, M, colors='blue')
matplotlib.pyplot.contourf(X, Y, M, colors=['r', 'g', 'b'])
```

gdje `colors` može primati string vrijednosti za boje definirane u poglavlju 3.3 ili vektor boja koje će se primijeniti na izolinije definiranim redoslijedom.

Moguće je dodati transparentnost boja i linija koja je definirana u poglavlju 3.3.3:

```
matplotlib.pyplot.contourf(X, Y, M, cmap='jet', alpha=0.4)
```

Argumenti koji su specifični za funkciju `contour` se odnose na izgled linija:

```
matplotlib.pyplot.contour(M, cmap='Greys', linewidths=4, linestyles=':')
matplotlib.pyplot.contour(M, cmap='jet', linewidths=[1, 2], linestyles=[':', '-.'])
```

gdje argument `linewidths` definira debljinu linija te može biti zadan kao jedan broj koji će se primijeniti na sve izolinije, ali i kao vektor može definirati više debljina za izolinije. Drugi argument je `linestyles` kojim se određuje stil linije korištene u grafu,

može isto biti zadan samo kao jedan string, ali i kao vektor stringova. Vrijednosti stringova za tipove linija su iste onima opisanim u tablici 3.1. Navedeni argumenti su vrlo koristan alat jer omogućuju bolje razlikovanje izolinija.

Funkcija `contourf` također ima argument koji je specifičan samo za nju, a služi za postavljanje tekstura na površine između izolinija i može biti vrlo koristan kada se crtaju crno bijeli grafovi. Argument se implementira na način:

```
matplotlib.pyplot.contourf(X, Y, M, cmap='Greys', hatches='\\')
matplotlib.pyplot.contourf(X, Y, M, cmap='Greys', hatches=['/','//','.'])
```

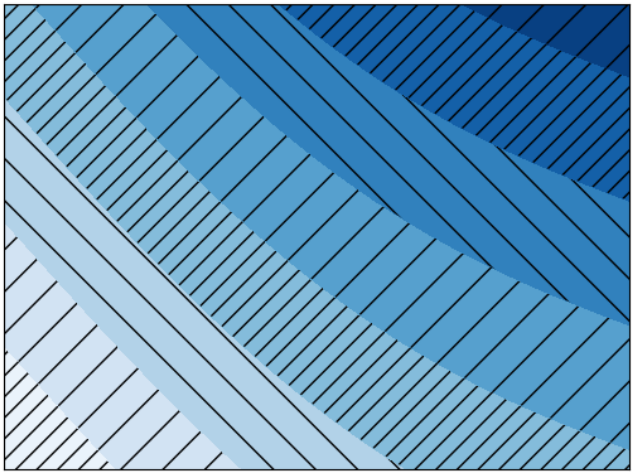
U `hatches` se postavljaju simboli koji se žele primijeniti kao teksture. Moguće je postaviti samo jedan string koji će biti na svim površinama, ali je moguće definirati i kao vektor stringova za više površina između izolinija.

Tablica 3.5 Teksture

Simbol	Tekstura
`'/'`	Kosa crta
`'´`	Obrnuta kosa crta
`'|'`	Vertikalna linija
`'-'`	Horizontalna linija
`'+'`	Plus
`'x'`	Križ
`'o'`	Mali krug
`'O'`	Veliki krug
`'.'`	Točka
`'*'`	Zvijezda

Python kod 3.19 Primjer `contourf` funkcije s argumentom `hatches` za teksture

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

def f(x, y):
    return x + y + 0.5*np.sin(x)

def ticks():
    return plt.xticks([]), plt.yticks([])

x = np.linspace(0, 3, 100)
y = np.linspace(0, 3, 100)
X, Y = np.meshgrid(x,y)

plt.contourf(X, Y, f(X,Y), cmap='Blues', hatches=['//','/', '\\'])
plt.show()
```



Slika 3.25
Prikaz tekstura na površini između izolinija, programski kod 3.19

3.7.3 Korisni alati uz prikaz izolinijskih grafova

Za bolje razumijevanje grafova kreiranih `contourf` i `contour` funkcijama dobro je iskoristiti dvije funkcije u modulu PyPlot. Radi se o funkcijama `clabel` i `colorbar`.

Funkcija `clabel` se koristi za dodavanje oznaka na izolinije i direktno je povezana s funkcijom `contour`. Implementira se na način:

```
matplotlib.pyplot.clabel(contour(X, Y, M))
```

gdje je argument funkcija `contour`. Funkcija `clabel` ispisuje na izolinijama brojčane vrijednosti tih istih izolinija kao tekst. Moguće je dodatno podesiti izgled oznaka na način:

```
matplotlib.pyplot.clabel(contour(M), inline=True, fontsize=5, colors='r', fmt='%.0f')
```

nakon funkcije `contour` je argument `inline` kojim se definira hoće li se osloboditi dio izolinije da bi stao tekst te je zadana vrijednost da hoće. Argument `fontsize` služi za određivanje veličine fonta teksta koji će se prikazati na izolinijama te može biti jednak broju za apsolutno postavljanje veličine ili stringu (npr. *'large'*) za relativno postavljanje veličine. Argumentom `fmt` se postavlja dužina zapisa decimalne točke na izoliniji, a argumentom `colors` boja koja je opisana u poglavlju 3.7.2.

Druga korisna funkcija `colorbar` služi za dodavanje table boja i vrijednosti pored prostora crtanja grafa. Uz tablu boja lakše je brojčano razlikovati vrijednosti boja na slici. Funkcionira tako da se se mapa boja formira u jedan stupac te se vrijednosti podataka dodaju na granice prijelaza boja. Implementira se na način:

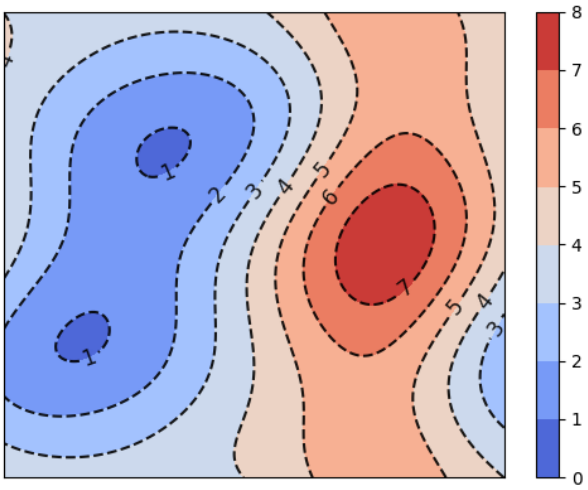
```
matplotlib.pyplot.colorbar()
matplotlib.pyplot.colorbar(orientation='horizontal')
```

Nije potrebno staviti niti jedan argument što znači da se tabla boja može automatski generirati na temelju podataka u grafu. Zadana pozicija table boja je vertikalno s desne strane grafa, ali moguće je dodati argument `orientation` za orijentaciju table boja.

Python kod 3.20 Izolinijski graf s `clabel` i `colorbar` funkcijama

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

```
def f(x, y):
    return 2 + np.sin(x) + np.cos(y)**2 + np.sin(x-y) + np.exp(np.sin(x))

x = np.linspace(-3, 3, 200)
y = np.linspace(-3, 3, 200)
X, Y = np.meshgrid(x,y)
Z = f(X, Y)

plt.grid()

isolines = plt.contour(X, Y, Z, colors='black', alpha=0.9, linewidths=1.5,
                       linestyles='--')

plt.contourf(X, Y, Z, cmap='coolwarm')
plt.clabel(isolines, inline=True, fmt='%.0f', fontsize='large', colors='black')
plt.colorbar(orientation='vertical')

plt.show()
```



Slika 3.26
Izolinijski graf s tablom boja i oznakama na izolinijama, programski kod 3.20

U programskom kodu 3.20 su iskorištene obje navedene funkcije. Važno je napomenuti da je funkcija `contour` proizvoljno spremljena u stvorenu varijablu *isolines* te onda postavljena kao argument u funkciju `clabel`.

3.8 Animacije

Ivana Lučin
Luka Grbčić

Modul za izradu animacija unutar knjižnice matplotlib je `animation`. Funkcija koja omogućava kreiranje animacija je `FuncAnimation`. Funkcija radi na principu uzastopnog pozivanja funkcije koja definira promjenu na prostoru crtanja. Primjer poziva animacije:

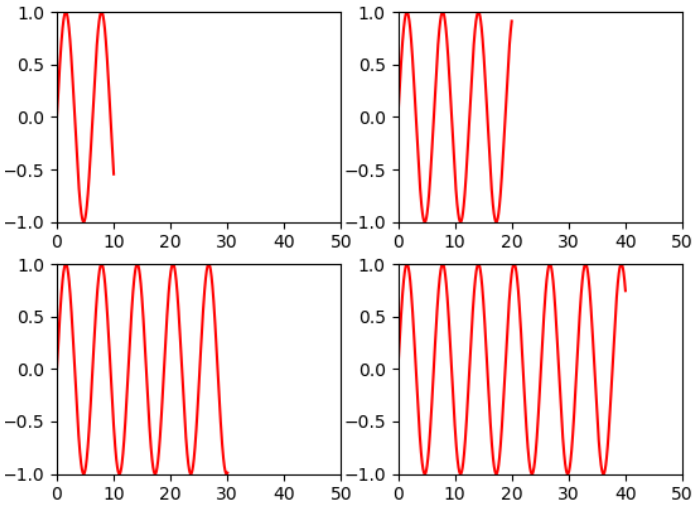
```
matplotlib.animation.FuncAnimation(fig, update_animation, frames=50)
```

U ovom slučaju `fig` predstavlja poveznicu na prostor crtanja koji definira korisnik, update_animation je naziv funkcije čijom definicijom korisnik definira na koji način će se prostor crtanja ažurirati te broj 50 predstavlja broj poziva te funkcije.

Python kod 3.21 Animacija funkcije sinusa

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

def update_line(t):
    x = np.linspace(0, t, 100*t)
    y = np.sin(x)
    line.set_data(x, y)
    return line

fig1 = plt.figure()
plt.xlim(0, 50)
plt.ylim(-1, 1)
line, = plt.plot([], [], 'r')

line_ani = animation.FuncAnimation(fig1,update_line, frames=50)
plt.show()
```

U programskom kodu 3.21 prikazan je jednostavan primjer animacije koja iscrtava sinusnu funkciju. Funkcija `update_line` kao argument prima broj poziva funkcije koji je definiran u `FuncAnimation` te u ovom slučaju iznosi 50. Naredbom `line.set_data` gdje je `line` poveznica s prostorom crtanja, pri svakom pozivu funkcije definira se novi set podataka koji se prikazuju na grafu.



Slika 3.27
Animacija sinus funkcije

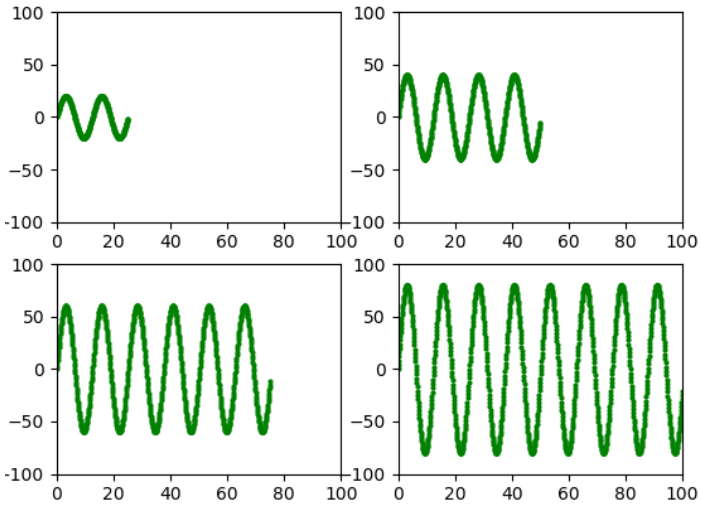
Funkciji se također mogu definirati dodatni argumenti:

```
matplotlib.animation.FuncAnimation(fig,update_animation,frames=500, interval=20,
                init_func=funkcija, repeat=False, fargs=[1,2])
```

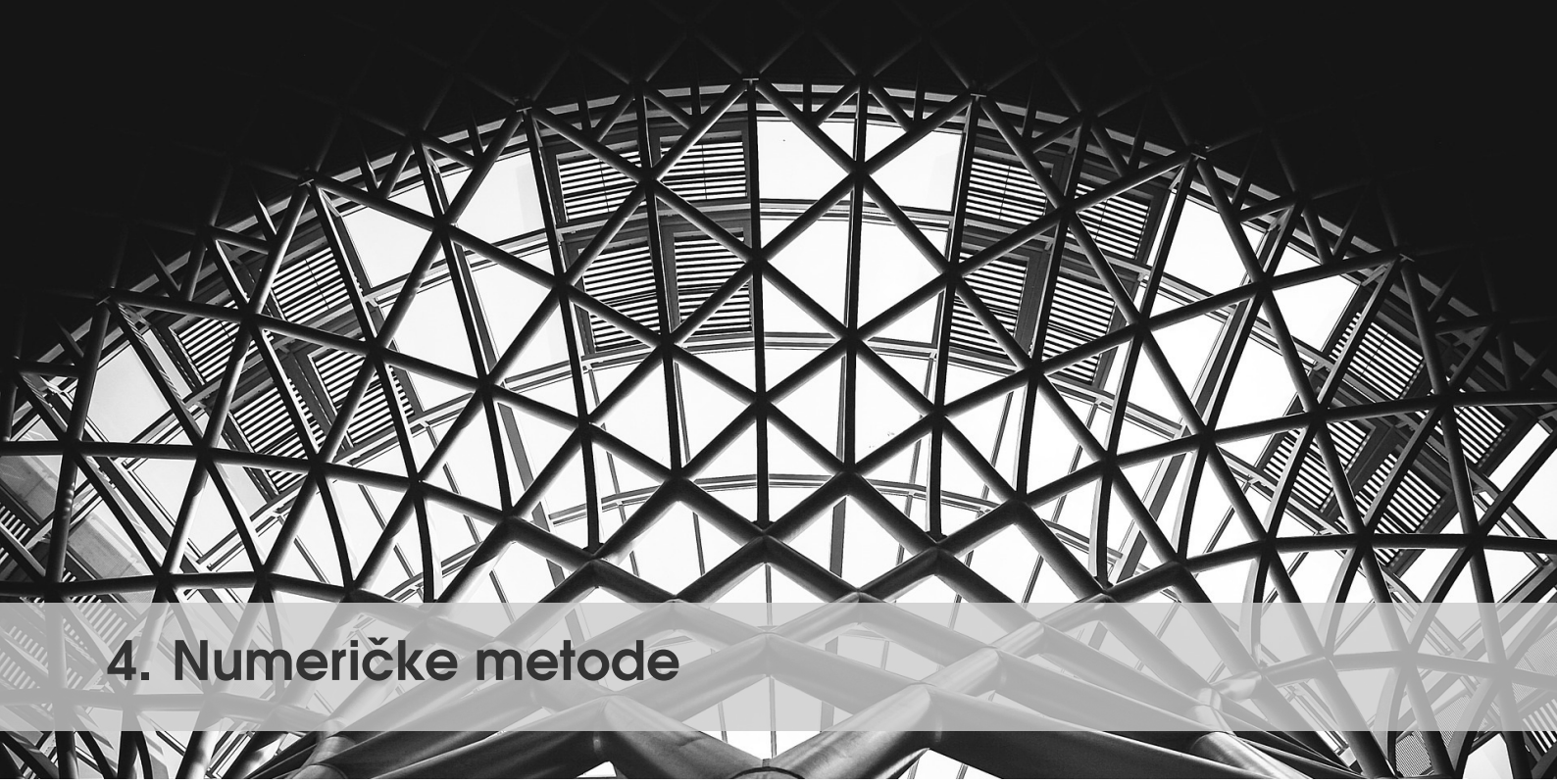
Python kod 3.22 Naprednije postavke animacije

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

def update_line(t,koef):
    x = np.linspace(0, t, 10*t)
    y = koef*t * np.sin(x/2)
    line.set_data(x, y)
    return line

def plot():
    plt.xlim(0, 100)
    plt.ylim(-100, 100)

fig1 = plt.figure()
line, = plt.plot([], [], 'go', markersize=2)

k = 0.8
line_ani = animation.FuncAnimation(fig1, update_line, frames=100, init_func=plot,
                                   interval=50, fargs=[k], repeat=False)

plt.show()
```

Gdje `interval` definira razmak između dva prikaza slike izražen u milisekundama. Argument `init_func` poziva funkciju u kojoj korisnik može definirati izgled prostora crtanja. Argumentom `repeat` može se zaustaviti ponavljanje animacije. Argumentom `fargs` mogu se definirati dodatni argumenti `update_line` ukoliko ona zahtjeva više argumenata. Primjena navedenih argumenata vidljiva je u programskom kodu 3.22.



Slika 3.28
Animacija sinus funkcije s naprednijim postavkama animacije



4. Numeričke metode

Rješavanje nelinearne jednadžbe
Interpolacija
Regresijska analiza
Numerička integracija
Rješavanje sustava linearnih jednadžbi
Fourierova analiza
Rastav na singularne vrijednosti

4.1 Rješavanje nelinearne jednadžbe

Stefan Ivić
Siniša Družeta

U inženjerstvu se često suočavamo sa nelinearnim jednadžbama sa jednom nepoznanicom. Uzmimo na primjer jednadžbu:

$$\log(x+1) = 3 \tag{4.1}$$

Kako je možemo riješiti? Svakako možemo provesti analitički postupak:

$$10^{\log(x+1)} = 10^3 \tag{4.2}$$

$$x + 1 = 10^3 \tag{4.3}$$

$$x = 999 \tag{4.4}$$

No, nisu sve nelinearne jednadžbe tako jednostavne. Uzmimo, na primjer, ovu:

$$(\sin x)^{x+3} - \ln x = \frac{3x^2 - 5x}{x\tan(1-x)}. \tag{4.5}$$

Očito je da nemamo jednostavan analitički postupak koji bi nas doveo do rješenja.

Postavlja se pitanje možemo li definirati metodu pomoću koje se može riješiti bilo koju nelinearnu jednadžbu, pa makar i približno?

> **Napomena**
> Sjetimo se da u inženjerstvu nesavršenost rješenja u principu nikada nije problem, jer se sve inženjerske veličine realno računaju samo do neke potrebne točnosti. Dok god kontroliramo točnost, možemo biti potpuno zadovoljni sa nesavršenim, tj. približnim rješenjem.

Može se primijetiti da se problem rješavanja nelinearne jednadžbe svodi na traženje vrijednosti x^*, koja predstavlja nul-točku funkcije $f(x)$, tj.

$$f(x^*) = 0, \tag{4.6}$$

gdje funkcija $f(x)$ predstavlja jednadžbu koje rješavamo. To je moguće jer se svaku jednadžbu može svesti na oblik u kojem na jednoj strani ima samo nulu.

Konkretno za naš primjer, jednadžbu (4.5) možemo uvijek zapisati kao

$$(\sin x)^{x+3} - \ln x - \frac{3x^2 - 5x}{x\tan(1-x)} = 0, \tag{4.7}$$

gdje sada lijevu stranu jednadžbe možemo proglasiti funkcijom $f(x)$:

$$f(x) = (\sin x)^{x+3} - \ln x - \frac{3x^2 - 5x}{x\tan(1-x)}, \tag{4.8}$$

a sama jednadžba (4.5) se sada pretvara u

$$f(x) = 0, \tag{4.9}$$

čije rješenje je nul-točka x^*.

Dakle, rješavanje nelinearne jednadžbe s jednom nepoznanicom je potpuno ekvivalentno traženju nul-točke funkcije i za tu namjenu imamo nekoliko numeričkih metoda na raspolaganju.

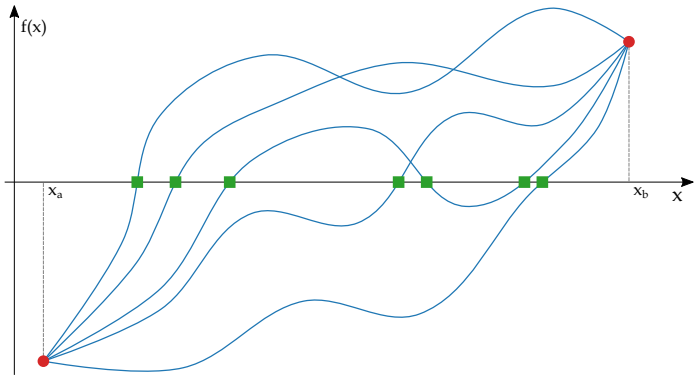
Napomena
Važno je napomenuti da numeričke metode za rješavanje nelinearne jednadžbe ili za pronalazak nul-točke funkcije "ne znaju" kako funkcija izgleda već samo mogu "zatražiti" evaluaciju funkcije u nekim točkama. Cilj je da se pronađe nul-točka sa što manje evaluacija funkcije tj. u što manje koraka metode, pošto funkcija čija se nul-točka traži može biti iznimno zahtjevna za evaluaciju (npr. računalna simulacija).

4.1.1 Metoda bisekcije

Metoda bisekcije traži nul-točku funkcije unutar unaprijed poznatih ograda $x_a < x^* < x_b$. U praksi najčešće nije nikakav problem odrediti ograde x_a i x_b odnosno najmanju i najveću očekivanu vrijednost za neku veličinu. Međutim, odabir početnog intervala ograđivanja $[x_a, x_b]$ treba zadovoljavati uvjet da funkcijska vrijednost u granicama intervala ima suprotan predznak, što se može matematički izreći kao

$$f(x_a) \cdot f(x_b) < 0. \tag{4.10}$$

Ukoliko je zadovoljen uvjet (4.10), a funkcija $f(x)$ je neprekidna na promatranom intervalu onda možemo biti sigurni da je rješenje (nul-točka) zaista unutar ograda $x_a < x^* < x_b$ (Slika 4.1). Ako funkcija ima prekid na početnom intervalu $[x_a, x_b]$, iako na tom intervalu može postojati nul-točka, nema garancije da ona zaista postoji. Metoda bisekcije, kao i druge metode ograđivanja, uz dobar odabir granica x_a i x_b koje zadovoljavaju izraz (4.10) sigurno može uspješno odrediti nul-točku neprekidne funkcije.



Slika 4.1
Ukoliko je funkcija neprekidna te mijenja predznak na intervalu $[x_a, x_b]$, onda sigurno ima barem jednu nul-točku unutar intervala.

Osnovna ideja metode bisekcije je uzastopno smanjivanje intervala kojim ograđujemo rješenje, do trenutka kada nam interval unutar kojeg znamo da se nalazi rješenje ne postane toliko uzak da praktički možemo bilo koju točku unutar njega usvojiti kao rješenje.

Interval unutar kojeg tražimo rješenje ćemo smanjiti tako da ćemo ga raspoloviti (tj. izračunat ćemo srednju točku intervala):

$$x_c = \frac{x_a + x_b}{2}. \tag{4.11}$$

Za novu točku x_c izračunat ćemo vrijednost funkcije $f(x_c)$ te primjenom ideje iz izraza (4.10) provjeriti da li je rješenje u lijevom podintervalu $[x_a, x_c]$ ili u desnom podintervalu $[x_c, x_b]$. Dovoljno je provjeriti samo jedan podinterval, recimo lijevi:

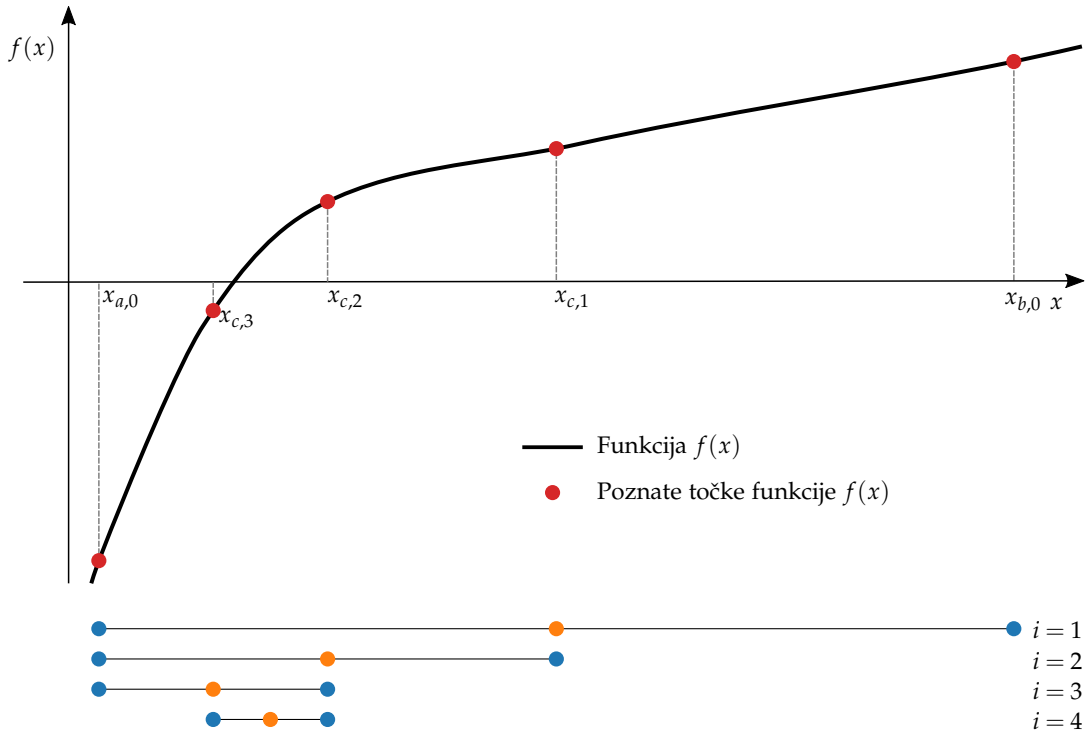
$$f(x_a) \cdot f(x_c) < 0, \tag{4.12}$$

s tim da potvrda ove tvrdnje znači da se nul-točka nalazi u lijevom (testiranom) podintervalu, a njena negacija da se nul-točka nalazi u drugom (desnom) podintervalu.

Sukladno rezultatu ovog testa, kao novi interval unutar kojeg ćemo u sljedećem ciklusu metode tražiti nul-točku izabiremo:

- $[x_a, x_c]$ ako je $f(x_a) \cdot f(x_c) < 0$ odnosno
- $[x_c, x_b]$ ako je $f(x_b) \cdot f(x_c) < 0$.

Za tako izabrani novi interval sigurni smo da ima iste osobine kao početni odnosno da sigurno sadrži barem jednu nul-točku. Sada ponavljamo opisani postupak: određujemo novo polovište x_c, provjeravamo da li je rješenje u lijevom ili desnom podintervalu i sužavamo interval unutar kojeg tražimo nul točku (Slika 4.2). Ovakve ponavljajuće postupke zovemo **iterativnim** postupcima, a jedan ciklus takvog postupka zove se **iteracija**.



Slika 4.2 Primjer metode bisekcije: određivanje srednjih točaka te sužavanje promatranog intervala

Ponavljanje postupka se vrši dok se recimo srednjom točkom x_c dovoljno ne približimo točnom rješenju, tj. do trenutka (odnosno iteracije i) kada se desi da vrijedi:

$$|f(x_c^{(i)})| < e_{max}, \tag{4.13}$$

gdje je e_{max} neka unaprijed dopuštena pogreška iznosa funkcije u aproksimaciji nul-točke. Tada možemo srednju točku x_c usvojiti kao približno rješenje $f(x_c^{(i)}) \approx 0$, tj. $x^* \approx x_c^{(i)}$.

Drugi uvjet zaustavljanja, često korišten u svim iterativnim metodama, je uvjet maksimalnog broja iteracija. Kod tog uvjeta iterativni postupak se ponavlja dok je $i < i_{max}$. Uvjeti zaustavljanja iterativnih metoda se gotovo uvijek kombiniraju, pri čemu se iteracijski postupak zaustavlja čim se zadovolji barem jedan od više ugrađenih uvjeta zaustavljanja.

Programski kod 4.1 u Pythonu implementira metodu bisekcije za traženje nul-točke funkcije $f(x) = 2x^2 - e^x + 4$.

Python kod 4.1 Metoda bisekcije

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

# Definicija funkcije
def f(x):
    return 2.0 * x**2.0 - np.exp(x) + 4.0

# Zahtjevaj unos, pretvori uneseni tekst u brojeve
# Dobre granice [-5, 4]
xa = float(input('Unesi lijevu granicu: '))
xb = float(input('Unesi desnu granicu: '))
eps = float(input('Unesi dopustenu pogresku: '))

# Provjeri mijenja li funkcija predznak na danom intervalu
if f(xa) * f(xb) < 0.0:
    print('Ispravne granice: funkcija f(x) mijenja predznak')
else:
    print('Neispravne granice: funkcija f(x) ne mijenja predznak')

xplot = np.linspace(xa, xb, 300)
plt.plot(xplot, f(xplot)) # Crtanje funkcije
plt.axhline(0, c='r') # Crtanje x osi
plt.axhline(eps, c='g', lw=0.5) # Crtanje gornje granice tolerancije
plt.axhline(-eps, c='g', lw=0.5) # Crtanje donje granice tolerancije
plt.plot([xa, xb], [f(xa), f(xb)], 'gs') # Početne točke

maxit = 1000 # Maksimalni broj iteracija
it = 0 # Trenutna iteracija

while it < maxit: # Iteriraj do maksimalnog broja iteracija
    it += 1
    xc = (xa + xb) / 2.0 # Točka bisekcije

    # Crtanje točaka
    plt.plot(xc, f(xc), 'ro')
    plt.annotate(str(it), xy=(xc, f(xc)))

    err = np.abs(f(xc))
    if err < eps:
        break # Ukoliko je greška manja od dopuštene, prekini iteracijski postupak
    elif f(xc) * f(xb) < 0.0:
        xa = xc # Zadrži desni podinterval
    else:
        xb = xc # Zadrži lijevi podinterval

print('Rješenje bisekcije: %f' % xc)
print('Broj iteracija metode bisekcije: %d' % it)

plt.show()
```

Konvergencija metode podrazumijeva brzinu i način kojim se metoda kroz iteracije približava točnom rješenju. Kod metode bisekcije, aproksimacija korijena funkcije je točka dijeljenja x_c pa za apsolutnu grešku ε za točku x_c u prvom koraku vrijedi da ne

može biti veća od polovice intervala ograđivanja:

$$\varepsilon^{(1)} = |x^* - x_c| \leq \frac{x_b - x_a}{2}, \tag{4.14}$$

pa za poznati broj koraka n apsolutna greška iznosi:

$$\varepsilon^{(n)} = \left|x^* - x_c^{(n)}\right| \leq \frac{x_b - x_a}{2^n}. \tag{4.15}$$

Pošto se promatrani interval prepolavlja u svakom koraku, isto vrijedi i za grešku ε:

$$\varepsilon^{(i+1)} = \frac{1}{2} \cdot \varepsilon^{(i)}. \tag{4.16}$$

Greška linearno ovisi o grešci prethodnog koraka, pa kažemo da metoda bisekcije ima **linearnu konvergenciju**.

> **Napomena**
> Red konvergencije iterativne metode određuje se kao konstantni broj p koji zadovoljava
>
> $$\lim_{n\to\infty} \frac{\varepsilon^{(n+1)}}{\left(\varepsilon^{(n)}\right)^p} = C \tag{4.17}$$
>
> gdje je C konstanta za koju vrijedi $C < 1$. Kod metode bisekcije $C = \frac{1}{2}$ i $p = 1$ za svaki n, bez obzira na limes $n \to \infty$.

Ako je dopuštena pogreška unaprijed zadana, moguće je odrediti broj koraka metode bisekcije. Zahtijevamo li tako preciznost rješenja ε_{max}, vrijedi:

$$|x^* - x_{c_n}| < \varepsilon_{max}, \tag{4.18}$$

što, izraženo pomoću granica početnog intervala i broja koraka, glasi:

$$\frac{x_b - x_a}{2^n} < \varepsilon_{max}, \tag{4.19}$$

iz čega se lako dobije uvjet za broj koraka potrebnih za postizanje zahtijevane preciznosti:

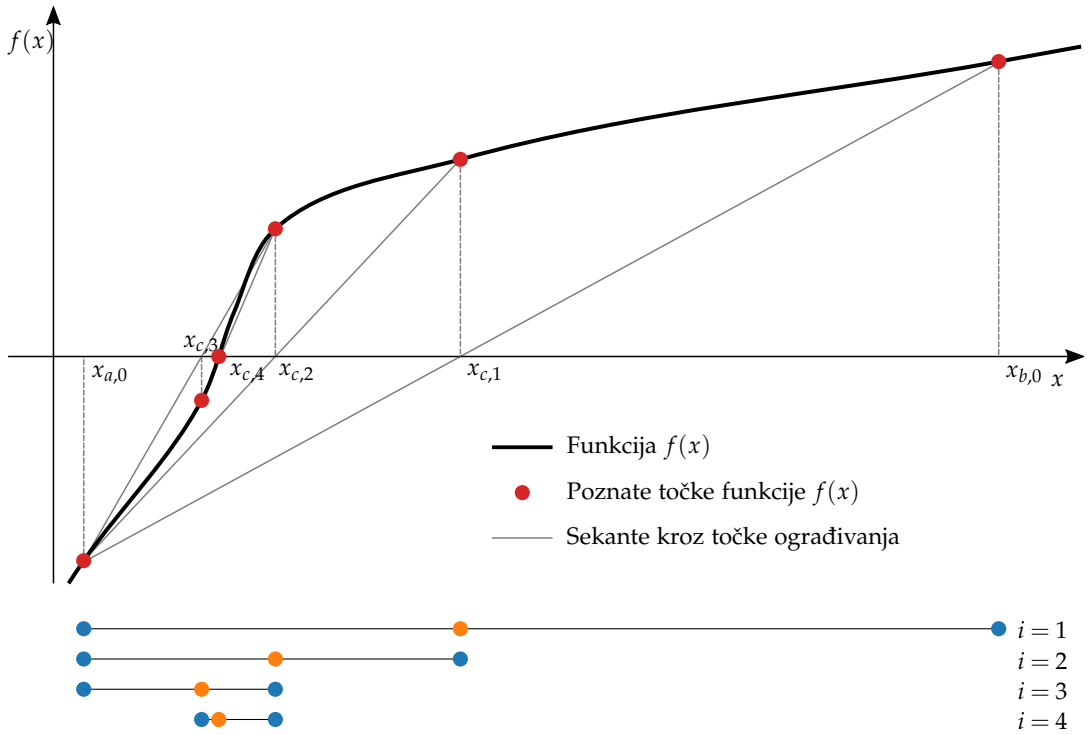
$$n > \frac{\log(x_b - x_a) - \log\varepsilon_{max}}{\log 2}. \tag{4.20}$$

4.1.2 Metoda regula-falsi

Promatramo li metodu bisekcije, možemo si postaviti pitanje da li je moguće interval dijeliti pametnije nego raspolavljanjem. Jedno rješenje je da interval podijelimo sekantom odnosno, drugim riječima, da funkciju aproksimiramo pravcem (kroz dvije granične točke x_a i x_b) i nul-točku tog pravca uzmemo kao točku razdiobe promatranog intervala na lijevi i desni podinterval (Slika 4.3).

Kao i kod metode bisekcije, za metodu regula-falsi podrazumijeva se da početni interval zadovoljava uvjet (4.10). Dijeljenje intervala $[x_a, x_b]$ pomoću nul-točke sekante x_c može se dobiti na temelju sličnosti trokuta:

$$\frac{x_b - x_c}{f(x_b)} = \frac{x_c - x_a}{-f(x_a)}, \tag{4.21}$$



Slika 4.3 Određivanje točke razdiobe intervala $[x_{a,i}, x_{b,i}]$ u metodi regula-falsi

iz čega slijedi:

$$x_c = \frac{x_a \cdot f(x_b) - x_b \cdot f(x_a)}{f(x_b) - f(x_a)}. \tag{4.22}$$

Valja uočiti da se gornji izraz može zapisati i kao

$$x_c = x_a - f(x_a) \cdot \frac{x_b - x_a}{f(x_b) - f(x_a)} \tag{4.23}$$

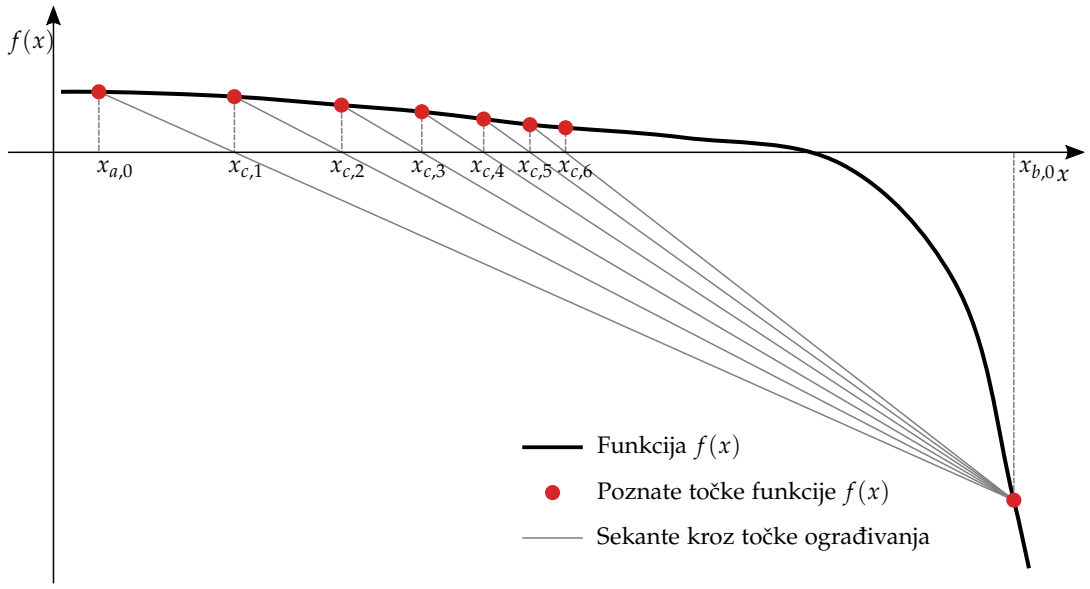
gdje izraz $(f(x_b) - f(x_a))/(x_b - x_a)$ odgovara koeficijentu nagiba sekante položene kroz točke $(x_a, f(x_a))$ i $(x_b, f(x_b))$.

Napomena
Intuitivno se može tumačiti da se regula-falsi metoda temelji na pretpostavci da je nul-točka funkcije bliža točki x_a (nego točki x_b) ukoliko je i $f(x_a)$ bliži nuli nego $f(x_b)$, i obrnuto.
Regula-falsi, kao i bisekcija, spada u metode ograđivanja (eng. *bracketing methods*).

Dalje se postupak nastavlja isto kao kod metode bisekcije (poglavlje 4.1.1), određivanjem novog suženog intervala, tj.:

- ako je $f(x_a) \cdot f(x_c) < 0$ novi interval je $[x_a, x_c]$,
- u suprotnom (tj. ako je $f(x_b) \cdot f(x_c) < 0$), novi interval je $[x_c, x_b]$.

Metoda regula-falsi bolja je od metode bisekcije, pa ima barem linearnu, a često i **superlinearnu konvergenciju**. Ipak, ona nailazi na poteškoće u traženju nul-točke nekih funkcija kada ne vrijedi pretpostavka da je rubna točka bliža nul-točki ako je



Slika 4.4 Primjer funkcije koja ima nepovoljan oblik kod kojeg dolazi do jednostrane konvergencije metode regula-falsi

njena funkcijska vrijednost bliža nuli (Slika 4.4). U takvim slučajevima metoda također sigurno konvergira, ali sporije od metode bisekcije.

Zbog ovog problema osmišljena je i modificirana regula-falsi metoda, u kojoj se dodatno vrši korekcija ako se desi da je neka granica intervala (bilo x_a ili x_b) ostala nepromijenjena u dvije iteracije. U takvom postupku modificirana točka dijeljenja intervala x_c' izračunava se po izrazu:

$$x_c' = \begin{cases} \dfrac{x_a \cdot f(x_b) - 2 \cdot x_b \cdot f(x_a)}{f(x_b) - 2 \cdot f(x_a)} \\ \dfrac{2 \cdot x_a \cdot f(x_b) - x_b \cdot f(x_a)}{2 \cdot f(x_b) - f(x_a)} \end{cases} . \tag{4.24}$$

4.1.3 Newton-Raphsonova metoda

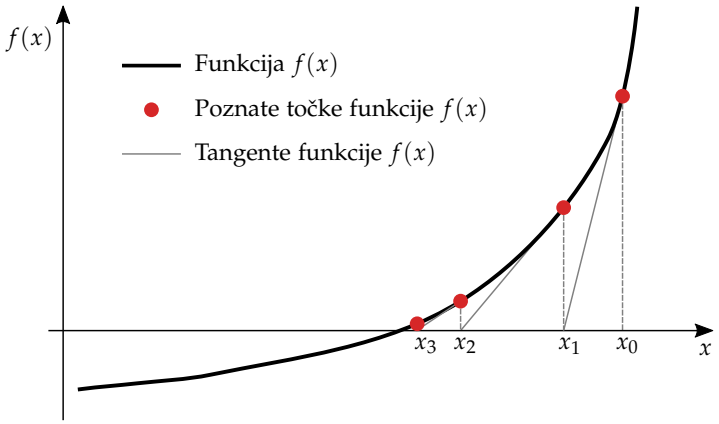
Mnogi matematički postupci temelje se na ideji da se problem koji se ne može riješiti zamijeni ili aproksimira nekim jednostavnijim problemom, za koji postoji rješenje. Ako tražimo nul-točku (tzv. "korijen") funkcije, općenito za proizvoljno tešku funkciju nemamo načina da dođemo do rješenja, no možemo probati funkciju zamijeniti nekom jednostavnijom funkcijom za koju znamo doći do rješenja.

Kao dobra aproksimacija funkcije (u okolici neke točke) često se uzima tangenta, što bi moglo biti korisno u traženju nul-točke, jer je tangenta obični pravac, kojemu je lako naći nul-točku. S druge strane, otežavajuća okolnost u ovom pristupu krije se u tome što nam za određivanje tangente mora biti poznata derivacija funkcije, što predstavlja veliki zahtjev po pitanju poznavanja funkcije čiju nul-točku tražimo.

Dakle, ako osim same funkcije $f(x)$ znamo u svakoj točki izračunati i derivaciju funkcije $f'(x)$, možemo te informacije iskoristiti za traženje nul-točke temeljem gore iznesene ideje. Takav postupak zove se Newton-Raphsonova metoda (ili kraće Newtonova metoda, ili metoda tangente).

Postupak je iterativan i počinje od početne točke x_0 (tj. pretpostavljenog rješenja). Na temelju početne točke izračunavamo drugu točku odnosno na temelju tekuće

izračunavamo narednu, do konvergencije (Slika 4.5).



Slika 4.5
Aproksimacija funkcije tangentom u Newton-Raphsonovoj metodi

Razvojem funkcije u Taylorov red u okolini točke x_0, funkciju $f(x)$ možemo aproksimirati sa

$$f(x) \approx f(x_0) + (x - x_0) \cdot f'(x_0) + (x - x_0) \cdot f''(x_0) + \ldots \tag{4.25}$$

Korištenjem samo prva dva člana funkciju $f(x)$ aproksimiramo sa tangentom kao:

$$f(x_1) \approx f(x_0) + (x_1 - x_0) \cdot f'(x_0) \tag{4.26}$$

gdje vrijedi $f(x_1) = 0$ pošto je traženi x_1 aproksimacija nul-točke funkcije $f(x)$, odnosno nul-točka tangente kojom smo aproksimirali funkciju $f(x)$.

Sada možemo generalizirati određivanje sljedeće točke u svakoj iteraciji i:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}. \tag{4.27}$$

Nova točka x_{i+1} je nul-točka tangente funkcije $f(x)$ u točki x_i, što i daje naziv metode: metoda tangente.

Do nul-točke ćemo doći tako da iteriramo po formuli (4.27) dok ne zadovoljimo neki uvjet zaustavljanja, npr. uvjet (4.13) ili pak neki uvjet konvergencije samog iterativnog postupka, npr.:

$$\left| \frac{x_i - x_{i-1}}{x_i} \right| < \varepsilon. \tag{4.28}$$

Programski kod 4.2 prikazuje implementaciju Newton-Raphsonove metode u Pythonu za traženje nul-točke funkcije $f(x) = 5x^2 + 0.5\sin(10 - x) - 10$.

Python kod 4.2 Newton-Raphsonova metoda

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

def f(x):
    return 5.0 * x**2.0 + 0.5 * np.sin(10.0 * x) - 10.0

def fder(x):
    return 10.0 * x + 5.0 * np.cos(10.0 * x)

# x1 = 0.3 # Nestabilnost zbog kvazi-horizontalnih tangenti!
```

```
x1 = float(input('Unesi blisku tocku x1 iz intervala [0, 3]: '))
eps = float(input('Unesi apsolutnu pogresku: '))

maxit = 100
it = 1
X = np.array([x1])

while it < maxit:
    it += 1
    Xi = X[-1] - f(X[-1]) / fder(X[-1])
    X = np.append(X, Xi)

    # crtanje tangenti
    plt.annotate(str(it-1), xy=(X[-2], f(X[-2])))
    plt.plot([X[-2], X[-1]], [f(X[-2]), 0], c='gray', lw=0.5)
    plt.plot([X[-1], X[-1]], [0, f(X[-1])], c='gray', lw=0.5)

    err = np.abs(X[-1] - X[-2])
    if err < eps:
        print('Nul-tocka je:', X[-1])
        break

xplot = np.linspace(0, 3, 100)
plt.plot(xplot, f(xplot))
plt.axhline(0, c='r')

plt.figure()
plt.title('Konvergencija metode')
plt.plot(np.arange(1, it+1), X)
plt.xlabel('Iteracije, $i$')
plt.ylabel('$x_i$')
plt.show()
```

Kakva je konvergencija Newton-Raphsonove metode? Pretpostavimo li da je funkcija $f(x)$ dva puta derivabilna, u okolini x_i vrijedi Taylorova formula:

$$f(x) = f(x_i) + f'(x_i)(x - x_i) + \frac{1}{2}f''(\xi)(x - x_i)^2, \tag{4.29}$$

uz $\xi \in [x_i, x]$. Konkretno za traženi korijen (tj. nul-točku) x^* vrijedi:

$$0 = f(x_i) + f'(x_i)(x^* - x_i) + \frac{1}{2}f''(\xi)(x^* - x_i)^2. \tag{4.30}$$

Iz Newton-Raphsonove formule znamo da je

$$0 = f(x_i) + f'(x_i)(x_{i+1} - x_i), \tag{4.31}$$

pa oduzimanjem dviju gornjih jednakosti dobivamo:

$$0 = f'(x_i)(x^* - x_{i+1}) + \frac{1}{2}f''(\xi)(x^* - x_i)^2. \tag{4.32}$$

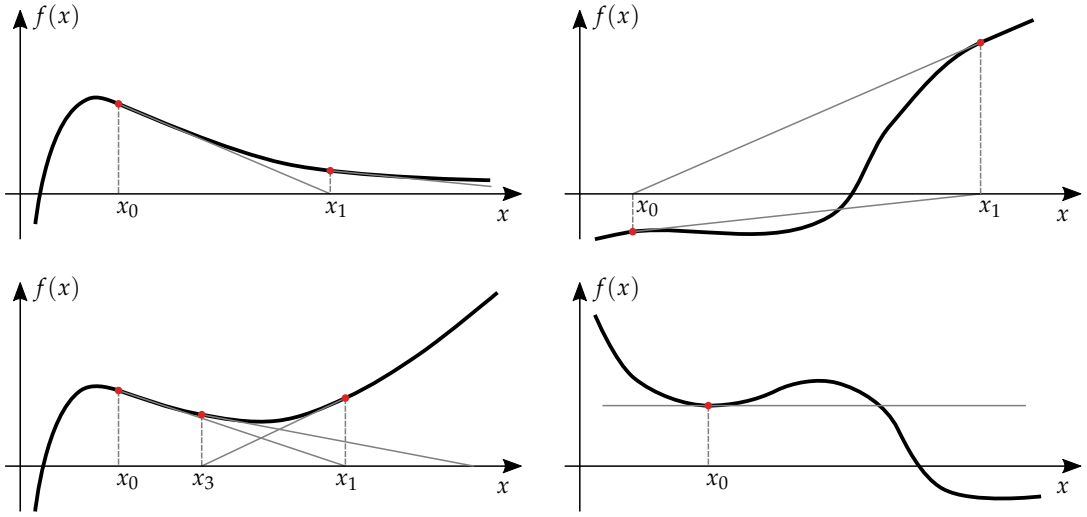
Kako su izrazi $(x^* - x_i)$ i $(x^* - x_{i+1})$ zapravo greške u koraku i i $i+1$, redom, slijedi da je

$$\varepsilon^{(i+1)} = -\frac{f''(\xi)}{2 \cdot f'(x_i)} \cdot (\varepsilon^{(i)})^2, \tag{4.33}$$

što znači da metoda konvergira **kvadratično**. Kada smo dovoljno blizu korijenu, kvadratična konvergencija metode znači da se broj točnih znamenki udvostručava sa svakim korakom.

Upravo radi vrlo brze konvergencije, Newton-Raphsonova metoda i jest najkorištenija od svih metoda za traženje korijena (u slučajevima kada znamo derivaciju funkcije).

S druge strane, metoda ima slabu globalnu konvergenciju, što znači da ako nismo dovoljno blizu nul-točke metoda će često divergirati. Naime, osim kod vrlo jednostavnih funkcija, što smo dalje od nul-točke veća je vjerojatnost da ćemo naići na lokalne minimume i maksimume, tj. na mjesta gdje su tangente horizontalne i kvazi-horizontalne (Slika 4.6).



Slika 4.6 Primjeri divergencije Newton-Raphsonove metode

Problem sa (kvazi-) horizontalnim tangentama može se uočiti i na samoj formuli (4.27) po kojoj se metoda provodi. Naime, lako je primijetiti da za jako malu vrijednost $f'(x_i)$ razlomak na desnoj strani izraza "eksplodira", dok je za slučaj $f'(x_i) = 0$ cijela formula doslovno "neizračunljiva" pošto dijeljenje sa nulom nije definirano.

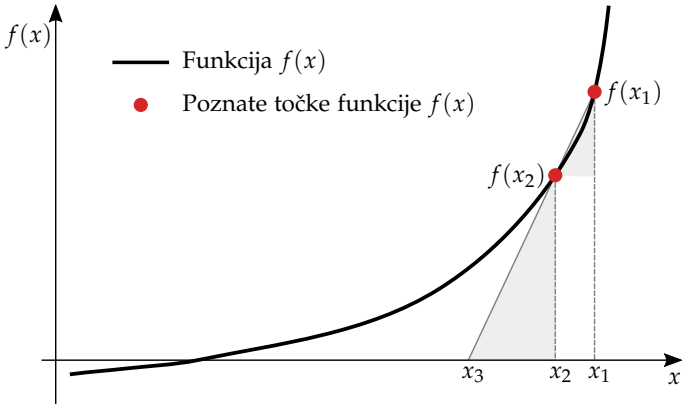
4.1.4 Metoda sekante

S obzirom da je dosta "skupo" zahtijevati poznavanje derivacije funkcije, postavlja se pitanje možemo li osmisliti metodu koja ne zahtijeva poznavanje derivacije, a da ima slična svojstva kao Newton-Raphsonova metoda. U tom smislu, ako je tangenta dobra aproksimacija funkcije, možemo primijetiti da je sekanta dobra aproksimacija tangente, posebno ako su dvije točke kroz koju provlačimo sekantu vrlo blizu jedna drugoj.

Pretpostavimo da su nam poznate dvije točke x_1 i x_2 koje su blizu traženog korijena, tada možemo funkciju f zamijeniti pravcem kroz točke $(x_1, f(x_1))$ i $(x_2, f(x_2))$ tj. sekantom. Iz sličnosti dvaju trokuta označenih na Slici 4.7 slijedi:

$$\frac{f(x_1) - f(x_2)}{x_1 - x_2} = \frac{f(x_2)}{x_2 - x_3}. \tag{4.34}$$

Iz gornje jednadžbe možemo izraziti generaliziranu formulu po kojoj možemo na



Slika 4.7
Sekanta koja prolazi kroz dvije točke: $(x_1, f(x_1))$ i $(x_2, f(x_2))$.

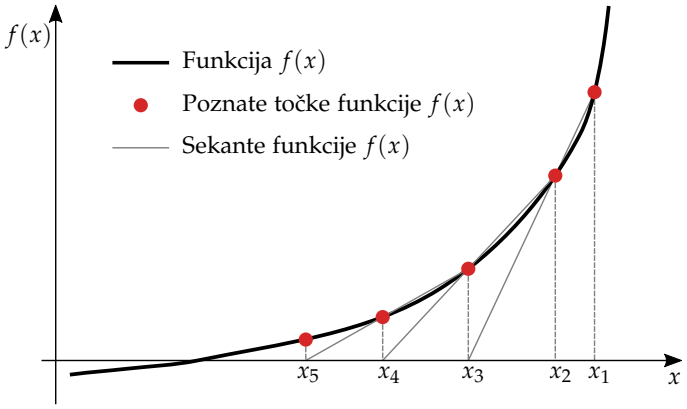
temelju zadnje (i) i predzadnje $(i-1)$ točke izračunati sljedeću $(i+1)$ točku:

$$x_{i+1} = x_i - \frac{f(x_i) \cdot (x_{i-1} - x_i)}{f(x_{i-1}) - f(x_i)}. \tag{4.35}$$

Dobivena točka je ništa drugo do nul-točka sekante koja prolazi kroz zadnju točku $(x_i, f(x_i))$ i predzadnju točku $(x_{i-1}, f(x_{i-1}))$. Iterativnim računanjem po formuli (4.35) približit ćemo se nul-točki funkcije f na željenu točnost (Slika 4.8).

Usporedimo li formulu metode sekante (4.35) sa Newton-Raphsonovom formulom (4.27), možemo uočiti da je to zapravo jedna te ista formula, samo što se kod metode sekante ne koristi prava derivacija $f'(x)$ nego približno izračunata derivacija:

$$f'(x_i) \approx \frac{f(x_{i-1}) - f(x_i)}{x_{i-1} - x_i}. \tag{4.36}$$



Slika 4.8
Približavanje nul-točki funkcije metodom sekante

Što se konvergencije tiče, može se pokazati da za metodu sekante vrijedi:

$$\varepsilon^{(i+1)} = c \cdot (\varepsilon^{(i)})^{1.618}, \tag{4.37}$$

gdje je 1.618 konstanta poznata kao *zlatni rez*. S obzirom da je ovaj eksponent veći od jedan, ovdje govorimo o superlinearnoj konvergenciji. To je sporija konvergencija od Newton-Raphsonove metode (cijena prelaska sa tangente na sekantu!), ali još uvijek bolja od metode bisekcije. Iako smo se zamjenom tangente sa sekantom riješili zahtjeva poznavanja derivacije funkcije, još uvijek je prisutan problem kvazi-horizontalnih tangenti, točnije kvazi-horizontalnih sekanti.

4.1.5 Usporedba metoda

Opisane metode za rješavanje nelinearne jednadžbe, tj. traženje nul-točke funkcije, mogu se međusobno usporediti po pitanju konvergencije (točnosti) i drugih osobina, kako je to rezimirano u Tablici 4.1.

Tablica 4.1 Usporedba metoda za traženje nul-točke funkcije

Metoda	Konvergencija	Prednosti	Nedostaci
Bisekcija	1	Robusnost	Spora konvergencija
Regula-falsi	>1	Robusnost	Jednostrana konvergencija
Newton-Raphson	2	Brza konvergencija	Zahtjeva poznavanje derivacije, problem horizontalne tangente
Sekanta	1.618	Brza konvergencija	Problem horizontalne sekante

U praksi bi prvi izbor trebala biti Newton-Raphsonova metoda, pod uvjetom da znamo izračunati derivaciju u svakoj točki funkcije. Ako ne znamo derivaciju, onda je prva alternativa metoda sekante. U slučaju da je funkcija preteška odnosno konvergencija prespora uslijed neugodnog izgleda funkcije, treba pokušati sa metodom regula-falsi, koja je hibrid metode ograđivanja i metode sekante, i nema problem sa horizontalnim sekantama. Kao zadnji izbor preostaje metoda bisekcije, koja se nerijetko pokaže i kao najbolji izbor, upravo zbog svoje izražene robusnosti.

4.1.6 Rješavanje nelinearne jednadžbe pomoću SciPy modula

SciPy modul nudi nekoliko metoda za rješavanje nelinarnih jednadžbi tj. za traženje nul-točki funkcija koje se nalaze u podmodulu `scipy.optimize`. Ovdje su navedene samo neke od tih metoda i to u svom osnovnom obliku, bez objašnjavanja dodatnih argumenata.

Metoda bisekcije je implementirana u funkciji bisect

```
scipy.optimize.bisect(f, a, b)
```

koja prima funkciju `f` čiju nul-točku tražimo te početne granice intervala ograđivanja `a` i `b`.

Funkcija `newton` u SciPy-u, ovisno o danim argumentima, omogućuje pozivanje dviju metoda za traženje nul-točke funkcije. Ukoliko se koriste samo nužni argumenti (funkcija `f` i približno/početno rješenje `x0`) onda se koristi metoda sekante. Ako se definira i `fprime`, kao derivacija funkcije `f`, onda se koristi metoda tangente tj. Newton-Raphsonova metoda. Sintaksa funkcije `newton` je:

```
scipy.optimize.newton(f, x0, fprime=None)
```

U praksi se često koriste hibridne metode, kao što su MINPACK hybrd i hybrj algoritmi koji imaju određene prednosti u odnosu na "osnovne" metoda, a implementirani su u funkciji `fsolve`:

```
scipy.optimize.fsolve(f, x0)
```

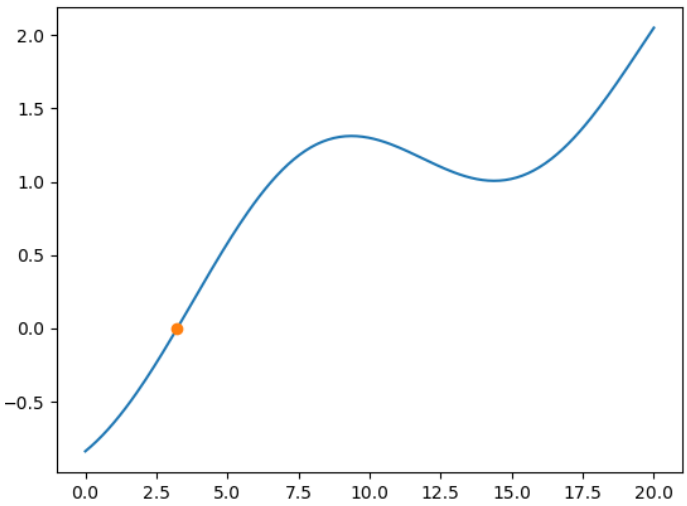
Argument `f` je funkcija čiji korijen tražimo a `x0` je početna točka tj. približno pretpostavljeno rješenje u čijoj okolini se vrši pretraga.

Programski kod 4.3 ilustrira korištenje funkcija iz SciPy modula za traženje nul-točke funkcije $f(x) = \sqrt{x+10} - \cos^2(x/5) - 3$.

Python kod 4.3 Traženje nul-točke pomoću SciPy funkcija

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as opt

# Funkcija
def f(x):
    return np.sqrt(x + 10) -  np.cos(x / 5) ** 2 - 3

# Derivacija funkcije
def df_dx(x):
    return 1 / (2 * np.sqrt(10 + x)) + 2 / 5 * np.cos(x / 5) * np.sin(x / 5)

# Metoda bisekcije
print('scipy.bisect')
x0_bisect = opt.bisect(f, 0, 20)
print(x0_bisect, f(x0_bisect))

# Metoda sekante
print('scipy.newton sekanta')
x0_newton = opt.newton(f, 7) # Ne konvergira za pocetni x=10
print(x0_newton, f(x0_newton))

# Newton-Raphsonova metoda
print('scipy.newton Newton-Raphsonova metoda')
x0_newton = opt.newton(f, 7, df_dx) # Sa derivacijom
print(x0_newton, f(x0_newton))

# fsolve
print('scipy.fsolve')
x0_fsolve = opt.fsolve(f, 7)
print(x0_fsolve[0], f(x0_fsolve[0]))

# Crtanje funkcije i nul-tocke
x = np.linspace(0, 20, 200)
plt.plot(x, f(x))
plt.plot(x0_fsolve, 0, 'o')
```

```
scipy.bisect
3.2308874320631276 2.0827783942e-13
scipy.newton sekanta
3.23088743206 0.0
scipy.newton Newton-Raphsonova metoda
3.23088743206 0.0
scipy.fsolve
3.23088743206 0.0
```



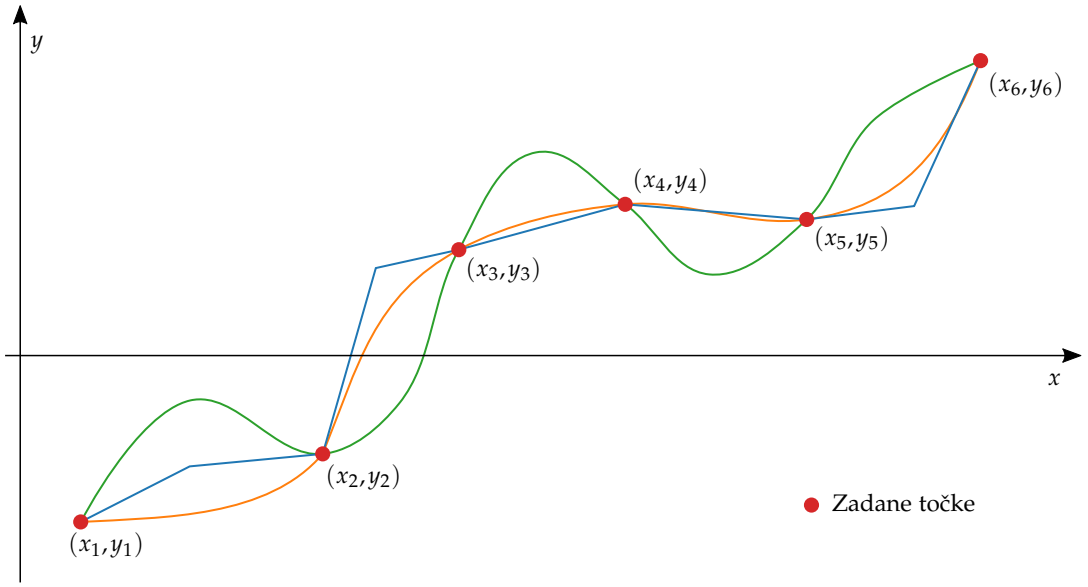
Slika 4.9
Traženje nul-točki pomoću metoda iz SciPy modula (Izvorni kod 4.3)

4.2 Interpolacija

Stefan Ivić
Siniša Družeta

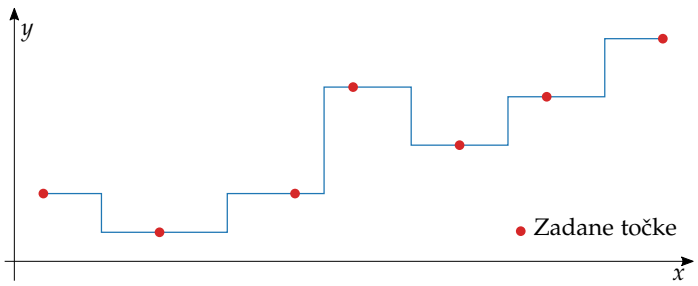
Uzmimo situaciju da imamo niz zadanih točaka (x_i, y_i) i moramo odrediti što se dešava sa vezom $y(x)$ u međuprostoru između zadanih točaka, tj za $x_i < x < x_{i+1}$. Ako su nam zadane (x_i, y_i) točke potpuno pouzdane (tj. potpuno točne, ili strogo zadane), onda se taj problem svodi na pronalazak krivulje koja prolazi kroz zadane točke i takav postupak zove se **interpolacija**.

Ima beskonačno načina kako možemo provesti interpolaciju, tj. povezati zadane točke u jednu krivulju (Slika 4.10).



Slika 4.10 Moguće interpolacijske krivulje na zadanim $x - y$ točkama

Jedno od najjednostavnijih rješenja bila bi "metoda najbližeg susjeda". Po ovoj metodi području u okolici pojedine točke $[x_i - \frac{x_{i-1}+x_i}{2}, x_i + \frac{x_i+x_{i+1}}{2}]$ jednostavno pridružujemo vrijednost y_i, kako je to i prikazano na Slici 4.11. Time dobijemo interpolacijsku funkciju koja je konstantna po dijelovima. Ovakav postupak zove se još i interpolacija nultog reda.



Slika 4.11
Interpolacija metodom najbližeg susjeda

4.2.1 Lagrangeova interpolacija

U praksi je puno češći slučaj da želimo imati glatke interpolacijske funkcije, tj. želimo povezati točke krivuljom koja nema skokove (diskontinuitete). Prvo rješenje koje bi nam moglo pasti na pamet bi bili polinomi. Znamo da se sa n poznatih točaka može definirati polinom stupnja $n-1$:

$$y(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \ldots + a_{n-1} \cdot x^{n-1}. \tag{4.38}$$

Pošto krivulja interpolacijskog polinoma prolazi kroz zadane točke, jednadžba (4.38) treba biti zadovoljena za svaku točku: $y(x_i) = y_i$. To vrijedi za svaku od n točaka, pa dobivamo sustav od n linearnih jednadžbi:

$$\begin{array}{ccccccccc} a_0 & + & a_1 \cdot x_1 & + & a_2 \cdot x_1^2 & + & \cdots a_{n-1} \cdot x_1^{n-1} & = & y_1 \\ a_0 & + & a_1 \cdot x_2 & + & a_2 \cdot x_2^2 & + & \cdots a_{n-1} \cdot x_2^{n-1} & = & y_2 \\ \vdots & & & & \vdots & & & & \\ a_0 & + & a_1 \cdot x_n & + & a_2 \cdot x_n^2 & + & \cdots a_{n-1} \cdot x_n^{n-1} & = & y_n \end{array} \tag{4.39}$$

gdje su $a_0, \ldots, a_{n-1}$ nepoznanice. Uvjet interpolacije polinoma $n-1$ stupnja se pretvara u ovih n jednadžbi, a interpolacijski polinom je jedinstven ukoliko je linearni sustav (4.39) rješiv tj. ukoliko je determinanta sustava različita od nule.

> **Napomena**
> Jedinstveni interpolacijski polinom možemo interpretirati na primjeru polinoma 2. stupnja (parabola). Kroz 3 točke prolazi samo jedan jedini polinom 2. stupnja. Doduše, ukoliko se dvije zadane točke preklapaju, determinanta sustava (4.39) jednaka je nuli i sustav nije jednoznačno rješiv. Drugim riječima, preklapanjem dviju točaka efektivno imamo samo jednu točku, pa slijedi da se kroz takve dvije točke može provući beskonačno mnogo polinoma 2. stupnja.

Koeficijenti sustava (4.39) poprimaju vrijednosti x_i^k (za $k = 0, 1, \ldots, n-1$), što kod većih vrijednosti k (tj. za polinome visokog stupnja) može rezultirati izrazito malim vrijednostima za $x_i < 1$, odnosno izrazito velikim vrijednostima za $x_i > 1$. Takve velike razlike u koeficijentima linearnog sustava uvelike otežavaju rješavanje sustava, što je preciznije objašnjeno u poglavlju 4.5. Zbog toga bi bilo poželjno da možemo ovakav polinom odrediti bez rješavanja linearnog sustava.

Pretpostavimo da traženi polinom $P_n(x)$, koji prolazi kroz n zadanih točaka, možemo dobiti kao zbroj n jednostavnijih polinoma:

$$P_n(x) = \sum_{i=1}^{n} (L_i(x) \cdot y_i) \tag{4.40}$$

gdje je $L_i(x)$ pojedini (i-ti) polinom, koji pak mora zadovoljavati sljedeće uvjete:

- $L_i(x_i) = 1$ (mora poprimiti vrijednost jedan kada prolazi točno kroz pripadnu zadanu i-tu točku x_i),
- $L_i(x_j) = 0$, $\forall i \neq j$ (mora poprimiti vrijednost nula kada prolazi kroz sve ostale točke x_j),

kako bi vrijedilo $P_n(x_i) = y_i$.

Takva svojstva ima polinom:

$$L_i(x) = \prod_{j=1, j\neq i}^{n} \frac{x - x_j}{x_i - x_j} = \frac{(x - x_1)}{(x_i - x_1)} \cdots \frac{(x - x_{i-1})}{(x_i - x_{i-1})} \cdot \frac{(x - x_{i+1})}{(x_i - x_{i+1})} \cdots \frac{(x - x_n)}{(x_i - x_n)}, \quad (4.41)$$

jer u slučaju za $x = x_j$ vrijedi:

$$\frac{x - x_j}{x_i - x_j} = \frac{x_j - x_j}{x_i - x_j} = 0, \quad (4.42)$$

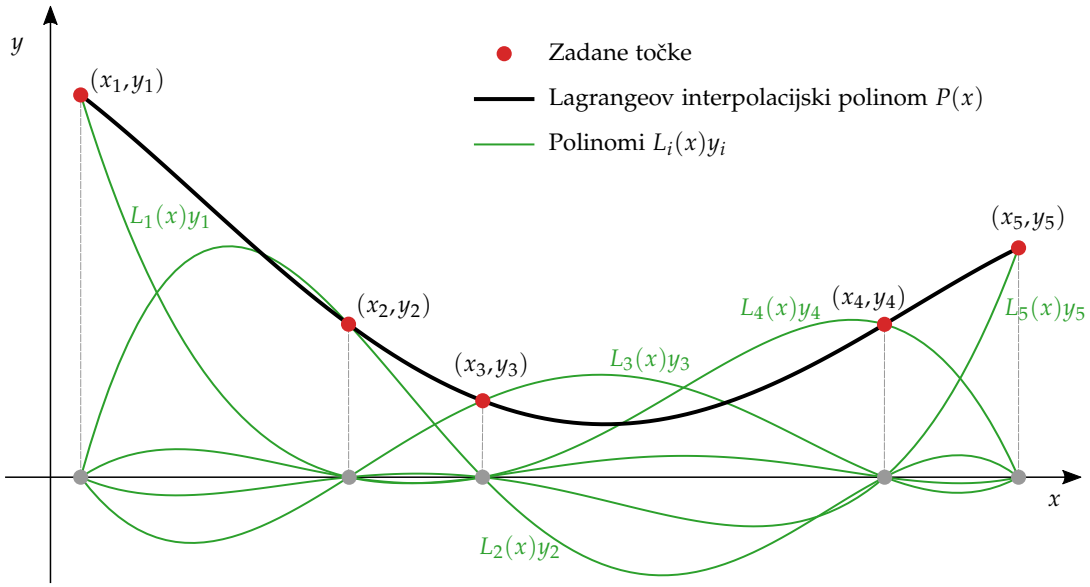
a u slučaju kad je $x = x_i$ imamo:

$$\frac{x - x_j}{x_i - x_j} = \frac{x_i - x_j}{x_i - x_j} = 1. \quad (4.43)$$

Sumiranjem $L_i(x) \cdot y_i$ za sve točke dobijemo Lagrangeov interpolacijski polinom $P_n(x)$ (izraz (4.40)), gdje je L_i tzv. Lagrangeov bazni polinom:

$$L_i(x) = \prod_{j=1, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}. \quad (4.44)$$

Da rezimiramo, Lagrangeov interpolacijski polinom dobije se zbrajanjem polinoma, od kojih svaki u pripadnoj mu točki x_i poprima pripadnu vrijednost y_i, dok u svim ostalim točkama x_j poprima vrijednost nula (Slika 4.12).



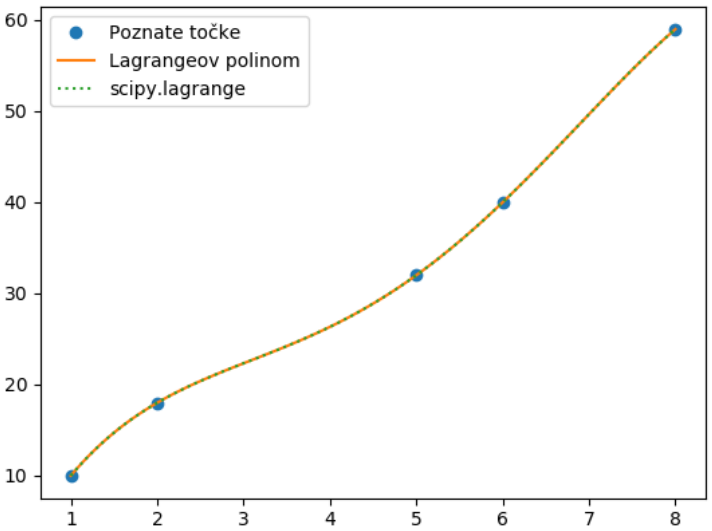
Slika 4.12 Konstrukcija Lagrangeovog interpolacijskog polinoma

Programski kod 4.4 prikazuje implementaciju Lagrangeove interpolacije u Pythonu.

Python kod 4.4 Lagrangeova interpolacija

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.interpolate as intrp

XY = np.loadtxt('tocke.txt')
```

```
n = np.shape(XY)[0]
X = XY[:, 0]
Y = XY[:, 1]

plt.plot(X, Y, 'o', label = 'Poznate točke')

Xint = np.linspace(np.min(X), np.max(X), 100)
Yint = np.zeros_like(Xint)
for ix, x in enumerate(Xint):
    P = 0
    for i in range(n):
        pi = 1.0
        for j in range(n):
            if j != i:
                pi = pi * (x - X[j]) / (X[i] - X[j])
        P = P + Y[i] * pi
    Yint[ix] = P
plt.plot(Xint, Yint, ls='-', label = 'Lagrangeov polinom')

# SciPy Lagrange
lanf = intrp.lagrange(X, Y)
print(lanf)
plt.plot(Xint, lanf(Xint), ls=':', label='scipy.lagrange')

plt.legend()
plt.show()
```

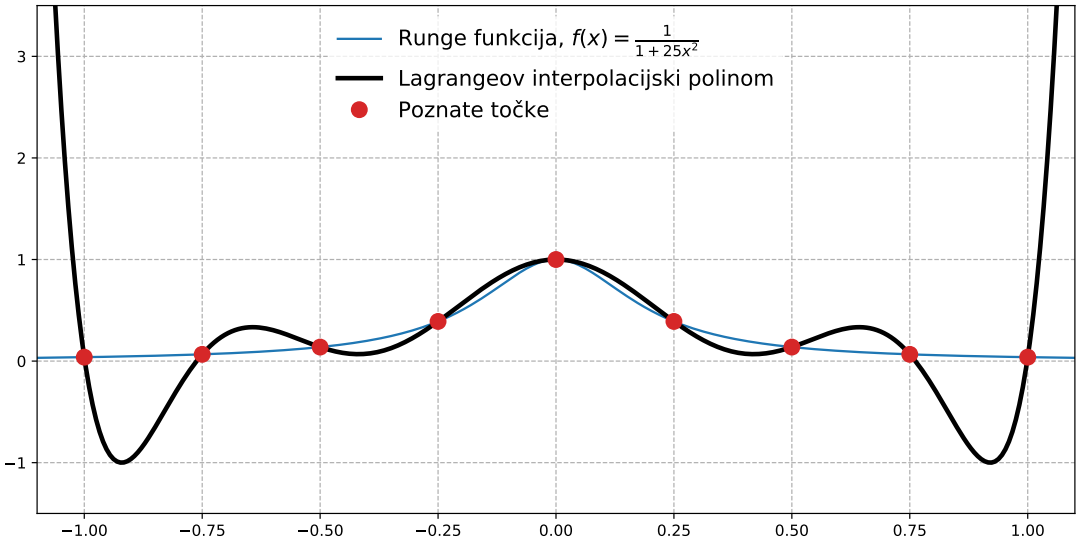


Slika 4.13
Primjer Lagrangeovog interpolacijskog polinoma

Ako znamo da Lagrangeov polinom ima tim viši stupanj što je veći broj točaka na kojima se provodi interpolacija i ako znamo da polinomi višeg stupnja imaju veći broj ekstrema (minimuma i maksimuma) onda je očito da Langrangeov polinom mora jako oscilirati. U tom smislu sa povećanjem stupnja polinoma ne dobivamo bolju interpolaciju, što je suprotno od onog što obično očekujemo. Kod interpolacije na većem broju točaka, a ponekad i na manjem broju ekvidistantnih točaka (Slika 4.14) oscilacije mogu biti vrlo veliki problem. Oscilacije su obično posebno izražene na rubovima, zbog čega se Lagrangeova interpolacija u principu uopće ne može koristiti za ekstrapolaciju.

Osim toga, svi Lagrangeovi bazni polinomi moraju se iznova računati kod svake promjene točaka ili dodavanja nove točke, što čini ovu metodu relativno skupom za

interpolaciju u slučajevima kada se točke mijenjaju u vremenu.



Slika 4.14 Oscilacije Lagrangeovog polinoma na malom broju ekvidistantnih točaka

4.2.2 Splajn interpolacija

S obzirom da interpolacija polinomima visokog stupnja uzrokuje oscilacije, može se za interpolaciju koristiti polinome niskog stupnja tako da ih se provlači kroz manje skupine točaka, pa se onda te polinome može pospajati u jedinstvenu krivulju. Drugim riječima, umjesto da radimo interpolaciju na svim točkama odjednom (Lagrangeova interpolacija), možemo raditi interpolaciju po dijelovima, ali tako da se sve te dobivene interpolacijske krivulje lijepo nadovezuju jedna na drugu. Takav postupak zove se splajn interpolacija (eng. *spline*).

Linearni splajn

Najjednostavnija vrsta splajn interpolacije je **linearni splajn** gdje se područje između dviju susjednih točaka aproksimira polinomom prvog stupnja tj. pravcem (Slika 4.15). Ukoliko je zadano $n+1$ točaka, potrebno je pronaći n pravaca koji spajaju po dvije susjedne točke (koje čine jedan segment). Tako jednadžba pravca za segment od točke i do točke $i+1$ (za $x \in [x_i, x_{i+1}]$) glasi:

$$f_i(x) = a_i + b_i \cdot x \tag{4.45}$$

za $i = 1, \ldots, n$. Za pronaći splajn potrebno je odrediti sve nepoznanice a_i, b_i ($2n$ nepoznanica), pri čemu pravac treba prolaziti kroz obje točke podintervala tj. mora vrijediti:

$$\begin{aligned} f_i(x_i) &= y_i, \qquad \text{za } i = 1, \ldots, n \\ f_i(x_{i+1}) &= y_{i+1}, \qquad \text{za } i = 1, \ldots, n \end{aligned} \tag{4.46}$$

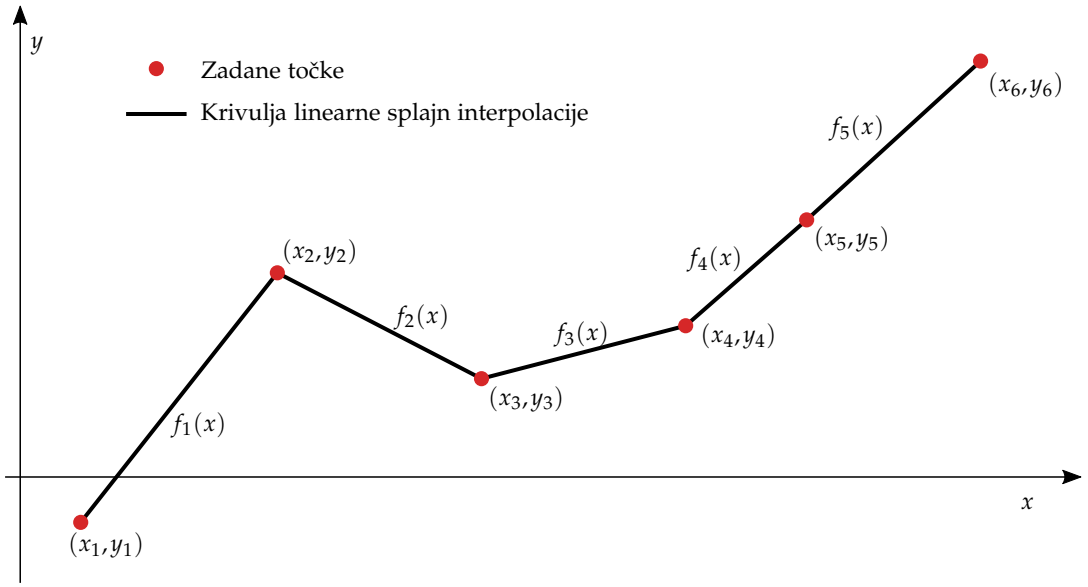
Uvrštavanjem jednadžbe pravca (4.45) u uvjete (4.46) dobije se sustav linearnih jednadžbi gdje su parametri pravaca a_i i b_i nepoznanice:

$$\begin{aligned} a_i + b_i x_i &= y_i, \qquad \text{za } i = 1, \ldots, n \\ a_i + b_i x_{i+1} &= y_{i+1}, \qquad \text{za } i = 1, \ldots, n \end{aligned} \tag{4.47}$$

Iako sustav jednadžbi (4.47) ima $2n$ jednadžbi i $2n$ nepoznanica, jednadžbe za svaki par a_i i b_i su nezavisne o ostalim jednadžbama, pa možemo eksplicitno definirati rješenje kao:

$$b_i = \frac{y_i - y_{i+1}}{x_i - x_{i+1}}, \qquad a_i = y_{i+1} - b_i x_{i+1}, \qquad \text{za } i = 1,\ldots,n \tag{4.48}$$

čime su definirane jednadžbe interpolacijskih pravaca za svaki i-ti segment omeđen točkama (x_i, y_i) i (x_{i+1}, y_{i+1}).



Slika 4.15 Skica interpolacije pomoću linearnog splajna

Kvadratni splajn

Treba primijetiti da je s linearnim splajnom osigurana samo neprekidnost interpolacijske funkcije, ali ne i njenih derivacija. Ukoliko želimo glatku interpolacijsku funkciju tj. glatki splajn, tada treba uvjetovati neprekidnost prve derivacije. Pošto za ispunjenje dodatnih uvjete treba više koeficijenata interpolacijskih funkcija po segmentima, mora se povisiti stupanj polinoma. Ako se za splajn interpolaciju koristi polinom 2. stupnja, onda govorimo o **kvadratnom splajnu**. U tom slučaju, između dvije susjedne poznate točke, koristimo kvadratni polinom

$$f_i(x) = a_i + b_i x + c_i x^2, \quad \text{za } i = 1,\ldots,n \tag{4.49}$$

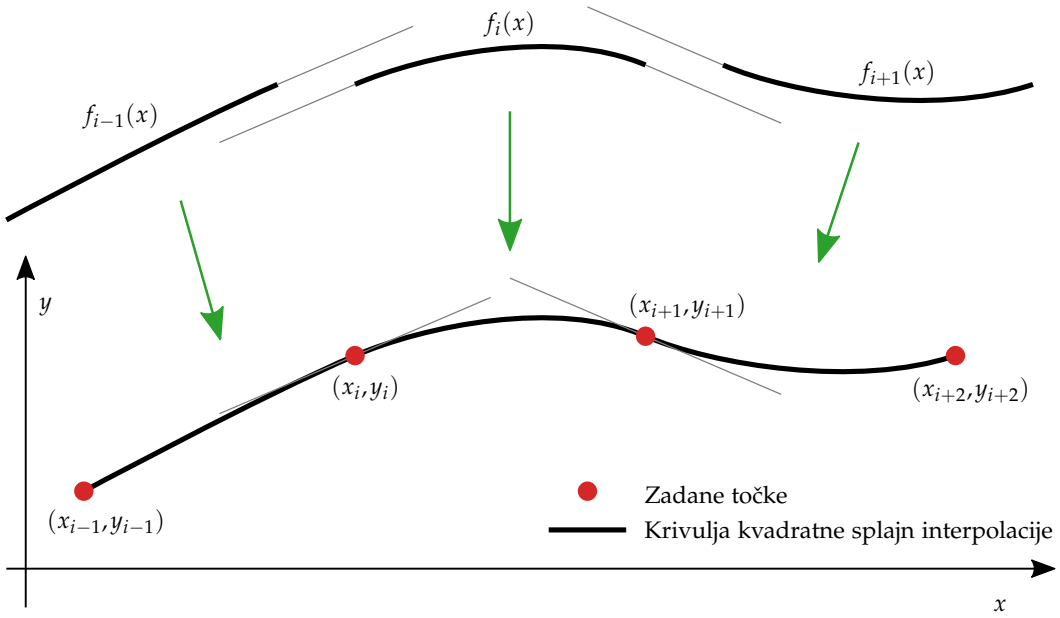
na svakom i-tom od ukupno n segmenata (Slika 4.16). Sada je potrebno pronaći $3n$ nepoznanica: a_i, b_i, c_i za $i = 1,\ldots,n$. Interpolacijski polinomi moraju prolaziti kroz rubne točke podintervala, kao i kod linearnog splajna, ali dodatno se zahtjeva da susjedni polinomi u točki dodira imaju istu prvu derivaciju. Na taj način postiže se glatkoća cjelokupne interpolacijske krivulje.

Navedne uvjete možemo definirati kao:

$$\begin{aligned} f_i(x_i) &= y_i, && \text{za } i = 1,\ldots,n \\ f_i(x_{i+1}) &= y_{i+1}, && \text{za } i = 1,\ldots,n \\ f_i'(x_{i+1}) &= f_{i+1}'(x_{i+1}), && \text{za } i = 1,\ldots,n-1 \end{aligned} \tag{4.50}$$

što čini sustav od $3n-1$ linearnih jednadžbi. Pošto je ukupno $3n$ nepoznatih koeficijenata interpolacijskih polinoma, nedostaje još jedan uvjet tj. jednadžba da bi sustav bio potpuno definiran. Za to je uobičajeno koristiti uvjet u prvoj točki $f_1''(x_0) = 0$.

Napomena
Ovisno o primjeni, dodatni uvjet koji je potrebno postaviti kod konstrukcije kvadratne splajn interpolacije može biti prilagođen zahtjevima ili potrebama primjene same interpolacije. Tako se, umjesto "standardnog uvjeta" $f_1''(x_0) = 0$, može propisati derivacija u prvoj ili zadnjoj točki – što se praktičnije može interpretirati kao nagib tangente u prvoj ili zadnjoj točki interpolacijske krivulje.



Slika 4.16 Skica konstrukcije kvadratnog splajna

Sustav jednadžbi koji proizlazi iz uvjeta (4.50), za razliku od onog kod linearnog splajna, nema eksplicitnih rješenja za svaki segment neovisno, pa se njegovom rješavanju pristupa pomoću neke od metoda opisanih u poglavlju 4.5. Sustav linearnih jednadžbi (4.50) može se zapisati kao:

$$\begin{aligned} a_i + b_i x_i + c_i x_i^2 &= y_i, && \text{za } i = 1,\ldots,n \\ a_i + b_i x_{i+1} + c_i x_{i+1}^2 &= y_{i+1}, && \text{za } i = 1,\ldots,n \\ b_i + 2c_i x_{i+1} &= b_{i+1} + 2c_{i+1} x_{i+1}, && \text{za } i = 1,\ldots,n-1 \end{aligned} \tag{4.51}$$

gdje su prva dva izraza uvjet prolaska kroz rubne točke segmenata. Treći izraz je uvjet glatkosti, dobiven derivacijom kvadratne jednadžbe (4.49) i uvrštavanjem u treću jednadžbu (4.50). Dodatni uvjet kojim upotpunjujemo sustav, definira se analogno, ovisno da li je definiran prvom ili drugom derivacijom u nekoj točki.

Kubični splajn

Ako želimo zadržati glatkoću i druge derivacije, što je direktno povezano sa glatkoćom zakrivljenosti, moramo koristiti još veći stupanj polinoma za interpoalciju segmenata. Analogno linearnom i kvadratnom splajnu, **kubični splajn** koristi polinome trećeg

stupnja po segmentima:

$$f_i(x) = a_i + b_i x + c_i x^2 + d_i x^3, \qquad \text{za } i = 1, \ldots, n \tag{4.52}$$

za konstrukciju interpolacijske krivulje kroz $n+1$ zadanih točaka.

Kako kod kubičnog splajna postoji još n dodatnih nepoznanica (ukupno $4n$) u odnosu na kvadratni splajn (koeficijenti kubnog polinoma c_i), potrebno je postaviti dodatne uvjete kako bi se svi koeficijenti mogli odrediti. Uz već korištene uvjete prolaska kroz rubne točke, te uvjet jednakih prvih derivacija na spojevima segmenata, kod kubičnog splajna se uvodi i uvjet da susjedni kubni polinomi imaju jednaku vrijednost druge derivacije u zajedničkoj točki.

Navedni se uvjeti mogu matematički izraziti kao:

$$\begin{aligned} f_i(x_i) &= y_i, && \text{za } i = 1, \ldots, n \\ f_i(x_{i+1}) &= y_{i+1}, && \text{za } i = 1, \ldots, n \\ f_i'(x_{i+1}) &= f_{i+1}'(x_{i+1}), && \text{za } i = 1, \ldots, n-1 \\ f_i''(x_{i+1}) &= f_{i+1}''(x_{i+1}), && \text{za } i = 1, \ldots, n-1. \end{aligned} \tag{4.53}$$

S obzirom da u sustavu (4.53) imamo $4n-2$ jednadžbi, potrebna su još dva uvjeta kako bi sustav bio jednoznačno rješiv. Uobičajeno se uvjetuje da je vrijednost druge derivacije u početnoj i krajnjoj točki jednaka nuli:

$$\begin{aligned} f_1''(x_0) &= 0, \\ f_n''(x_n) &= 0, \end{aligned} \tag{4.54}$$

a tako definirani kubični splajn nazivamo "prirodni splajn".

Analogno uvjetima (4.51) za kvadratni splajn, uvjete prolaska kroz točke te uvjet kontinuirane prve derivacije možemo izraziti linearnim jednadžbama:

$$\begin{aligned} a_i + b_i x_i + c_i x_i^2 + d_i x_i^3 &= y_i, && \text{za } i = 1, \ldots, n \\ a_i + b_i x_{i+1} + c_i x_{i+1}^2 + d_i x_{i+1}^3 &= y_{i+1}, && \text{za } i = 1, \ldots, n \\ b_i + 2c_i x_{i+1} + 3d_i x_{i+1}^2 &= b_{i+1} + 2c_{i+1} x_{i+1} + 3d_{i+1} x_{i+1}^2, && \text{za } i = 1, \ldots, n-1 \end{aligned} \tag{4.55}$$

Za dodatne uvjete koji sadrže druge derivacije vrijedi $f_i''(x) = 2c_i + 6d_i x$, pa se uvjeti neprekidnosti druge derivacije također svode na linearne jednadžbe:

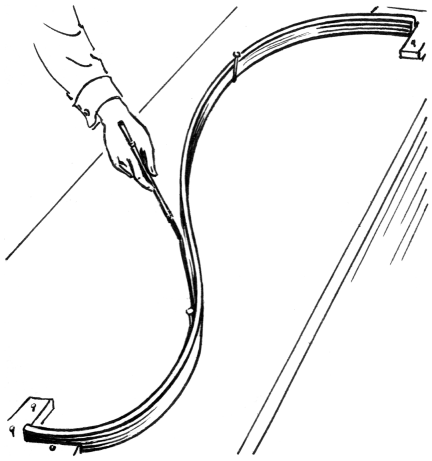
$$2c_i + 6d_i x_{i+1} = 2c_{i+1} + 6d_{i+1} x_{i+1}, \qquad \text{za } i = 1, \ldots, n-1, \tag{4.56}$$

a uvjeti prirodnog splajna na rubnim točkama (4.54) na

$$\begin{aligned} 2c_1 + 6d_1 x_1 &= 0, \\ 2c_n + 6d_n x_n &= 0, \end{aligned} \tag{4.57}$$

Konačne koeficijente kubnih polinoma u kubnom splajnu dobiva se rješavanjem sustava od ukupno $4n$ linearnih jednadžbi definiranih izrazima (4.55), (4.56) i (4.57).

Zanimljivo je da je kubični splajn korišten i prije nego li je matematički formaliziran. Naime, nekada su za crtanje krivulja korištene drvene ili metalne trake (eng. *spline* – otud i potječe ime interpolacijske metode), fizički (npr. čavlima) ograničene u nekoliko točaka (Slika 4.17). Krivulje koje je postizala tako deformirana traka našle su mnogle primjene u dizajnu brodova i zrakoplova. Kubični splajn je zapravo matematički ekvivalent takve elastične trake, a "prirodni splajn" opisuje elastičnu traku sa slobodnim krajevima.



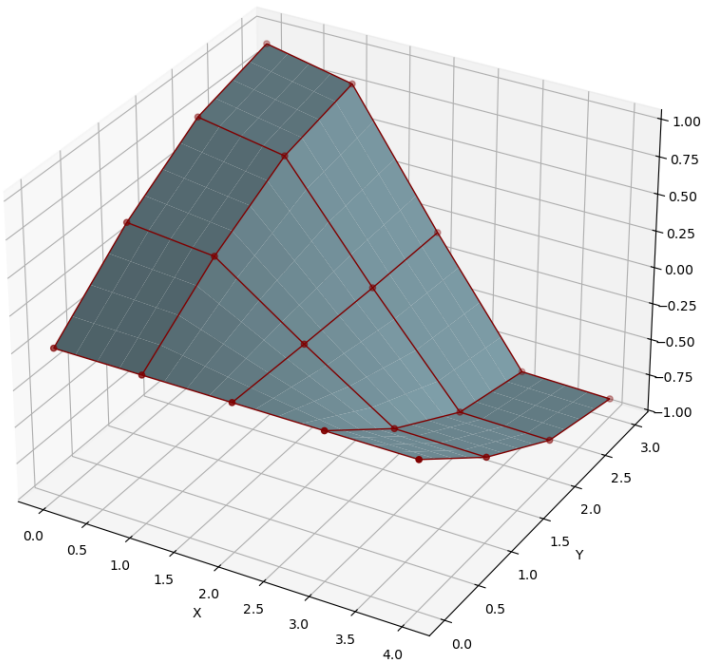
Slika 4.17
Kubični splajn je matematički ekvivalent savijene drvene trake

4.2.3 2D interpolacija

Bilinearna interpolacija

Bilinearnom interpolacijom se mogu procijeniti nepoznate vrijednosti u točki unutar pravokutnika na temelju vrijednosti u njegova četiri vrha. Metoda se provodi tako da se najprije primijeni linearna interpolacija u jednom smjeru, a zatim u drugom. Redoslijed postupka, tj. redoslijed smjerova po kojima se provodi linearna interpolacija, nije bitan i nema utjecaj na rješenje. Metodu je lako implementirati i jednostavno se primjenjuje na većem setu točaka na pravokutnoj mreži, računski je učinkovita i stabilna, što je čini pogodnom za primjene u stvarnom vremenu. Manje je precizna u usporedbi sa složenijim 2D interpolacijskim metodama te ne osigurava glatkoću interpolacijske plohe (Slika 4.18).

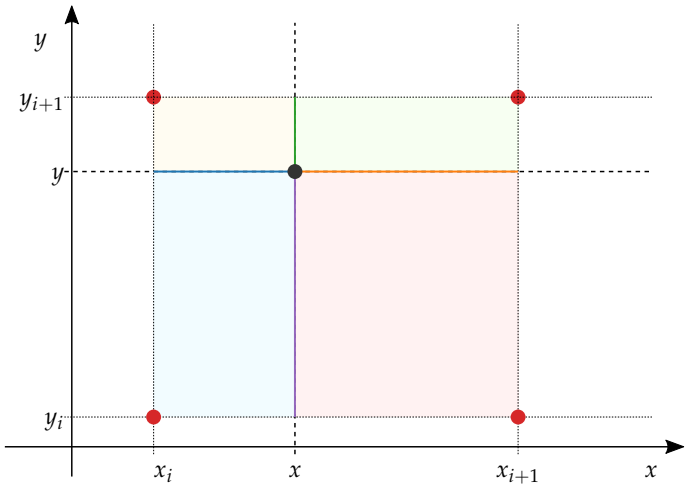
Bilinearna interpolacija se često koristi u obradi slike, primjerice pri skaliranju ili rotaciji, kako bi se dobili glatki prijelazi između piksela. Međutim, može uzrokovati zamućenje slike jer ne uzima u obzir nagle promjene ili rubove.



Slika 4.18
Primjer bilinearne interpolacije

Za željenu točku (x,y), potrebno je odrediti poznate koordinate na pravokutnoj

mreži (x_i, x_{i+1} i y_j, y_{j+1}) između kojih se točka nalazi. Te točke formiraju pravokutnu ćeliju (Slika 4.19).



Slika 4.19
Skica bilinearne interpolacije

Interpolacijska vrijednost se izračunava eksplicitno iz poznatih vrijednosti F na okolnim točkama pravokutne mreže i težinskih faktora udaljenosti točke od rubova ćelije. Rezultati ovise o udaljenosti interpolirane točke od sva četiri susjedna vrha, pri čemu bliže točke imaju veći utjecaj:

$$\begin{aligned} f(x,y) =& F_{i,j} \cdot \frac{x_{i+1}-x}{x_{i+1}-x_i} \cdot \frac{y_{j+1}-y}{y_{j+1}-y_j} \\ &+ F_{i+1,j} \cdot \frac{x-x_i}{x_{i+1}-x_i} \cdot \frac{y_{j+1}-y}{y_{j+1}-y_j} \\ &+ F_{i,j+1} \cdot \frac{x_{i+1}-x}{x_{i+1}-x_i} \cdot \frac{y-y_j}{y_{j+1}-y_j} \\ &+ F_{i+1,j+1} \cdot \frac{x-x_i}{x_{i+1}-x_i} \cdot \frac{y-y_j}{y_{j+1}-y_j} \end{aligned} \tag{4.58}$$

Bikubična interpolacija

Bikubična interpolacija je napredna metoda interpolacije koja za izračun vrijednosti u jednoj točki koristi 16 okolnih uzoraka raspoređenih u mreži 4×4. Temelji se na kubičnim polinomima koji se primjenjuju u oba smjera, pri čemu je producirana interpolacijska funkcija umnožak polinoma po oba smjera. Matematički, vrijednost se opisuje kubičnom funkcijom čiji koeficijenti ovise o vrijednostima točaka i njihovim derivacijama ili aproksimacijama derivacija.

Ovakva formulacija omogućuje glatke prijelaze i kontinuirane prve derivacije (Slika 4.20), što rezultira manjim vizualnim artefaktima. Zbog veće računske zahtjevnosti koristi se kada je kvaliteta interpolacije važnija od brzine izvođenja.

Vrijednost u točki (x,y), koja se nalazi u ćeliji omeđenoj točkama x_i, x_{i+1} i y_j, y_{j+1}, je definirana funkcijom:

$$f(x,y) = \sum_{i=0}^{3}\sum_{i=0}^{3} a_{i,j} \cdot \left(\frac{x-x_i}{x_{i+1}-x_i}\right)^i \cdot \left(\frac{y-y_j}{y_{j+1}-y_j}\right)^j \tag{4.59}$$

gdje je 16 nepoznatih koeficijenata $a_{i,j}$ potrebno odrediti iz pretpostavljenih uvjeta. Ti uvjeti određuju osobine interpolacijske funkcije u promatranoj ćeliji, uključujući

glatkoću na rubovima ćelije što je osigurano neprekidnošću prve derivacije (u oba smjera) i mješovite derivacije na rubovima ćelije:

Funkcijska vrijednost:

$$\left.\begin{aligned} f(x_i, y_i) &= F_{i,j} \\ f(x_{i+1}, y_i) &= F_{i+1,j} \\ f(x_i, y_{i+1}) &= F_{i,j+1} \\ f(x_{i+1}, y_{i+1}) &= F_{i+1,j+1} \end{aligned}\right\} \textit{4 uvjeta}$$

Prva derivacija funckije (po x):

$$\left.\begin{aligned} f_x(x_i, y_i) &= \nabla_x F_{i,j} \\ f_x(x_{i+1}, y_i) &= \nabla_x F_{i+1,j} \\ f_x(x_i, y_{i+1}) &= \nabla_x F_{i,j+1} \\ f_x(x_{i+1}, y_{i+1}) &= \nabla_x F_{i+1,j+1} \end{aligned}\right\} \textit{4 uvjeta}$$

Prva derivacija funckije (po y):

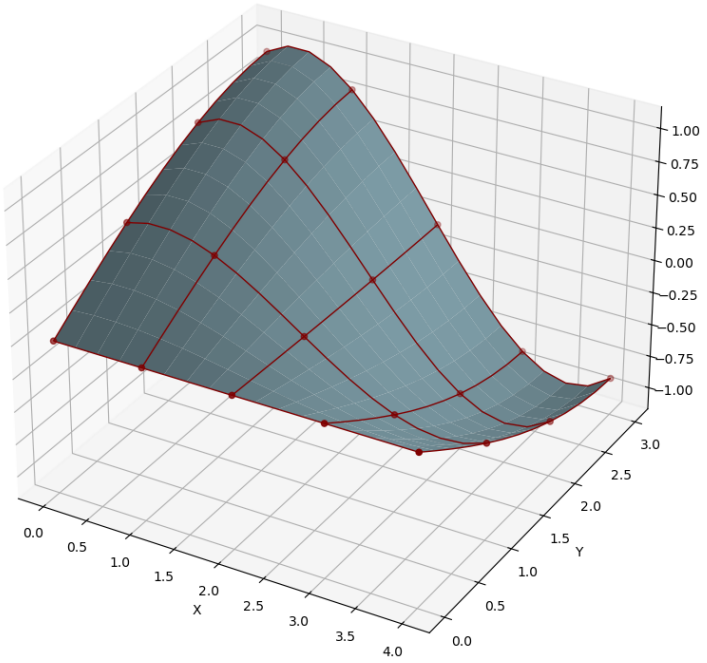
$$\left.\begin{aligned} f_y(x_i, y_i) &= \nabla_y F_{i,j} \\ f_y(x_{i+1}, y_i) &= \nabla_y F_{i+1,j} \\ f_y(x_i, y_{i+1}) &= \nabla_y F_{i,j+1} \\ f_y(x_{i+1}, y_{i+1}) &= \nabla_y F_{i+1,j+1} \end{aligned}\right\} \textit{4 uvjeta}$$

Mješovita derivacija funckije (po x i y:)

$$\left.\begin{aligned} f_{xy}(x_i, y_i) &= \nabla_{xy} F_{i,j} \\ f_{xy}(x_{i+1}, y_i) &= \nabla_{xy} F_{i+1,j} \\ f_{xy}(x_i, y_{i+1}) &= \nabla_{xy} F_{i,j+1} \\ f_{xy}(x_{i+1}, y_{i+1}) &= \nabla_{xy} F_{i+1,j+1} \end{aligned}\right\} \textit{4 uvjeta}$$

Ovi uvjeti, čine sustav od 16 linearnih jednadžbi s 16 nepoznanica ($a_{i,j}$ za $i, j = 0, \ldots, 3$) za svaku ćeliju pravokutne mreže.

Funkcijske vrijednosti u interpolacijskim točkama korištene u prva četiri uvjeta su, naravno, zadane i poznate, međutim derivacije u tim točkama nisu. Uobičajeno je da se one aproksimiraju, najčešće numerički iz zadane mreže točaka (npr. koristeći `numpy.gradient` funkciju). Tada su poznati sve vrijednosti u postavljenim uvjetima i moguće je rješavanje linearnog sustava te izračun interpolirane vrijednosti (4.59) za pojedinu ćeliju.



Slika 4.20
Primjer bikubične interpolacije

4.2.4 Interpolacija pomoću SciPy modula

Modul SciPy nudi zaseban modul `interpolate` koji je posvećen isključivo rješavanju interpolacijskih problema. Iako je modul bogat jednodimenzionalnim, dvodimenzionalnim i višedimenzionalnim interpolacijskim metodama i raznim pomoćnim mogućnostima, ovdje ćemo samo spomenuti osnovne metode za 1D interpolaciju koje su opisane u prethodnim poglavljima.

Lagrangeov interpolacijski polinom može se lako izračunati koristeći funkciju `lagrange`, čija je osnovna sintaksa:

```
scipy.interpolate.lagrange(X, Y)
```

gdje su `X` i `Y` vektori iste veličine koji sadrže koordinate zadanih točaka. Funkcija `lagrange` vraća specijalni objekt koji se može koristiti za izračun polinoma istom sintaksom kao da koristimo funkciju, ali nudi i druge alate i informacije specifične za polinome.

Sve splajn interpolacijske metode mogu se pozivati pomoću funkcije `make_interp_spline`, koja kreira optimizacijsku funkciju.

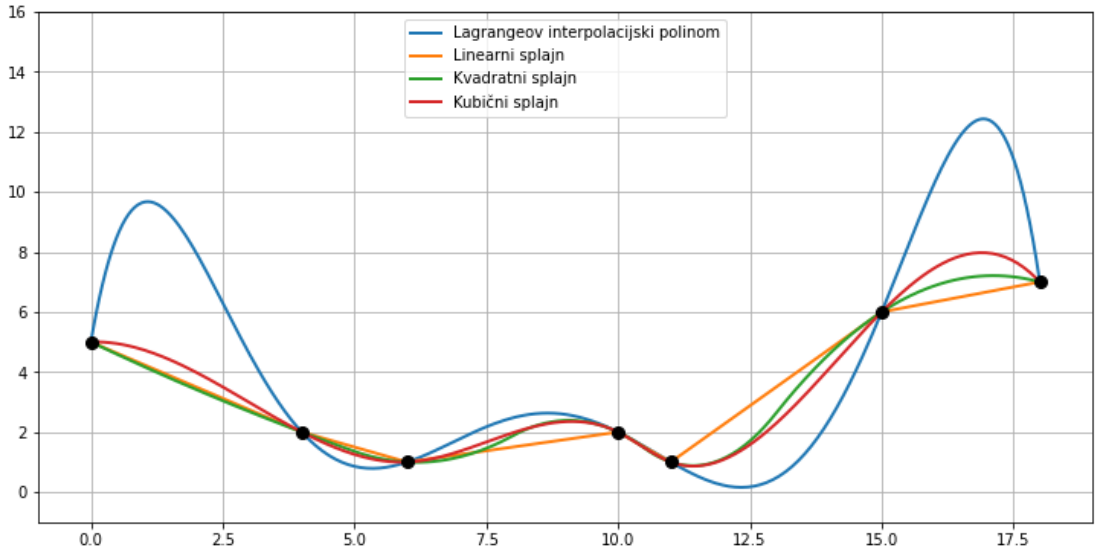
```
scipy.interpolate.make_interp_spline(X, Y, k)
```

gdje su `X` i `Y` koordinate zadanih točaka, a argument `k` definira vrstu baznog splajna. Vrsta baznog polinoma se određuje njegovim stupnjem, odnosno `'k=1'` odgovara linearnom, `'k=2'` kvadratnom, a `'k=3'` kubičnom splajnu. Ukoliko se vrsta splajna ne postavi onda se dobije kubični splajn. Objekt koji vraća funkcija `make_interp_spline` se može koristiti za evaluaciju točaka na isti način kao da se radi o običnoj funkciji.

Primjer korištenja SciPy interpolacijskih metoda te njihova usporedba demonstrirana je u Programskom kodu 4.5, a dobivena vizualizacija je prikazana na Slici 4.21. Očito je da sa porastom stupnja polinoma rastu oscilacije, kao što je i jasno zašto se kao prvi izbor u praksi najčešće koriste splajn interpolacije.

Python kod 4.5 Interpolacija metodama SciPy modula i njihova usporedba

```
import numpy as np
import scipy.interpolate as intrp
import matplotlib.pyplot as plt

# Zadane točke
n = 5
X = [0, 4, 6, 10, 11, 15, 18]
Y = [5, 2, 1, 2, 1, 6, 7]

# Gusta podjela x varijable
x = np.linspace(0, 18, 300)

# Interpolacijske funkcije dobivene pomoću scipy.inetrpolate modula
f1 = intrp.lagrange(X, Y)
f2 = intrp.make_interp_spline(X, Y, 1)
f3 = intrp.make_interp_spline(X, Y, 2)
f4 = intrp.make_interp_spline(X, Y, 3)

# Vizualizacija
plt.figure(figsize=(10, 5))
plt.plot(x, f1(x), lw=2, label='Lagrangeov interpolacijski polinom')
plt.plot(x, f2(x), lw=2, label='Linearni splajn')
plt.plot(x, f3(x), lw=2, label='Kvadratni splajn')
plt.plot(x, f4(x), lw=2, label='Kubični splajn')

plt.plot(X, Y, 'ko', ms=8)
plt.grid()
plt.xlim(-1, 19)
plt.ylim(-1, 16)
plt.legend(loc='upper center')
plt.subplots_adjust(left=0.03, right=0.99, bottom=0.05, top=0.98)
plt.show()
```



Slika 4.21 Usporedba interpolacijskih metoda prema izvornom kodu 4.5

Više-dimenzionalna interpolacija na pravokutnim mrežama omogućena je pomoću klase `RegularGridInterpolator` koja prima koordinate točaka u redcima matrice te pripadajuće funkcijske vrijednosti. Koristi se sintaksa

```
scipy.interpolate.RegularGridInterpolator(tocke, F)
```

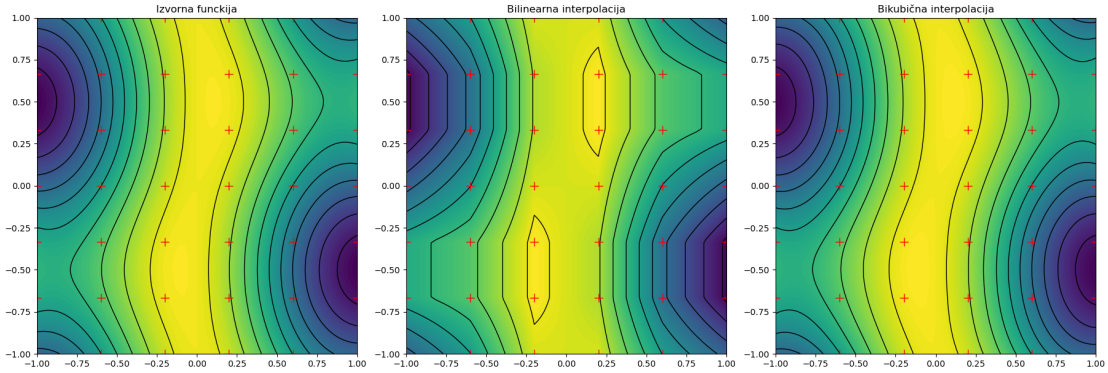
pri čemu je argument `tocke` matrica veličine `[n_points, n_dimensions]`, dok je `F` vektor veličine `[n_points]`.

Primjer upotrebe `RegularGridInterpolator` klase za bilinearnu i bikubičnu interpolaciju na dvo-dimenzionalnim podacima dan je u Izvornom kodu 4.6, a rezultati su prikazani na Slici 4.22.

Python kod 4.6 Interpolacija metodama SciPy modula i njihova usporedba

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

# Izvorna funkcija
def f(x, y):
    return np.cos(np.pi*x) + x * np.sin(np.pi*y)
x_mg, y_mg = np.meshgrid(np.linspace(-1, 1, 101),
                         np.linspace(-1, 1, 101))
z = f(x_mg, y_mg)

# Interpolacijske točke
X = np.linspace(-1, 1, 6)
Y = np.linspace(-1, 1, 7)
X_mg, Y_mg = np.meshgrid(X, Y, indexing='ij')
Z = f(X_mg, Y_mg)

# Bilinearna interpolacija
bilinear = sp.interpolate.RegularGridInterpolator((X, Y), Z)
z_bilinear = bilinear((x_mg, y_mg))

# Bikubična interpolacija
bicubic = sp.interpolate.RegularGridInterpolator((X, Y), Z, method='cubic')
z_bicubic = bicubic((x_mg, y_mg))

# Vizualizacija
fig, [ax1, ax2, ax3] = plt.subplots(figsize=(18, 6), ncols=3, constrained_layout=
    True)

ax1.set_title('Izvorna funckija')
ax1.contourf(x_mg, y_mg, z, 100)
ax1.contour(x_mg, y_mg, z,
            levels=10,
            colors='k',
            linestyles='-',
            linewidths=1)

ax2.set_title('Bilinearna interpolacija')
ax2.contourf(x_mg, y_mg, z_bilinear, 100)
ax2.contour(x_mg, y_mg, z_bilinear,
            levels=10,
            colors='k',
            linestyles='-',
            linewidths=1)

ax3.set_title('Bikubična interpolacija')
```

```
ax3.contourf(x_mg, y_mg, z_bicubic, 100)
ax3.contour(x_mg, y_mg, z_bicubic,
            levels=10,
            colors='k',
            linestyles='-',
            linewidths=1)

for ax in [ax1, ax2, ax3]:
    ax.plot(X_mg, Y_mg, 'r+', ms=10)
plt.show()
```

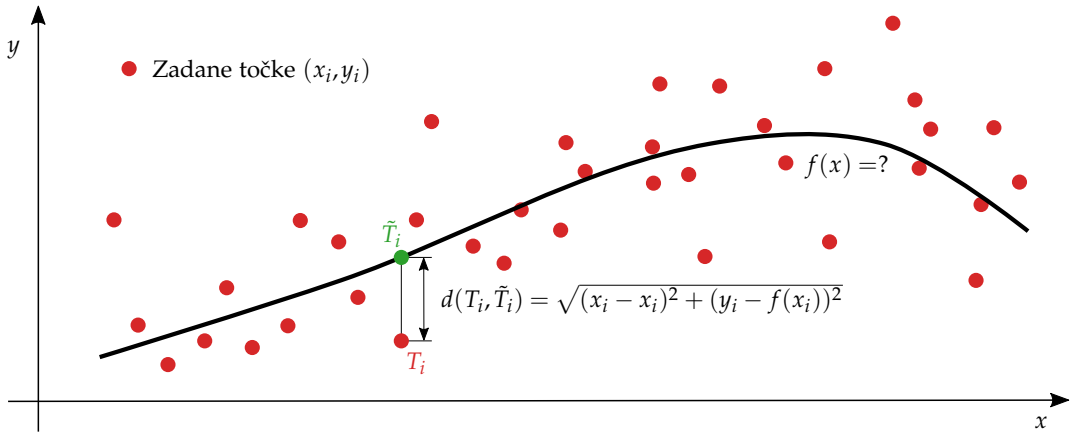


Slika 4.22 Usporedba interpolacijskih metoda prema izvornom kodu 4.6

4.3 Regresijska analiza

Stefan Ivić
Siniša Družeta

Zamislimo da promatramo određene parametre između kojih očekujemo određenu vezu (zavisnost). Recimo da imamo dvije varijable, x i y, gdje na temelju stanja varijabli u pojedinim točkama (x_i, y_i) za $i = 1,2,...,n$ vidimo da postoji određena veza $y(x)$ (Slika 4.23). Pri tome točke nisu potpuno pouzdane (točne), bilo zbog pogreške u prikupljanju podataka (tipično su to pogreške mjerenja) ili zbog toga što nam u analizi nedostaje utjecaj nekih trećih parametara o kojima nemamo podataka. U tom slučaju, regresijska analiza podrazumijeva određivanje funkcijske veze $y = f(x)$ koja najbolje opisuje zavisnost u podacima.



Slika 4.23 Točke u kojima postoji veza između varijabli x i y

Dakle, ako pretpostavljamo funkcijsku zavisnost $y = f(x)$ i izaberemo neku vrstu regresijske funkcije $f(x)$, treba nam postupak koji bi nam omogućio da jednoznačno odredimo najbolju $f(x)$ od svih mogućih zamislivih funkcija tog tipa. Drugim riječima, potrebno je pronaći funkciju "najbližu" danim točkama, ali koja ne prolazi točno kroz zadane točke, nego nekako između njih.

Kako odrediti koliko je ova funkcija "blizu" zadanih točaka? Za svaku zadanu točku $T_i(x_i, y_i)$ svakako možemo odrediti udaljenost do njoj pripadne točke na regresijskoj funkciji $\tilde{T}_i(x_i, f(x_i))$, tj. možemo usporediti koliko se razlikuje $y_i(x_i)$ od $f(x_i)$. Dakle, jednostavno izračunamo udaljenost između točaka T_i i $\tilde{T}_i$:

$$d(T_i, \tilde{T}_i) = \sqrt{(x_i - x_i)^2 + (y_i - f(x_i))^2}. \tag{4.60}$$

Postupak moramo provesti za sve točke $i = 1,2,...,n$ i logično bi bilo sve dobivene udaljenosti zbrojiti, no razni pristupi rješavanja regresijske krivulje pokazuju da je praktično i korisno ih pri tome još i kvadrirati. Time dobijemo ukupnu "ocjenu" regresijske funkcije, definiranu kao:

$$S = \sum_{i=1}^{n} d(T_i, \tilde{T}_i)^2 = \sum_{i=1}^{n} (y_i - f(x_i))^2. \tag{4.61}$$

Sada nam preostaje da nekako odredimo onu $f(x)$ za koju će suma S biti najmanja. Pošto biranje funkcije f iz skupa svih mogućih funkcija nije izvedivo, odabire se model

funkcije, npr. linearni, polinomni, eksponencijalni ili sl. Time se određivanje nepoznate funkcije prebacuje na računanje nepoznatih parametara odnosno koeficijenata modela.

4.3.1 Linearna regresija

Ako se odlučimo da regresijska funkcija $f(x)$ bude pravac

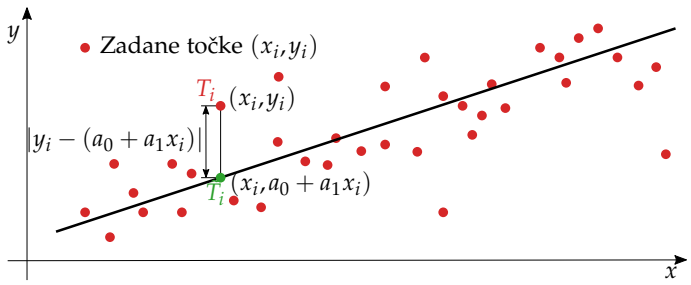
$$y = a_0 + a_1 \cdot x, \tag{4.62}$$

onda se ukupna odstupanja od danih točaka svode na:

$$S = \sum_{i=1}^{n} (y_i - (a_0 + a_1 x_i))^2, \tag{4.63}$$

gdje su a_0 i a_1 nepoznati koeficijenti regresijskog pravca.

Određivanje regresijskog pravca podrazumijeva određivanje najboljeg od svih mogućih pravaca koji prolaze otprilike kroz zadane točke (Slika 4.24). Najbolji pravac će biti onaj koji će imati najmanju sumu S i ako primijetimo da ona ovisi o parametrima pravca a_0 i a_1 onda možemo tražiti minimum funkcije $S(a_0, a_1)$.



Slika 4.24
Skica regresijskog pravca

S obzirom da je $S(a_0, a_1)$ funkcija dvije varijable, minimum sume S će biti postignut za one vrijednosti a_0 i a_1 za koje vrijedi:

$$\begin{aligned} \frac{\partial S}{\partial a_0} &= 0, \\ \frac{\partial S}{\partial a_1} &= 0. \end{aligned} \tag{4.64}$$

Kada to izračunamo:

$$\begin{aligned} \frac{\partial S}{\partial a_0} &= \sum_{i=1}^{n} 2(y_i - (a_0 + a_1 x_i)) \cdot (-1), \\ \frac{\partial S}{\partial a_1} &= \sum_{i=1}^{n} 2(y_i - (a_0 + a_1 x_i)) \cdot (-x_i), \end{aligned} \tag{4.65}$$

dobijemo sustav dviju linearnih jednadžbi:

$$\begin{aligned} a_0 \cdot n \quad &+ \quad a_1 \cdot \sum_{i=1}^{n} x_i \quad = \quad \sum_{i=1}^{n} y_i \\ a_0 \cdot \sum_{i=1}^{n} x_i \quad &+ \quad a_1 \cdot \sum_{i=1}^{n} x_i^2 \quad = \quad \sum_{i=1}^{n} x_i y_i. \end{aligned} \tag{4.66}$$

Rješenjem ovog sustava dobivamo koeficijente a_0 i a_1 koji daju najmanje moguće ukupno odstupanje od zadanih točaka S, čime smo odredili regresijski pravac $y = a_0 + a_1 x$.

4.3.2 Polinomna regresija

Uzmemo li za regresijsku funkciju polinom višeg stupnja, potrebno je odabrati polinomni model tj. stupanj polinoma. Koeficijenti takvog polinoma se nalaze iz ranije navedenih kriterija regresije. Treba primjetiti da je polinom linearan u odnosu na svoje koeficijente pa regresija daje i u ovom slučaju linearne jednadžbe za nepoznate koeficijente polinoma.

Dakle, za polinom m-tog stupnja

$$f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m, \tag{4.67}$$

postupak regresije svodi se na određivanje parametara $a_0, a_1, ..., a_m$.

To znači da moramo minimizirati sumu

$$S = \sum_{i=1}^{n} (y_i - (a_0 + a_1 x + a_2 x^2 + ... + a_m x^m))^2 \tag{4.68}$$

odnosno mora vrijediti:

$$\frac{\partial S}{\partial a_j} = 0, \quad j = 0, 1, ..., m \tag{4.69}$$

Ako izračunamo sve potrebne parcijalne derivacije

$$\begin{aligned} \frac{\partial S}{\partial a_j} &= \sum_{i=1}^{n} 2(y_i - (a_0 + a_1 x_i + \cdots + a_m x_i^m)) \cdot (-x_i^j) \\ &= -2 \sum_{i=1}^{n} \left(y_i x_i^j - \left(\sum_{k=0}^{m} a_k x_i^{j+k} \right) \right) \end{aligned} \tag{4.70}$$

za $j = 0, 1, ..., m$, dobit ćemo linearni sustav:

$$\sum_{k=0}^{m} a_k \left(\sum_{i=1}^{n} x_i^{k+j} \right) = \sum_{i=1}^{n} y_i x_i^j \tag{4.71}$$

za $j = 0, 1, ..., m$. Ako uvedemo oznake:

$$\begin{aligned} \alpha_{jk} &= \sum_{i=1}^{n} x_i^{k+j}, \quad j, k = 0, 1, ..., m \\ \beta_j &= \sum_{i=1}^{n} y_i \cdot x_i^j, \quad j = 0, 1, ..., m \end{aligned} \tag{4.72}$$

sustav možemo zapisati i skraćeno:

$$\sum_{k=0}^{m} \alpha_{jk} \cdot a_k = \beta_j, \quad j, k = 0, 1, ..., m. \tag{4.73}$$

Rješavanjem ovog sustava od $m + 1$ jednadžbi i $m + 1$ nepoznanica dobit ćemo koeficijente regresijskog polinoma.

4.3.3 Primjena linearne regresije na složenije modele

Ako podacima na kojima provodimo regresiju nisu pogodni za linearnu ili polinomnu regresiju, možemo jednostavnim matematičkim "trikovima" složenije regresijske modele svesti na linearnu regresiju. Ovdje ćemo pokazati tri primjera:

- Eksponencijalni regresijski model
- Regresijski model baziran na potenciji
- Regresijski model rasta do zasićenja

Pod eksponencijalnim modelom podrazumijevamo upotrebu regresijske funkcije

$$y = ae^{bx}, \tag{4.74}$$

gdje se moramo riješiti nelinearnosti u odnosu na parametre modela da bi mogli provesti postupak određivanja parametara a i b. To možemo napraviti logaritmiranjem:

$$\ln y = \ln a + bx, \tag{4.75}$$

nakon čega možemo provesti linearnu regresiju na točkama $(x_i, \ln y_i)$, čime ćemo odrediti parametre $\ln a$ i b (Slika 4.25 A).

Uzmemo li model baziran na potenciji (eng. *power equation*)

$$y = ax^b, \tag{4.76}$$

ponovno si možemo pomoći logaritmiranjem:

$$\ln y = \ln a + b \ln x, \tag{4.77}$$

gdje se sada točke $(\ln x_i, \ln y_i)$ mogu modelirati linearnom regresijom, čime ćemo dobiti parametre $\ln a$ i b (Slika 4.25 B).

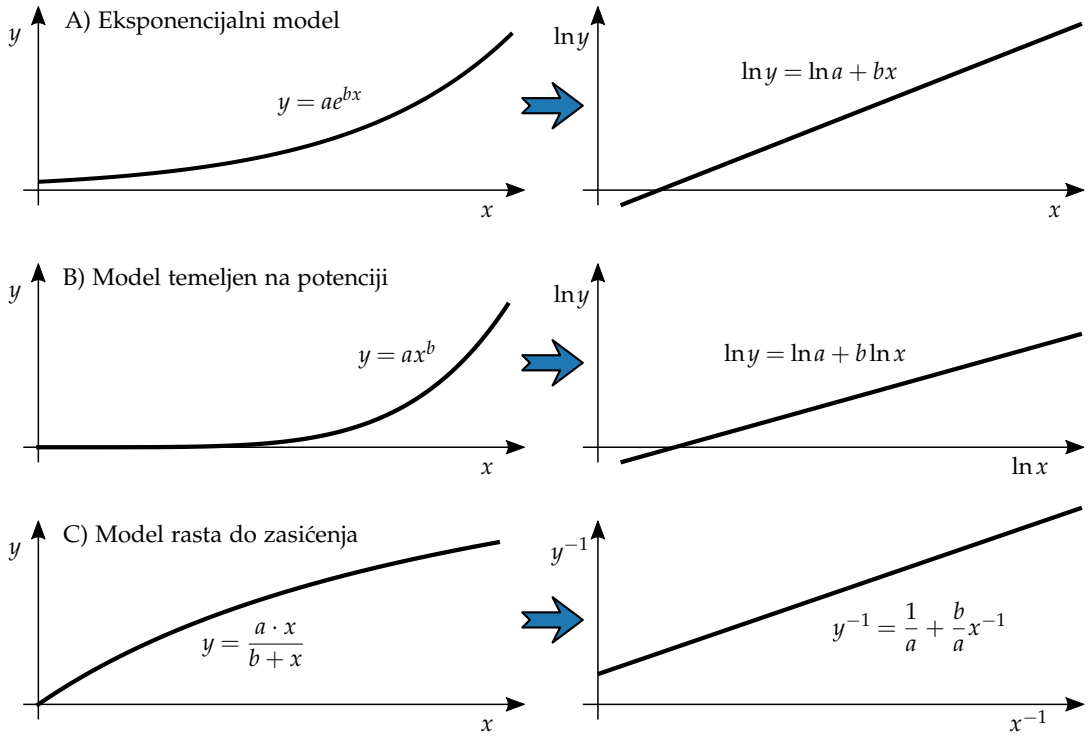
Jednadžba rasta do zasićenja (eng. *saturation-growth-rate equation*) glasi:

$$y = \frac{a \cdot x}{b + x} \tag{4.78}$$

i može se svesti na oblik

$$y^{-1} = \frac{1}{a} + \frac{b}{a} x^{-1}, \tag{4.79}$$

nakon čega se točke (x_i^{-1}, y_i^{-1}) mogu modelirati linearnom regresijom, čime ćemo odrediti parametre $\frac{1}{a}$ i $\frac{b}{a}$ (Slika 4.25 C).



Slika 4.25 Svođenje na linearni model: eksponencijalni model (A), model temeljen na potenciji (B) i model rasta do zasićenja (C)

4.3.4 Regresijska analiza pomoću NumPy modula

Za kreiranje polinoma možemo koristiti `Polynomial` objekt modula NumPy (podmodul `polynomial`) koji nam omogućuje prilagođavanje polinoma željenog stupnja zadanim točkama. Osnovna sintaksa funkcije glasi:

```
numpy.polynomial.Polynomial.fit(x, y, deg)
```

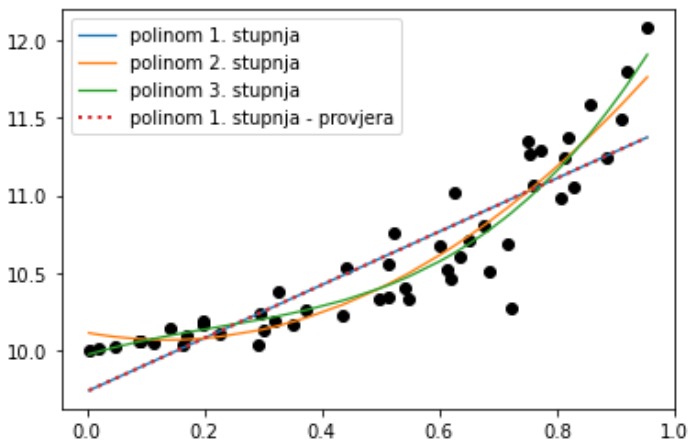
Argumenti `x` i `y` su nizovi brojeva tj. koordinate poznatih točaka. Parametar `deg` definira stupanj regresijskog polinoma m. Funkcija `Polynomial.fit` kreira objekt koji se može koristiti kao običnu funkciju za potrebe izračunavanja vrijednosti polinoma za željeni x. Koeficijente regresijskog polinoma možemo dobiti očitanjem `Polynomial.convert().coef` atributa, čime ćemo dobiti koeficijente redom od najnižeg stupnja $(a_0, a_1, a_2, \ldots, a_m)$. Više informacija o radu s polinomima može se naći u poglavlju 2.6.

Primjena `Polynomial.fit` funkcije te ručni izračun koeficijenata regresijskog pravca prikazani su u kodu 4.7, uz vizualizaciju rezultata na Slici 4.26.

Python kod 4.7 Polinomna regresija

```
import numpy as np
from numpy.polynomial import Polynomial
import matplotlib.pyplot as plt

# Učitavanje i vizualizacija poznatih podataka
X, Y = np.loadtxt('set_1.txt', unpack=True)
plt.plot(X, Y, 'ko')

# Polinom 1. stupnja
f1 = Polynomial.fit(X, Y, 1)
```

```
# Polinom 2. stupnja
f2 = Polynomial.fit(X, Y, 2)

# Polinom 3. stupnja
f3 = Polynomial.fit(X, Y, 3)

plt.plot(X, f1(X), lw=1.25, label='polinom 1. stupnja')
plt.plot(X, f2(X), lw=1.25, label='polinom 2. stupnja')
plt.plot(X, f3(X), lw=1.25, label='polinom 3. stupnja')

# Linearna regresija "na ruke"
sumX = np.sum(X)
sumY = np.sum(Y)
sumX2 = np.sum(X**2)
sumXY = np.sum(X*Y)

# Priprema linearnog sustava
A = np.array([[len(X), sumX],
              [sumX, sumX2]])
B = np.array([sumY, sumXY])

# Rješavanje sustava
a_m = np.linalg.solve(A, B)

# Definicija polinoma
def funkcija(x):
    return a_m[0] + a_m[1]*x

# Krajnje točke regresijskog pravca
p1 = [min(X), funkcija(min(X))]
p2 = [max(X), funkcija(max(X))]

plt.plot([p1[0], p2[0]], [p1[1], p2[1]], lw=2, ls=':', label='polinom 1. stupnja -
    provjera')
plt.legend()
```



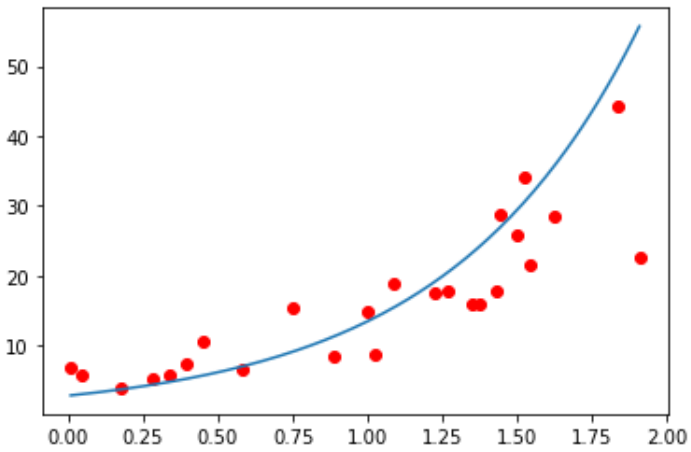
Slika 4.26
Rezultati polinomne regresije

Programska implementacija eksponencijalne regresije svođenjem na linearni model opisana je u kodu 4.8 (rezultat na slici 4.27).

Python kod 4.8 Eksponencijalna regresija putem linearnog modela

```
import numpy as np
```

```
from numpy.polynomial import Polynomial
import matplotlib.pyplot as plt

X, Y = np.loadtxt('set_2.txt', unpack=True)
plt.plot(X, Y, 'ro')

polyreg = Polynomial.fit(X, np.log(Y), 1)
b, ln_a = polyreg.convert().coef

x = np.linspace(np.min(X), np.max(X), 200)
y = np.exp(ln_a) * np.exp(x * b)
plt.plot(x, y)
```



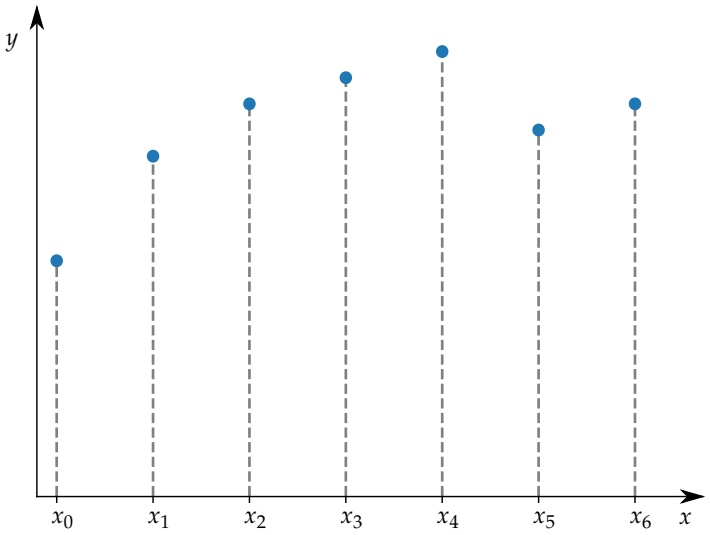
Slika 4.27
Rezultati eksponencijalne regresije

4.4 Numerička integracija

Jerko Škifić

Numerička integracija podrazumijeva aproksimaciju određenog integrala $\int_a^b f(x)\mathrm{d}x$, što je posebno korisno u situacijama u kojima:

- nije moguće izraziti analitičko rješenje
- otežana evaluacija integrala
- funkcija $f(x)$ je zadana kao niz diskretnih točaka (Slika 4.28)



Slika 4.28
Primjer funkcije zadane nizom točaka

Formalno, za definiranu funkciju $f : [a,b] \to \mathcal{R}$ na zatvorenom intervalu $[a,b]$ definiramo particiju intervala $\mathcal{P}$:

$$\mathcal{P} = \{[x_0,x_1],[x_1,x_2],\dots[x_{n-1},x_n]\} \tag{4.80}$$

za koju vrijedi $a = x_0 < x_1 < x_2 < \cdots < x_n = b$.

Sada možemo odrediti širinu svake particije intervala Δx_i, pomoću izraza

$$\Delta x_i = x_i - x_{i-1}. \tag{4.81}$$

Nadalje, za svaki interval odredimo x_i^* na način da vrijedi $x_i^* \in [x_{i-1},x_i]$.

Tek sada možemo izračunati određeni integral pomoću sljedećeg izraza:

$$I = \int_a^b f(x)dx = \lim_{||\Delta x_i||\to 0}\sum_{i=1}^{n} f(x_i^*)\Delta x_i, \tag{4.82}$$

gdje n označava broj intervala.

Newton-Cotes formule

Razvijen je veći broj metoda za numeričku integraciju, od kojih se najčešće zbog svoje jednostavnosti primjenjuju *Newton-Cotes* formule. Osnovna ideja se svodi na supstituciju podintegralne funkcije funkcijom koju je lako integrirati:

$$I = \int_a^b f(x)\mathrm{d}x \approx \int_a^b f_n(x)\mathrm{d}x, \tag{4.83}$$

gdje se funkcija f_n obično definira kao polinom n-tog stupnja:

$$f_n(x) = a_0 + a_1 x + a_2 x^2 + \ldots a_n x^n. \tag{4.84}$$

4.4.1 Pravilo lijeve, desne i srednje točke

S praktične strane, ukoliko nam je poznata funkcija f, postoje dvije sitne prepreke koje nam otežavaju numeričko rješavanje određenog integrala pomoću izraza (4.82):

- Širina intervala Δx_i ne može biti beskonačno malena
- Moramo iznaći način da primjereno odaberemo x_i^*

U tom slučaju je moguće približno riješiti određeni integral

$$I = \int_a^b f(x)dx \approx \sum_{i=1}^{n} f(x_i^*)\Delta x_i. \tag{4.85}$$

Ovdje se širine intervala odrede pomoću izraza (4.81), dok se odabir x_i^* obično svodi na jedan od tri slučaja:

$$x_i^* = \begin{cases} x_{i-1} & \text{lijeva točka intervala } \Delta x_i \\ \frac{1}{2}(x_{i-1} + x_i) & \text{sredina intervala } \Delta x_i \\ x_i & \text{desna točka intervala } \Delta x_i \end{cases} \tag{4.86}$$

U sva tri slučaja, f_n iz izraza (4.83) je konstanta $f_n(x_i^*) = a_0$ (tj. polinom nultog stupnja), s tim da za a_0 uzimamo funkcijsku vrijednost početka, sredine ili kraja intervala.

Sada možemo napisati tri jednostavne metode za izračunavanje određenog integrala. Odredimo jedinstvenu širinu intervala $\Delta x = \Delta x_1 = \Delta x_2 = \Delta x_3 = \cdots = \Delta x_n$ pomoću izraza

$$\Delta x = \frac{b-a}{n},$$

gdje je, ponovimo, n broj intervala.

Odaberemo li za x_i^* lijevu točku intervala, konstruirali smo metodu pod nazivom **pravilo lijeve točke** (Slika 4.29a) te izraz (4.85) postaje

$$I \approx \Delta x \sum_{i=1}^{n} f(x_{i-1}) = \frac{b-a}{n} \sum_{i=1}^{n} f(x_{i-1}). \tag{4.87}$$

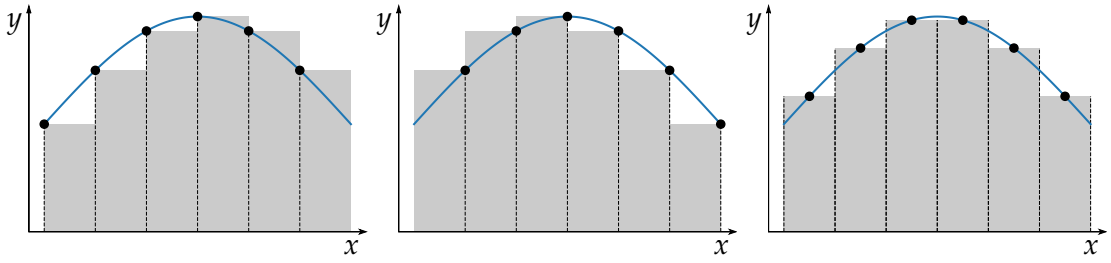
U slučaju odabira desne točke intervala za x_i^*, konstruirali smo metodu pod nazivom **pravilo desne točke** (Slika 4.29b), čime izraz (4.85) postaje

$$I \approx \Delta x \sum_{i=1}^{n} f(x_i) = \frac{b-a}{n} \sum_{i=1}^{n} f(x_i). \tag{4.88}$$

Odabir sredine intervala za x_i^* rezultira numeričkom metodom naziva **pravilo srednje točke** (Slika 4.29c). Izraz (4.85) postaje

$$I \approx \Delta x \sum_{i=1}^{n} f(x_{i-1/2}) = \frac{b-a}{n} \sum_{i=1}^{n} f(x_{i-1/2}), \tag{4.89}$$

gdje je $f(x_{i-1/2}) = f\left(\frac{1}{2}(x_{i-1} + x_i)\right)$.



(a) Pravilo lijeve točke (b) Pravilo desne točke (c) Pravilo srednje točke

Slika 4.29 Primjer integracije funcije $f(x) = \sin(x) + 1$ na intervalu $[0, \pi]$ pravilom lijeve, desne i srednje točke

Implementacija pravila srednje točke u Pythonu dana je u programskom kodu 4.9:

Python kod 4.9 Pravilo srednje točke

```
import numpy as np

def f(t):
    # definicija podintegralne funkcije
    return np.sin(t)

a, b = 1, 2     # granice integriranja
n = 20          # broj intervala
h = (b - a) / n # širina intervala

x = np.linspace(a + h/2, b - h/2, n)
y = f(x)
integral = h * np.sum(y)

print('Integral: ', integral)
print('Točan rezultat: ', np.cos(1) - np.cos(2))
```

Greška pravila srednje točke

Procjena greške pravila srednje točke izražava se sljedećim izrazom:

$$E = \frac{(b-a)^3}{24n^2} f''(\zeta), \tag{4.90}$$

gdje su a i b granice integriranja, a n broj intervala (segmenata). Iznenađujuće, iako je vrlo jednostavna, metoda srednje točke može biti dovoljno točna za većinu praktičnih primjena. Osim toga, iz izraza (4.90) vidljivo da je metoda potpuno točna za polinome prvog stupnja, jer za $f''(\zeta) = 0$ dobijemo grešku $E = 0$. Slučaj u kojem je $b - a = 0$ nema smisla razmatrati, a za $n \to \infty$ svakako mora vrijediti $E \to 0$.

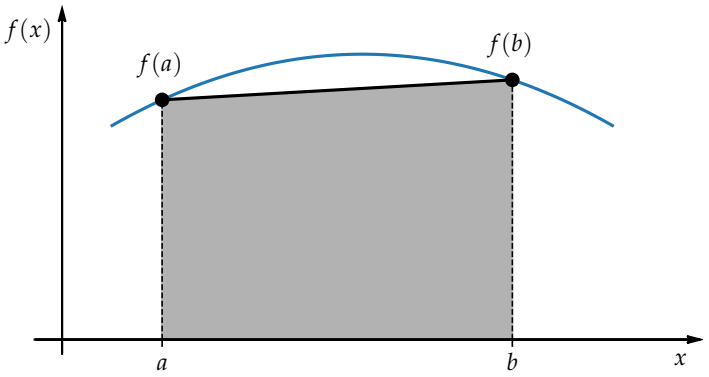
4.4.2 Trapezna formula

Numerička aproksimacija podintegralne funkcije pravcem:

$$f_n(x) = a_0 + a_1 x$$

rezultira najraširenijom metodom numeričke integracije – **trapeznom formulom**.

Ovdje se površina ispod pravca može izračunati koristeći izraz za površinu trapeza $A_t = (b-a)\frac{f(b)+f(a)}{2}$ (Slika 4.30).

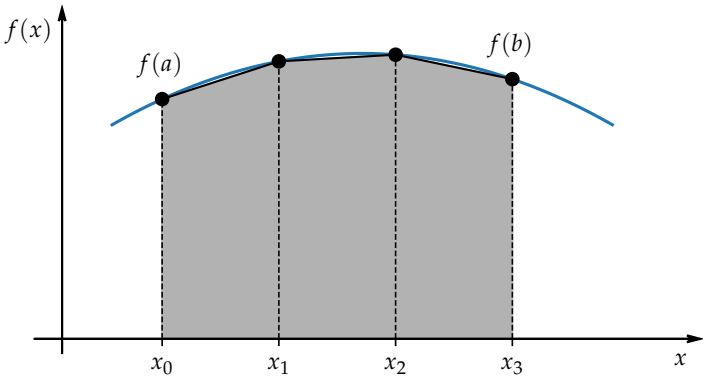


Slika 4.30
Integracija trapeznom formulom

Slijedi da je površina ispod pravca jednaka:

$$I = \int_a^b f(x)\mathrm{d}x \approx \frac{1}{2}(b-a)\left(f(a)+f(b)\right). \tag{4.91}$$

Iz Slike 4.30 jasno je vidljivo da bi mogli dobiti puno bolji rezultat ako bi interval podijelili na više manjih (Slika 4.31), integrirali segmente trapeznom formulom te ih na kraju zbrojili.



Slika 4.31
Integracija više podintervala trapeznom formulom

Podijelimo li tako interval $[a,b]$ na više manjih, označivši ih $x_0 = a$ i $x_n = b$, možemo napisati:

$$I = \int_a^b f(x)\mathrm{d}x \approx \int_{x_0}^{x_1} f(x)\mathrm{d}x + \int_{x_1}^{x_2} f(x)\mathrm{d}x + \ldots + \int_{x_{n-1}}^{x_n} f(x)\mathrm{d}x.$$

Ostaje nam odrediti $x_1 \ldots x_{n-1}$. Podijelimo li interval na n jednakih dijelova širine $h = (b-a)/n$, tada je $x_1 = x_0 + h$, $x_2 = x_1 + h$, ili $x_i = x_{i-1} + h$. Za svaki segment zasebno primijenimo trapeznu formulu (4.91) i možemo pisati:

$$I = \int_a^b f(x)\mathrm{d}x \approx h\frac{f(x_0)+f(x_1)}{2} + h\frac{f(x_1)+f(x_2)}{2} + \ldots h\frac{f(x_{n-1})+f(x_n)}{2}.$$

Presložimo li članove iz gornjeg izraza, dobit ćemo konačnu kompozitnu trapeznu formulu:

$$I = \int_a^b f(x)\mathrm{d}x \approx \frac{h}{2}\left(f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(x_n)\right), \tag{4.92}$$

ili ako h izrazimo preko ukupnog intervala, dobijemo

$$I = \int_a^b f(x)\mathrm{d}x \approx (b-a)\frac{1}{2n}\left(f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(x_n)\right), \tag{4.93}$$

Rudimentarna Python implementacija trapezne formule može se vidjeti u programskom kodu 4.10:

Python kod 4.10 Trapezna formula

```
import numpy as np

def f(t):
    # definicija podintegralne funkcije
    return np.sin(t)

a, b = 1, 2     # granice integriranja
n = 20          # broj intervala
h = (b - a) / n # širina intervala

x = np.linspace(a, b, n+1)  # n+1 točka za n intervala
y = f(x)
integral = h/2 * (y[0] + 2*np.sum(y[1:-1]) + y[-1])

print('Integral: ', integral)
print('Točan rezultat: ', np.cos(1) - np.cos(2))
```

Procjena greške trapezne formule se izražava izrazom

$$E = \frac{(b-a)^3}{12n^2} f''(\zeta). \tag{4.94}$$

Usporedimo li ovu grešku s greškom metode srednje točke (4.90), vidjet ćemo da je metoda srednje točke točnija. No, kako je razlika praktički zanemariva, u praksi se ipak preferira trapezna formula. Kao i metoda srednje točke, trapezna formula za polinome prvog stupnja daje točan rezultat (jer za $f''(\zeta) = 0$ vrijedi $E = 0$).

Broj intervala n u izrazima (4.90) i (4.94) zahtijeva posebnu pažnju. Možemo primijetiti da ako se broj intervala poveća dva puta, greška se smanjuje četiri puta. Dakle, ako za n_1 intervala izračunamo grešku metode E_1, za $n_2 = 4n_1$ intervala slijedi $E_2 = \frac{1}{16}E_1$.

Formalno, metode su drugog reda točnosti, što se izražava još i kao $O(h^2)$, jer je greška ovisna o veličini koraka h.

4.4.3 Simpsonove formule

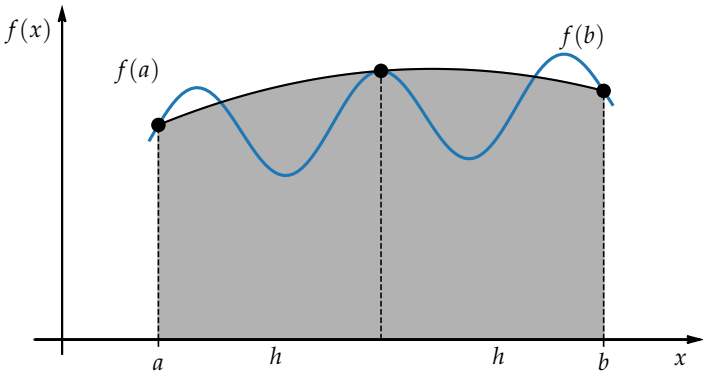
Zamijenimo li aproksimaciju podintegralne funkcije iz izraza (4.84) polinomom drugog stupnja, odnosno $f_n(x) = f_2(x) = a_0 + a_1 x + a_2 x^2$, tada možemo pisati:

$$I = \int_a^b f(x)dx \approx \int_a^b f_2(x)dx.$$

Drugim riječima, potrebno je izračunati integral polinoma drugog stupnja unutar granica $[a,b]$ (Slika 4.32).

Ostaje samo odrediti koeficijente polinoma, za što odabiremo tri ekvidistantne točke, tj. početak, sredinu i kraj intervala:

- $T_0 : a, f_2(a)$,
- $T_1 : (a+b)/2, f_2\left((a+b)/2\right)$ i
- $T_2 : b, f_2(b)$.



Slika 4.32 Integracija polinomom drugog stupnja

Umjesto rješavanja linearnog sustava od tri jednadžbe, dovoljno je upotrijebiti Lagrangeovu interpolaciju (poglavlje 4.2.1). Naime, uvedemo li točke $x_0 = a$, $x_1 = (b+a)/2$ i $x_2 = b$ te iskoristimo li izraz (4.40), možemo konstruirati interpolacijski polinom:

$$I \approx \int_{x_0}^{x_2} \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} f(x_0) + \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} f(x_1) + \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} f(x_2) \mathrm{d}x.$$

Nakon što definiramo $h = (x_2 - x_0)/2 = (x_1 - x_0)$ i provedemo algebarske manipulacije, dolazimo do konačnog izraza

$$I \approx \frac{h}{3}\left(f(x_0) + 4f(x_1) + f(x_2)\right), \tag{4.95}$$

koji se naziva **Simpsonova 1/3 formula**.

Kao i kod trapezne formule, moguće je veliki interval podijeliti na više manjih i provoditi integraciju po dijelovima. Odnosno,

$$I = \int_a^b f(x)\mathrm{d}x \approx \int_{x_0}^{x_2} f(x)\mathrm{d}x + \int_{x_2}^{x_4} f(x)\mathrm{d}x + \ldots + \int_{x_{n-2}}^{x_n} f(x)\mathrm{d}x.$$

Ostaje nam odrediti $x_1 \ldots x_{n-1}$. Podijelimo li interval na n jednakih dijelova širine $h = (b-a)/n$, tada je $x_1 = x_0 + h$, $x_2 = x_1 + h$, ili $x_i = x_{i-1} + h$. Za svaka dva segmenta primijenimo Simpsonovu 1/3 formulu (4.95) i možemo pisati:

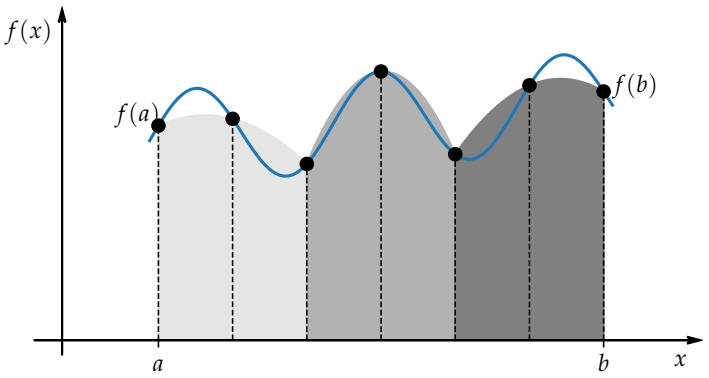
$$I = \int_a^b f(x)\mathrm{d}x \approx \frac{h}{3}\left(f(x_0) + 4f(x_1) + f(x_2)\right) + \frac{h}{3}\left(f(x_2) + 4f(x_3) + f(x_4)\right) + \ldots + \frac{h}{3}\left(f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)\right)$$

Presložimo li članove iz gornjeg izraza, dobit ćemo konačnu **kompozitnu Simpsonovu 1/3 formulu**:

$$I \approx \frac{h}{3}\left(f(x_0) + 4\sum_{i=1,3,5,\ldots}^{n-1} f(x_i) + 2\sum_{i=2,4,6,\ldots}^{n-2} f(x_i) + f(x_n)\right). \tag{4.96}$$

Neparne točke su srednji član iz Simpsonove formule (4.95), dok su parni članovi rubni te se oni dvaput pribrajaju. Ovdje vrijedi napomenuti da broj intervala n mora uvijek biti paran broj.

Slika 4.33 prikazuje kompozitnu integraciju domene podijeljene na šest intervala. Integracija po dijelovima se provodi za svaka dva intervala zasebno (na Slici označeni različitim nijansama sive boje).



Slika 4.33
Integracija više intervala Simpsonovom 1/3 formulom

Osnovna Python implementacija Simpsonove 1/3 formule može se vidjeti u programskom kodu 4.11:

Python kod 4.11 Simpsonova 1/3 formula

```
import numpy as np

def f(t):
    # definicija podintegralne funkcije
    return np.sin(t)

a, b = 1, 2     # granice integriranja
n = 20          # broj intervala
h = (b - a) / n # širina intervala

x = np.linspace(a, b, n+1)  # n+1 točka za n intervala
y = f(x)
integral = h/3 * (y[0] + 4*np.sum(y[1:-1:2]) + 2*np.sum(y[2:-2:2]) + y[-1])

print('Integral: ', integral)
print('Točan rezultat: ', np.cos(1) - np.cos(2))
```

Procjena greške Simpsonove 1/3 formula izražava se kao:

$$E = \frac{(b-a)^5}{2880}\overline{f}^{(4)}(\zeta) \tag{4.97}$$

za dva intervala, odnosno

$$E = \frac{(b-a)^5}{180n^4}\overline{f}^{(4)}(\zeta), \tag{4.98}$$

za n intervala, gdje je $\overline{f}^{(4)}$ prosječna četvrta derivacija na promatranom intervalu.

Već samo površnom analizom gornjeg izraza možemo doći do zaključka da je metoda točna i za polinome trećeg stupnja, iako podintegralnu funkciju aproksimiramo polinomom drugog stupnja. Nadalje, povećanje broja intervala dva puta rezultira smanjenjem greške šesnaest puta.

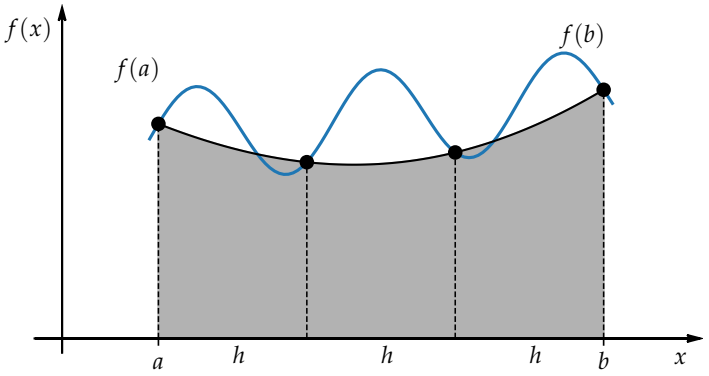
Umjesto aproksimacije podintegralne funkcije iz izraza (4.84) polinomom drugog stupnja, možemo koristiti polinom trećeg stupnja: $f_n(x) = f_3(x) = a_0 + a_1x + a_2x^2 + +a_3x^3$. Tada vrijedi:

$$I = \int_a^b f(x)dx \approx \int_a^b f_3(x)dx.$$

Upotrijebimo li istu tehniku kao i kod Simpsonove 1/3 formule, doći ćemo do izraza za **Simpsonovu 3/8** formulu:

$$I \approx \frac{3}{8}h\left(f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)\right). \tag{4.99}$$

Izraz (4.99) koristi četiri interpolacijske točke, ili tri intervala, što je za jedan više u odnosu na u odnosu na Simpsonovu 1/3 formulu (Slika 4.34).



Slika 4.34
Integracija tri intervala Simpsonovom 3/8 formulom

I ovdje je moguće podijeliti veliki interval na više manjih i provesti integraciju po dijelovima. Odnosno,

$$I = \int_a^b f(x)\mathrm{d}x \approx \int_{x_0}^{x_3} f(x)\mathrm{d}x + \int_{x_3}^{x_6} f(x)\mathrm{d}x + \ldots + \int_{x_{n-3}}^{x_n} f(x)\mathrm{d}x.$$

Ostaje nam odrediti $x_1 \ldots x_{n-1}$. Podijelimo li interval na n jednakih dijelova širine $h = (b-a)/n$, tada je $x_1 = x_0 + h$, $x_2 = x_1 + h$, ili $x_i = x_{i-1} + h$. Za svaka tri segmenta primijenimo Simpsonovu 3/8 formulu (4.99) i možemo pisati:

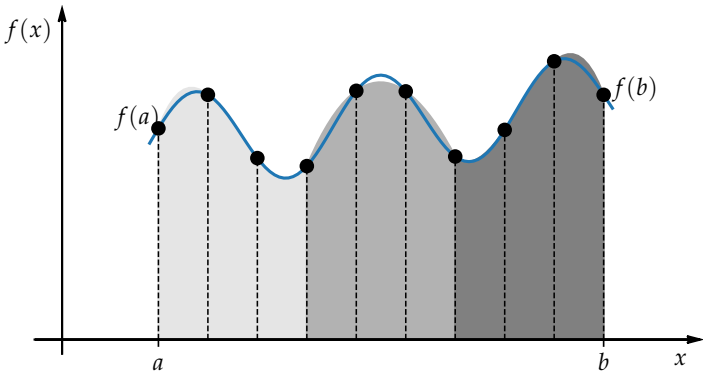
$$\begin{aligned} I = \int_a^b f(x)\mathrm{d}x \approx & \frac{3}{8}h\left(f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)\right) + \\ & \frac{3}{8}h\left(f(x_3) + 3f(x_4) + 3f(x_5) + f(x_6)\right) + \ldots \\ & + \frac{3}{8}h\left(f(x_{n-3}) + 3f(x_{n-2}) + 3f(x_{n-1}) + f(x_n)\right) \end{aligned}$$

Presložimo li članove iz gornjeg izraza, dobit ćemo konačnu **kompozitnu Simpsonovu 3/8 formulu**:

$$I \approx \frac{3}{8}h\left(f(x_0) + 3\sum_{i=1,4,7,\ldots}^{n-2} f(x_i) + 3\sum_{i=2,5,8,\ldots}^{n-1} f(x_i) + 2\sum_{i=3,6,9,\ldots}^{n-3} f(x_i) + f(x_n)\right). \tag{4.100}$$

Ovdje vrijedi napomenuti da broj intervala n mora uvijek biti djeljiv s brojem 3.

Slika 4.35 prikazuje kompozitnu integraciju domene podijeljene na devet intervala. Integracija po dijelovima se provodi za svaka tri intervala zasebno (na slici označeni različitim nijansama sive boje).



Slika 4.35
Integracija više intervala Simpsonovom 3/8 formulom

Greška Simpsonove 3/8 formule dana je izrazom:

$$E = \frac{(b-a)^5}{6480} f^{(4)}(\zeta),$$

iz čega je vidljivo da je točnost u odnosu na Simpsonovu 1/3 formulu neznatno bolja (konkretno 2,25 puta), ali zahtijeva jednu evaluaciju funkcije više, dok je apsolutno točna za polinome trećeg stupnja, isto kao i Simpsonova 1/3 metoda. Iz tih razloga šira uporaba Simpsonove 3/8 formule nije opravdana.

Napomena
Vidjeli smo da Simpsonova 1/3 i 3/8 metoda zahtijevaju da područje integracije bude podijeljeno na paran broj intervala odnosno na broj intervala djeljiv sa tri, a da bi pokrile cijelo područje integracije. To naravno u praksi ne predstavlja nikakvo realno ograničenje, jer se broj intervala n ionako određuje automatski i može se uvijek odrediti na način da bude prilagođen metodi koja se koristi.

4.4.4 Rombergova integracija

Metoda kojom je moguće povećati točnost trapezne formule bez povećanja broja intervala, postiže se inteligentnom matematičkom manipulacijom, koja spada u širu grupu tzv. Richardsonovih ekstrapolacija. Osnovna ideja je smanjiti grešku koristeći vrijednost integrala za dva različita broja intervala.

Probajmo točnije izračunati vrijednost integrala koristeći dobivene vrijednosti za dva različita koraka, h_1 i h_2. Sjetimo se da zbroj rezultata integracije trapeznom formulom (4.92) i njene greške (4.94) daje točno rješenje, odnosno možemo pisati:

$$I = I_T(h) + E(h),$$

gdje je I točna vrijednost integrala, $I_T(h)$ vrijednost integrala dobivenog kompozitnom trapeznom formulom širine intervala $h = (b-a)/n$ i $E(h)$ njena pripadajuća greška. Ako izračunamo integral sa dvije različite širine intervala, h_1 i h_2, tada vrijedi

$$I(h_1) + E(h_1) = I(h_2) + E(h_2). \tag{4.101}$$

Grešku trapezne formule iz izraza (4.94) možemo izraziti i preko veličine koraka h:

$$E = -\frac{b-a}{12} h^2 f''(\zeta),$$

a kako je f'' konstantna neovisno o veličini koraka h, onda vrijedi

$$\frac{E(h_1)}{E(h_2)} \approx \frac{h_1^2}{h_2^2}.$$

Prebacimo $E(h_2)$ na desnu stranu:

$$E(h_1) \approx E(h_2) \left(\frac{h_1}{h_2}\right)^2$$

i uvrstimo u izraz (4.101), dobit ćemo:

$$I(h_1) + E(h_2) \left(\frac{h_1}{h_2}\right)^2 \approx I(h_2) + E(h_2) \tag{4.102}$$

Iz gornjeg izraza lako možemo dobiti procjenu greške na temelju izračuna integrala $I(h_1)$ i $I(h_2)$

$$E(h_2) \approx \frac{I(h_1) - I(h_2)}{1 - (h_1/h_2)^2} \tag{4.103}$$

Uvrstimo li $E(h_2)$ iz (4.103) u

$$I = I(h_2) + E(h_2),$$

dobili smo točniji izraz:

$$I \approx I(h_2) + \frac{1}{(h_1/h_2)^2 + 1} \left(I(h_2) - I(h_1)\right) \tag{4.104}$$

Može se pokazati da je greška izraza (4.104) četvrtog reda ($O(h^4)$), i to kombiniranjem dvije metode niže točnosti ($O(h^2)$).

Ako odlučimo koristiti $h_2 = h_1/2$ dobijemo vrlo praktičan specijalni slučaj metode, u kojem prvo provodimo integraciju na izabranom nizu točaka (tj. sa korakom h_2), a zatim ponovno na istom nizu, ali ovaj put uz preskakanje svake druge točke (tj. s duplo većim korakom $h_1 = 2h_2$). U tom slučaju se izraz (4.104) pretvara u

$$I \approx I(h_2) + \frac{1}{2^2 - 1} \left(I(h_2) - I(h_1)\right),$$

odnosno

$$I \approx \frac{4}{3} I(h_2) - \frac{1}{3} I(h_1) \tag{4.105}$$

Ova metoda je iznimno popularna, naročito u situacijama kada je evaluacija podintegralne funkcije skupa, jer je dovoljno evaluirati funkciju samo na gušćoj podjeli, nakon čega koristimo iste točke dvaput, za $I(h_2)$ i za $I(h_1)$.

Jednostavna Python implementacija Rombergove integracije dana je u programskom kodu 4.12:

Python kod 4.12 Rombergova integracija

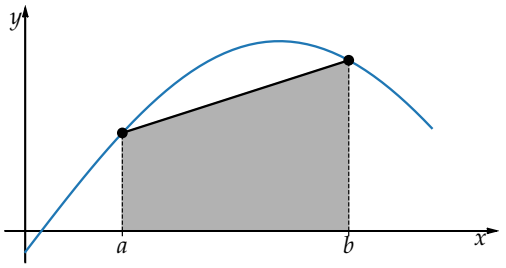
```
import numpy as np

def f(t):
```

```
    # definicija podintegralne funkcije
    return np.sin(t)

def trapezna(h, y):
    return h/2 * (y[0] + 2*np.sum(y[1:-1]) + y[-1])

a, b = 1, 2         # granice integriranja
n = 20              # broj intervala
h2 = (b - a) / n    # širina koraka
h1 = 2 * h2         # dvostruko veća širina koraka

x = np.linspace(a, b, n+1)  # n+1 točka za n intervala
y_gusta = f(x)
i_h1 = trapezna(h1, y_gusta[::2])
i_h2 = trapezna(h2, y_gusta)
integral = 4 / 3 * i_h2 - 1/3*i_h1

print('Integral za h_1: ', i_h1)
print('Integral za h_2: ', i_h2)
print('Romberg integral: ', integral)
print('Točan rezultat: ', np.cos(1) - np.cos(2))
```

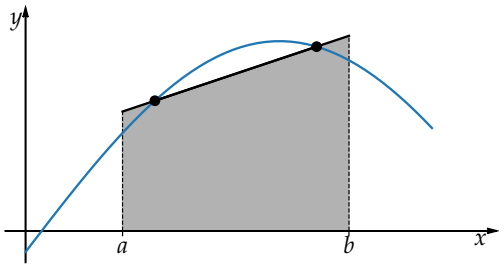
4.4.5 Gaussova integracija

Trapezna formula (4.91) se zasniva na izračunavanju površine trapeza koji spajaju krajnje točke integracijskog intervala. Kako pravac mora proći kroz krajnje točke intervala, takva evaluacija može rezultirati velikom greškom (Slika 4.36a).

Moguće je znatno poboljšati procjenu integrala ako umjesto krajnjih točaka intervala pažljivo odaberemo točke unutar intervala i izračunamo površinu ispod krivulje (Slika 4.36b). **Gaussova integracija** je tek jedna od tehnika koje iskorištavaju ovu strategiju.



(a) Trapezna formula: površina ispod pravca koji prolazi kroz krajnje točke intervala



(b) Gaussova integracija: površina ispod pravca koji prolazi kroz dvije pažljivo odabrane točke unutar intervala

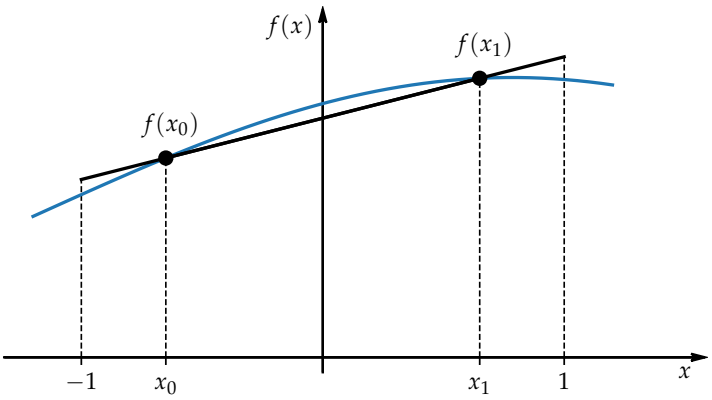
Slika 4.36 Ideja Gaussove integracije

Općenito, Gaussove integracijske formule imaju oblik

$$I = \int_a^b f(x) \approx \sum_{i=0}^{n-1} w_i f(x_i) \tag{4.106}$$

gdje su w_i težinski koeficijenti, a točke integracije x_i nisu unaprijed poznate, već se izračunavaju tako da greška takve formule bude najmanja.

Ako se ograničimo na integracijsku formulu sa dvije točke i postavimo granice integriranja na intervalu $[-1,1]$ (Slika 4.37), tada izraz (4.106) postaje



Slika 4.37
Gaussova integracija kroz dvije točke, x_0 i x_1

$$I = \int_{-1}^{1} f(x) \approx \sum_{i=0}^{1} w_i f(x_i) = w_0 f(x_0) + w_1 f(x_1) \tag{4.107}$$

Potrebno je odrediti četiri nepoznanice, w_0, x_0, w_1 i x_1. Također, postavljamo uvjet prema kojem želimo točan izračun integrala na već spomenutom intervalu $[-1,1]$ za sljedeće funkcije: $f(x) = 1$, $f(x) = x$, $f(x) = x^2$ i $f(x) = x^3$.

Odnosno, potrebno je riješiti sustav od četiri jednadžbe:

$$\begin{aligned}
w_0 f(x_0) + w_1 f(x_1) &= \int_{-1}^{1} 1 dx = 2 \\
w_0 f(x_0) + w_1 f(x_1) &= \int_{-1}^{1} x dx = 0 \\
w_0 f(x_0) + w_1 f(x_1) &= \int_{-1}^{1} x^2 dx = \frac{2}{3} \\
w_0 f(x_0) + w_1 f(x_1) &= \int_{-1}^{1} x^3 dx = 0
\end{aligned} \tag{4.108}$$

Rješenjem gurnjeg sustava (4.108) se tada lako dobiju koeficijenti

$$\begin{aligned}
w_0 &= w_1 = 1 \\
x_0 &= -x_1 = -\frac{1}{\sqrt{3}},
\end{aligned}$$

što rezultira konačnom **Gaussovom integracijskom formulom kroz dvije točke** (*Gauss two point formula*):

$$I \approx f(-\frac{1}{\sqrt{3}}) + f(\frac{1}{\sqrt{3}}). \tag{4.109}$$

Evaluacijom funkcije unutar intervala smo time uspjeli postići poboljšanje u točnosti. Naime, Newton-Cotesove metode (trapezna, Simpsonova, itd.) su točne samo za polinome $(n-1)$ stupnja, a Gaussove formule su točne za polinome stupnja $(2n-1)$. Analiza greške pokazuje da je

$$E_t = \frac{2^{2n+1}(n!)^4}{(2n+1)((2n)!)^3} f^{2n}(x), \quad x \in [-1,1] \tag{4.110}$$

Napomena
Metode ovog tipa se mogu primijeniti samo u slučajevima u kojima možemo izračunati vrijednost funkcije $f(x)$ u bilo kojoj točki, čime su takve metode neprimjenjive za tablično zadane funkcije.

Želimo li pak izračunati integral funkcije $f(x)$ na intervalu $[a,b]$, potrebno je provesti preslikavanje iz zadanog intervala na kanonski interval $[-1,1]$):

$$\int_a^b f(x)dx = \int_{-1}^{1} g(\zeta)d\zeta.$$

Upotrijebimo linearnu transformaciju:

$$x = c_0 + c_1\zeta, \tag{4.111}$$

gdje su koeficijenti c_0 i c_1 nepoznanice koje tek treba odrediti. Srećom, možemo iskoristiti sljedeće činjenice:

- za $x = a$, $\zeta = -1$ i
- za $x = b$, $\zeta = 1$.

Uvrstimo li gore navedene uvjete u izraz (4.111), dobit ćemo sustav od dvije jednadžbe i dvije nepoznanice:

$$a = c_0 + c_1(-1)$$
$$b = c_0 + c_1(1),$$

čijim rješavanjem napokon dobijamo koeficijente:

$$c_0 = \frac{b+a}{2}$$
$$c_1 = \frac{b-a}{2}. \tag{4.112}$$

Time izraz (4.111) poprima konačan oblik

$$x = \frac{(b+a)+(b-a)\zeta}{2}, \tag{4.113}$$

iz čega slijedi

$$dx = \frac{b-a}{2}d\zeta. \tag{4.114}$$

Konačno, korištenjem izraza (4.113) i (4.114) lako je moguće provesti supstituciju i riješiti integral.

Kao i kod Newton-Cotesovih formula, povećanje točnosti može se postići subdivizijom osnovnog intervala integracije.

Primjer 4.1 Izračunati integral $f(x) = 0.2 + 25x - 200x^2 + 675x^3 - 900x^4 + 400x^5$ na intervalu od $[0,0.8]$

Odredimo granice intervala: $a = 0$, $b = 0.8$ i uvrstimo u izraze (4.113) i (4.114) čime dobijemo

$$x = 0.4 + 0.4\zeta$$
$$dx = 0.4d\zeta.$$

Oba izraza sada možemo uvrstiti u zadanu jednadžbu:

$$I = \int_0^{0.8} (0.2 + 25x - 200x^2 + 675x^3 - 900x^4 + 400x^5)dx$$

$$g(\zeta) = \int_{-1}^{1} \Big(0.2 + 25(0.4 + 0.4\zeta) - 200(0.4 + 0.4\zeta)^2 + \\ + 675(0.4 + 0.4\zeta)^3 - 900(0.4 + 0.4\zeta)^4 + 400(0.4 + 0.4\zeta)^5\Big)0.4d\zeta$$

Na kraju nam ostaje samo izračunati integral:

$$I \approx g(-\frac{1}{\sqrt{3}}) + g(\frac{1}{\sqrt{3}}) \approx 11.82257$$

Isto rješenje se može postići i korištenjem `SciPy` funkcije `fixed_quad`, što je prikazano u programskom kodu (4.13). ■

Python kod 4.13 Integracija dvostrukog integrala trapeznom formulom

```
from scipy.integrate import fixed_quad

def f(t):
    # definicija podintegralne funkcije
    return 0.2 + 25 * t - 200 * t ** 2 + 675 * t ** 3 - 900 * t ** 4 + 400 * t ** 5

a, b = 0, 0.8                          # granice integriranja
I_g, none = fixed_quad(f, a, b, n=2)   # funkcija, granice integracije i
                                       # red integracije

print('I_g: {}'.format(I_g))
```

4.4.6 Integracija tablično zadane funkcije

U inženjerskoj praksi je često potrebno izračunati određeni integral gdje je funkcija $f(x)$ zadana kao niz diskretnih točaka (Slika 4.28 s početka poglavlja). U tom slučaju možemo razlikovati dva različita scenarija: razmak između susjednih točke može biti jednolik ili nije moguće garantirati uniformnu udaljenost između točaka. Formalnije, može vrijediti

- $\Delta x = \Delta x_1 = \Delta x_2 = \Delta x_3 = \cdots = \Delta x_n = \text{const.}$ ili
- $\Delta x_1 \neq \Delta x_2 \neq \Delta x_3 \neq \cdots \neq \Delta x_n$.

U slučaju gdje razmak između točaka nije uniforman, nije moguće koristiti Newton-Cotesove formule u svom kompaktnom obliku (npr. izraz 4.93), već u ponešto izmijenjenom obliku. Trapezna formula bi tada poprimila oblik

$$I = \int_a^b f(x)\mathrm{d}x \approx \Delta x_0 \frac{f(x_0) + f(x_1)}{2} + \Delta x_1 \frac{f(x_1) + f(x_2)}{2} + \ldots \Delta x_{n-1} \frac{f(x_{n-1}) + f(x_n)}{2}.$$

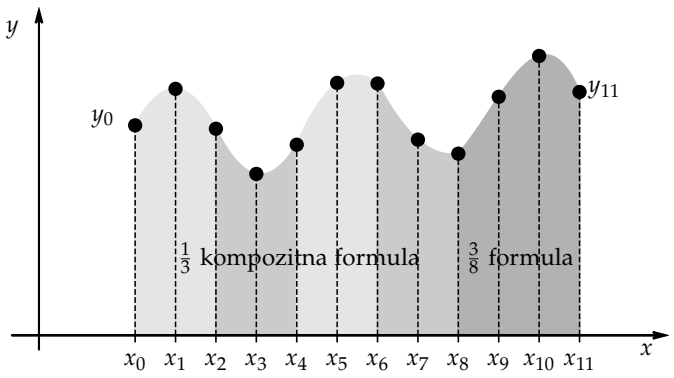
Analogno vrijedi za metode lijeve i desne točke.

U slučaju gdje vrijedi $\Delta x = \text{const.}$, možemo upotrijebiti Newton-Cotesove formule, gdje se može upotrijebiti trapezna formula (4.93), metoda lijeve (4.87) ili desne točke (4.88).

Želimo li pak povećati točnost izračunavanja integrala, možemo koristiti Rombergovu integraciju (4.105), uz ograničenje gdje broj intervala n mora biti paran broj. Isto ograničenje vrijedi i za Simpsonovu $1/3$ formulu (4.96), dok za Simpsonovu $3/8$ formulu (4.100), broj intervala n mora biti djeljiv s tri.

Kombiniranjem kompozitne Simpsonove $1/3$ formule i Simpsonove $3/8$ formule moguće je provesti numeričku integraciju proizvoljnog broja intervala. Slika 4.38 prikazuje 12 uniformnno raspoređenih točaka, čime je definirano ukupno $n = 11$ intervala.

U slučaju parnog broja intervala, koristi se isključivo kompozitna $1/3$ Simpsonova formula. U suprotnom, zadnja tri intervala se integriraju Simpsonovom $3/8$ formulom, a ostali kompozitnom Simpsonovom $1/3$ formulom.



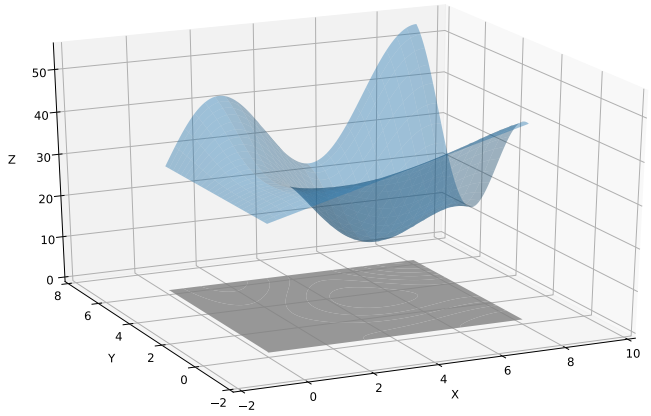
Slika 4.38
Kombiniranje Simpsonove $1/3$ i $3/8$ formule za proizvoljan broj tablično zadanih točaka

4.4.7 Višestruki integrali

Evaluacija višestrukih integrala

$$I = \int \cdots \int_V f(x_1, \ldots, x_k)\, dx_1 \ldots dx_k$$

se svodi na numeričku evaluaciju svakog jednostrukog integrala zasebno. Zbog jednostavnosti ćemo razmatrati samo dvostruki integral (Slika 4.39).



Slika 4.39
Funkcija $f(x,y) = 2y\sin x + 2x\cos y + 30$ za $x \in [0,8]$ i $y \in [0,6]$

Napišemo li dvostruki integral

$$I = \int_c^d \int_a^b f(x,y)dxdy = \int_c^d \left(\int_a^b f(x,y)dx \right) dy$$

te razdijelimo li interval $[c,d]$ na n_y jednakih dijelova veličine $h_y = (d-c)/n_y$ i interval $[a,b]$ na n_x jednakih dijelova veličine $h_x = (b-a)/n_x$, imamo sve potrebno za numeričko izračunavanje volumena ispod plohe. Za svaku diskretiziranu y vrijednost potrebno je numerički izračunati unutarnji integral $\int_a^b f(x,y)dx$. Tako dobivene vrijednosti je tada potrebno integrirati duž y osi unutar već definiranih granica $[c,d]$.

Rudimentarna Python implementacija rješavača dvostrukog integrala može se vidjeti u programskom kodu 4.14.

Python kod 4.14 Integracija dvostrukog integrala trapeznom formulom

```
import numpy as np

def f(x, y):
    return 2 * y * np.sin(x) + 2 * x * np.cos(y) + 30

def trapezna(h, y):
    return h/2 * (y[0] + 2*np.sum(y[1:-1]) + y[-1])

a, b = 0, 8         # granice integriranja
c, d = 0, 6         # granice integriranja
nx, ny = 50, 50     # broj intervala
hx = (b-a)/nx
hy = (d-c)/ny

xd = np.linspace(a, b, nx+1)
yd = np.linspace(c, d, ny+1)
X, Y = np.meshgrid(xd, yd)
F = f(X, Y)

int_x = np.zeros_like(yd)
# unutarnji integral za svaki yd
for i in range(yd.shape[0]):
    int_x[i] = trapezna(hx, F[i])

# vanjski integral
int_y = trapezna(hy, int_x)

print('Integral: ', int_y)
print('Točan rezultat: ', 4*(369+16*np.sin(6)-9*np.cos(8)))
```

Broj evaluacija funkcije višestrukog integrala značajno raste, odnosno, ako za svaku dimenziju odaberemo diskretizaciju od 100 točaka, ukupni broj evaluacija funkcije iznosi 100^N, gdje je N broj dimenzija višestrukog integrala. U tom slučaju je potrebno koristiti metode višeg reda točnosti (npr. Simpsonove formule, Rombergova integracija), a često se pribjegava i drugim, naprednijim tehnikama (adaptivna ili Monte-Carlo integracija).

4.4.8 Integriranje pomoću SciPy modula

SciPy modul nudi više metoda za numeričko integriranje u podmodulu `scipy.integrate`. Ovdje su navedene samo neke od tih metoda i to u svom osnovnom obliku, bez objašnjavanja dodatnih argumenata.

Implementirane su trapezna, Rombergova i Simpsonova 1/3 formula funkcijama `trapezoid`, `romb` i `simpson`:

```
scipy.integrate.trapezoid(y, x)
scipy.integrate.romb(y, h)
scipy.integrate.simpson(y, x)
```

gdje je `y` lista evaluiranih vrijednosti funkcije za zadane vrijednosti `x`, a `h` u Rombergovoj integraciji predstavlja korak integracije (Δx).

Napomena
Broj točaka u Rombergovoj integraciji mora biti $n = 2^k + 1$, gdje je k prirodni broj.

Primjer korištenja navedenih `SciPy` funkcija dan je u programskom kodu 4.15.

Python kod 4.15 Integracija integrala funkcijama iz `scipy.integrate` podmodula

```
import numpy as np
from scipy.integrate import trapezoid, romb, simpson

def f(t):
    # definicija podintegralne funkcije
    return np.sin(t)

a, b = 1, 2     # granice integriranja
n = 17          # broj točaka

x, h = np.linspace(a, b, n, retstep=True)
y = f(x)

I_t = trapezoid(y, x)
I_r = romb(y, h)
I_s = simpson(y, x)

print('I_t: ', I_t)
print('I_r: ', I_r)
print('I_s: ', I_s)
```

Učinkovitiji numerički izračun određenog integrala moguće je provesti upotrebom adaptivne integracije pomoću funkcije `quad`:

Python kod 4.16 Integracija `quad` funkcijom

```
import numpy as np
from scipy.integrate import quad

def f(t):
    # definicija podintegralne funkcije
    return np.sin(t)

a, b = 1, 2 # granice integriranja
I_q, err = quad(f,a, b)

print(f'I_q: {I_q}, err: {err}')
```

4.5 Rješavanje sustava linearnih jednadžbi

Siniša Družeta
Jerko Škifić

Prilikom računalnog modeliranja prirodnih procesa, tehničkih sustava i sl., često se matematički problem koji se rješava svodi na problem rješavanja velikog broja linearnih jednadžbi.

Sustav linearnih jednadžbi s jednakim broj jednadžbi i nepoznanica možemo zapisati ovako:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots a_{2n}x_n &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \ldots a_{3n}x_n &= b_3 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \ldots a_{nn}x_n &= b_n \end{aligned} \tag{4.115}$$

ili kraće, kao matričnu jednadžbu:

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{4.116}$$

gdje su matrica sustava $\mathbf{A}$, vektor nepoznanica $\mathbf{x}$ i vektor desne strane (ili vektor slobodnih članova) $\mathbf{b}$ zadani kako slijedi:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \ldots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \ldots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \ldots & a_{3n} \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \ldots & a_{nn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix}. \tag{4.117}$$

Podrazumijeva se da su matrica $\mathbf{A}$ i vektor $\mathbf{b}$ zadani, a vektor nepoznanica $\mathbf{x}$ je nepoznat i treba ga odrediti. Pri tome treba napomenuti da linearni sustav jednadžbi ne mora nužno imati jedinstveno rješenje ili imati rješenje uopće. Ilustracija mogućih slučajeva rješivosti sustava s tri jednadžbe i tri nepoznanice dana je na Slici 4.40.

Slaba uvjetovanost

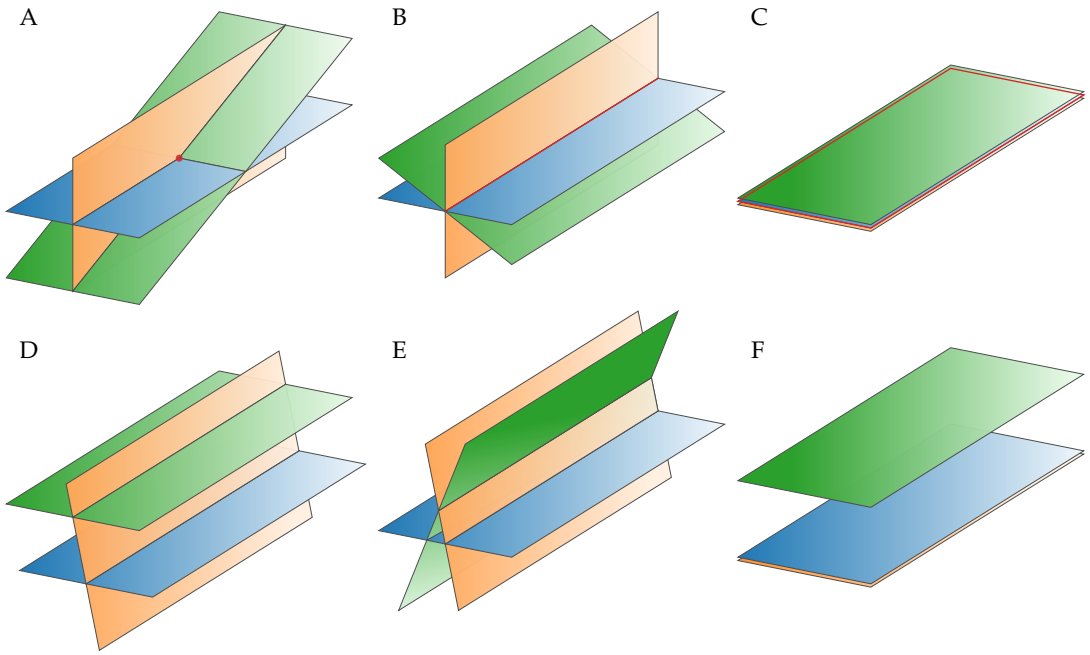
Kod nekih sustava jednadžbi mala varijacija matrice sustava $\mathbf{A}$ može uzrokovati velike promjene u rješenju $\mathbf{x}$. Takvi sustavi jednadžbi zovu se **slabo uvjetovani** (*ill conditioned*) sustavi.

Promotrimo jedan primjer. Sustav dvije jednadžbe sa dvije nepoznanice:

$$\begin{aligned} x_1 + 2x_2 &= 10 \\ 1.1x_1 + 2x_2 &= 10.4 \end{aligned} \tag{4.118}$$

ima rezultat $\mathbf{x} = [4 \;\; 3]^{\mathrm{T}}$. Promijenimo li malo vrijednost prvog koeficijenta druge jednadžbe, recimo sa $a_{21} = 1.1$ na $\tilde{a}_{21} = 1.05$, sustav se mijenja u:

$$\begin{aligned} x_1 + 2x_2 &= 10 \\ 1.05x_1 + 2x_2 &= 10.4, \end{aligned} \tag{4.119}$$



Slika 4.40 Sustav od tri linearne jednadžbe s tri nepoznanice prikazan ravninama u prostoru. Sustav ima jedinstveno riješenje: točka (A). Sustav ima beskonačno riješenja: pravac (B) ili ravnina (C). Sustav nema riješenja: (D), (E) i (F).

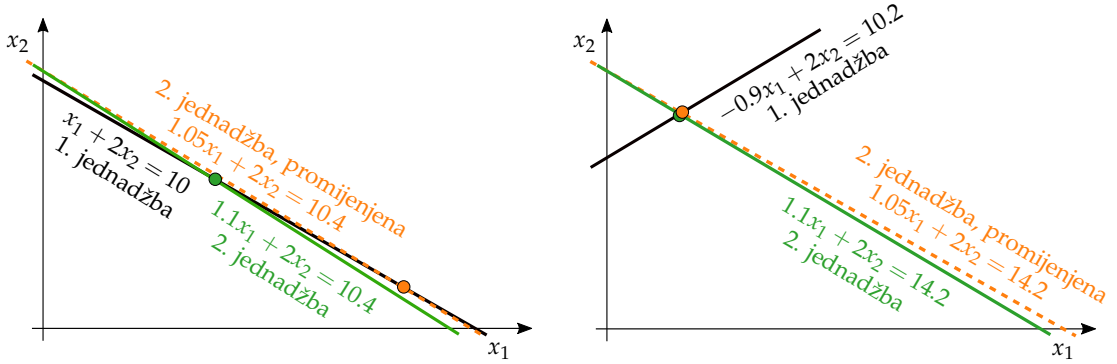
a njegovo rješenje postaje $\tilde{\mathbf{x}} = [8 \;\; 1]^{\mathrm{T}}$!

Kao suprotan primjer, uzmimo naizgled sličan sustav:

$$\begin{aligned} -0.9x_1 + 2x_2 &= 10.2 \\ 1.1x_1 + 2x_2 &= 14.2, \end{aligned} \tag{4.120}$$

koji ima rješenje $\mathbf{x} = [2 \;\; 6]^{\mathrm{T}}$. Promijenimo li mu drugu jednadžbu na isti način kao u prethodnom primjeru ($a_{21} = 1.1$ postaje $\tilde{a}_{21} = 1.05$), rješenje se mijenja u $\tilde{\mathbf{x}} = [2.051 \;\; 6.023]^{\mathrm{T}}$, što svakako nije dramatična promjena.

U geometrijskom smislu slaba uvjetovanost znači da su pravci prve i druge jednadžbe skoro paralelni, pa i mala nesigurnost u koeficijentima sustava ili netočnost u izračunu može uzrokovati velike promjene u rezultatu, tj. izračunatom sjecištu, što kod dobro uvjetovanog sustava nije slučaj (Slika 4.41).



Slika 4.41 Grafički prikaz slabo uvjetovanog sustava (lijevo) i dobro uvjetovanog sustava (desno) dvije jednadžbe s dvije nepoznanice

Napomena
S obzirom da sustave linearnih jednadžbi rješavamo na računalu, nestabilnost u koeficijentima sustava ne mora biti samo posljedica nesigurnosti u procijenjenim parametrima, nesavršenog prikupljanja podataka ili grešaka u prethodnim proračunima, već i greške zaokruživanja brojeva na računalu odnosno samog ograničenja računala u smislu točnosti zapisa decimalnih brojeva. Stoga nam dobra uvjetovanost sustava daje dodatnu garanciju u ispravnost računalom dobivenog rješenja.

Slabo uvjetovane sustave možemo prepoznati po tome što su gotovo singularni, tj. $det(A) \approx 0$. No, bolja mjera za slabu uvjetovanost je tzv. **kondicija matrice**:

$$\text{Cond}(\mathbf{A}) = ||\mathbf{A}|| \cdot ||\mathbf{A}^{-1}||, \tag{4.121}$$

gdje je

$$||\mathbf{A}|| = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}^2} \tag{4.122}$$

norma matrice. Što je veća kondicija matrice sustava, to je sustav slabije uvjetovan.

Drugi veliki problem kod računalnih programa za rješavanje sustava linearnih jednadžbi je memorija. U primjeni su vrlo česte matrice koeficijenata izuzetno velikih dimenzija; npr. postupak koji izračunava jednu jednadžbu na kvadratnoj numeričkoj mreži od samo 100×100 točaka podrazumijeva izračunavanje $100^2 = 10^4$ jednadžbi, što daje matricu sustava $10^4 \times 10^4$ sa ukupno 10^8 elemenata! S druge strane, većina tih koeficijenata je često jednaka nuli, odnosno takve matrice su rijetke (eng. *sparse*), pa u tom slučaju nije racionalno zauzeti memoriju za sve članove sustava, već se pamte samo elementi koji su različiti od nule (i njihove pozicije). Stoga za rijetke matrice postoje i posebni rješavači, kao i programski alati za rad s njima, no njima se ovdje nećemo baviti.

4.5.1 Gaussova eliminacija

Tipični način rješavanja sustava linearnih jednadžbi je upotrebom postupka koji se zove Gaussova eliminacija. Njen cilj je pretvoriti postojeći sustav jednadžbi u drugi sustav, kojem je matrica sustava gornje-trokutasta (i koji naravno daje isto rješenje).

Drugim riječima, provodimo transformaciju

$$\mathbf{A}\mathbf{x} = \mathbf{b} \quad \rightarrow \quad \mathbf{A}^*\,\mathbf{x} = \mathbf{b}^* \tag{4.123}$$

gdje su

$$\mathbf{A}^* = \begin{bmatrix} a_{11}^* & a_{12}^* & a_{13}^* & \dots & a_{1n}^* \\ & a_{22}^* & a_{23}^* & \dots & a_{2n}^* \\ & & a_{33}^* & \dots & a_{3n}^* \\ & & & \dots & \vdots \\ & & & \dots & a_{nn}^* \end{bmatrix}, \quad \mathbf{b}^* = \begin{bmatrix} b_1^* \\ b_2^* \\ b_3^* \\ \vdots \\ b_n^* \end{bmatrix}. \tag{4.124}$$

Postupak Gaussove eliminacije provodi se unaprijed (od prve jednadžbe prema posljednjoj) i možemo ga za primjer pokazati na sustavu tri jednadžbe sa tri nepoznanice:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned} \tag{4.125}$$

Prvu jednadžbu pomnožimo multiplikatorom $m_{21} = a_{21}/a_{11}$ (koeficijent na glavnoj dijagonali a_{11} se naziva *pivot* koeficijentom) čime se dobiva:

$$a_{21}x_1 + \frac{a_{21}}{a_{11}}a_{12}x_2 + \frac{a_{21}}{a_{11}}a_{13}x_3 = \frac{a_{21}}{a_{11}}b_1, \tag{4.126}$$

nakon čega je oduzmemo od druge jednadžbe:

$$(a_{21} - a_{21})x_1 + (a_{22} - \frac{a_{21}}{a_{11}}a_{12})x_2 + (a_{23} - \frac{a_{21}}{a_{11}}a_{13})x_3 = b_2 - \frac{a_{21}}{a_{11}}b_1 \tag{4.127}$$

čime druga jednadžba sad efektivno poprima novi oblik, u kojemu je prva nepoznanica eliminirana. Time dobijemo sustav koji izgleda ovako:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a'_{22}x_2 + a'_{23}x_3 &= b'_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3, \end{aligned} \tag{4.128}$$

gdje je:

$$\begin{aligned} a'_{22} &= a_{22} - \frac{a_{21}}{a_{11}}a_{12}, \\ a'_{23} &= a_{23} - \frac{a_{21}}{a_{11}}a_{13}, \\ b'_2 &= b_2 - \frac{a_{21}}{a_{11}}b_1. \end{aligned} \tag{4.129}$$

Ovaj postupak ponavljamo i na trećoj jednadžbi odnosno prvu jednadžbu sada pomnožimo s a_{31}/a_{11} i oduzmemo je od treće jednadžbe, čime dobijemo sustav:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a'_{22}x_2 + a'_{23}x_3 &= b'_2 \\ a'_{32}x_2 + a'_{33}x_3 &= b'_3, \end{aligned} \tag{4.130}$$

gdje je:

$$a'_{32} = a_{32} - \frac{a_{31}}{a_{11}}a_{12}, \tag{4.131}$$

$$a'_{33} = a_{33} - \frac{a_{31}}{a_{11}}a_{13}, \tag{4.132}$$

$$b'_3 = b_3 - \frac{a_{31}}{a_{11}}b_1. \tag{4.133}$$

U drugom koraku postupka uzmemo drugu jednadžbu, pomnožimo je s a'_{32}/a'_{22} i oduzmemo je od treće jednadžbe, čime provodimo transformaciju treće jednadžbe i dobijemo sustav:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a'_{22}x_2 + a'_{23}x_3 &= b'_2 \\ a''_{33}x_3 &= b''_3, \end{aligned} \tag{4.134}$$

gdje je:

$$a''_{33} = a'_{33} - \frac{a'_{32}}{a'_{22}}a'_{23}, \tag{4.135}$$

$$b''_3 = b'_3 - \frac{a'_{32}}{a'_{22}}b'_2. \tag{4.136}$$

Ovakav postupak promjene a_{ij} i b_i zove se **eliminacija unaprijed** i općenito se može zapisati na sljedeći način:

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}} a_{kj}^{(k-1)}, \qquad i = k+1,\ldots,n, \qquad j = k+1,\ldots,n,$$
$$b_i^{(k)} = b_i^{(k-1)} - \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}} b_k^{(k-1)}, \qquad i = k+1,\ldots,n, \tag{4.137}$$

gdje je n broj redaka matrice, a korak $k = 1,2,\ldots,n-1$. Element $a_{kk}^{(k-1)}$ iz gornjeg izraza (4.137) naziva se *pivot* koeficijentoma, a kvocijenti $m_{ik} = \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}}$ nazivaju se *multiplikatorima*.

Na ovaj način u konačnici dobijemo gornje-trokutastu matricu sustava $\mathbf{A}^*$. Uz korištenje zapisa $[\mathbf{A} \mid \mathbf{b}]$, cijeli postupak može se zapisati ovako:

$$[\mathbf{A} \mid \mathbf{b}] = \left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array}\right] \text{ početni izgled sustava} \tag{4.138}$$

$$\left[\mathbf{A}' \mid \mathbf{b}'\right] = \left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ 0 & a'_{22} & a'_{23} & b'_2 \\ 0 & a'_{32} & a'_{33} & b'_3 \end{array}\right] \text{ nakon 1. koraka} \tag{4.139}$$

$$\left[\mathbf{A}'' \mid \mathbf{b}''\right] = \left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ 0 & a'_{22} & a'_{23} & b'_2 \\ 0 & 0 & a''_{33} & b''_3 \end{array}\right] = [\mathbf{A}^* \mid \mathbf{b}^*] \quad \begin{array}{l}\text{nakon 2. koraka,} \\ \text{tj. na kraju postupka}\end{array} \tag{4.140}$$

Dakle, općenito govoreći, u Gaussovoj eliminaciji se prvu jednadžbu koristi za eliminaciju prve nepoznanice iz druge i svih nižih jednadžbi, drugu jednadžbu se zatim koristi za eliminaciju druge nepoznanice iz treće i svih nižih jednadžbi i tako dalje, sve do predposljednje jednadžbe koju se koristi za eliminaciju predposljednje nepoznanice iz posljednje jednadžbe. Ovakvim postupkom dobijemo gornje-trokutastu matricu, na kojoj je lako provesti postupak rješavanja sustava linearnih jednadžbi.

Sam postupak dobivanja rješenja zove se **supstitucija** i provodi se **unatrag**, od posljednje jednadžbe prema prvoj. Naime, pošto je matrica sustava nakon provedene eliminacije sada gornje-trokutasta:

$$\left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ & a'_{22} & a'_{23} & b'_2 \\ & & a''_{33} & b''_3 \end{array}\right], \tag{4.141}$$

posljednja jednadžba je zapravo trivijalna jednadžba s jednom nepoznanicom (x_3) i kao takvu možemo je direktno riješiti:

$$x_3 = b''_3 / a''_{33}. \tag{4.142}$$

Rješenje x_3 dobiveno iz treće jednadžbe možemo uvrstiti (supstituirati) u drugu jednadžbu, čime nam druga jednadžba postaje jednadžba s jednom nepoznanicom (x_2), pa možemo riješiti i nju:

$$x_2 = (b_2' - a_{23}' x_3)/a_{22}'. \tag{4.143}$$

Konačno, sada možemo x_2 i x_3 uvrstiti u prvu jednadžbu da dobijemo i posljednje rješenje x_1:

$$x_1 = (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11}. \tag{4.144}$$

Općenito postupak supstitucije možemo zapisati ovako:

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}} \tag{4.145}$$

$$x_i = \frac{b_i^{(i-1)} - \sum_{j=i+1}^{n} a_{ij}^{(i-1)} x_j}{a_{ii}^{(i-1)}} \qquad \text{za } i = n-1,\ n-2,\ \ldots,\ 1. \tag{4.146}$$

Da rezimiramo, metoda Gaussove eliminacije sastoji se od dviju faza: **eliminacije unaprijed** i **supstitucije unatrag**.

Programska implementacija Gaussove eliminacije za rješavanje sustava linearnih jednadžbi koji je definiran u vanjskim tekstualnim datotekama dana je u nastavku (Python kod 4.17)

Python kod 4.17 Gaussova eliminacija

```
"""
format ulaznih datoteka:

A.txt:
a11 a12 a13 ... a1n
a21 a22 a23 ... a2n
an1 an2 an3 ... ann

B.txt:
b1
b2
...
bn
"""

import numpy as np

A = np.loadtxt('A.txt')
B = np.loadtxt('B.txt')

n = np.size(B)

for k in range(n-1):
    for i in range(k+1, n):
        faktor = A[i, k] / A[k, k]
        for j in range(k+1, n):
            A[i, j] = A[i, j] - faktor*A[k, j]
        B[i] = B[i] - faktor*B[k]

X = np.empty(np.shape(B))
for i in range(n-1, -1, -1):
```

```
    suma = 0
    for j in range(i+1, n):
        suma = suma + A[i, j]*X[j]
    X[i] = (B[i]-suma) / A[i, i]

print(X)
```

Gaussova eliminacija u svom osnovnom (gore objašnjenom) obliku lako može naići na jedan problem. Naime, nije ništa neobično ako se desi da je *pivot* element matrice sustava (tj. element na glavnoj dijagonali) jednak nuli, na primjer:

$$\begin{aligned} 2x_2 + 3x_3 &= 8 \\ 4x_1 + 6x_2 + 7x_3 &= -3 \\ 2x_1 + x_2 + 6x_3 &= 5. \end{aligned} \tag{4.147}$$

U ovom sustavu prvi *pivot* koeficijent a_{11} jednak je nuli i kako u postupku eliminacije prvu jednadžbu množimo s a_{21}/a_{11} imamo problem djeljenja s nulom koje nije moguće provesti! Postupak Gaussove eliminacije uopće ne predviđa ovakvu mogućnost nego "naivno" očekuje da ni jedan element glavne dijagonale neće biti jednak nuli, zbog čega je nazivamo još i **naivna Gaussova eliminacija**.

Rješenje za to predstavlja postupak **pivotiranja** u kojem se provodi zamjena redaka matrica **A** i **b** (tj. promjena redoslijeda jednadžbi) ili pak zamjena i redaka i stupaca matrice **A** (tj. promjena redoslijeda jednadžbi i redoslijeda nepoznanica). U prvom slučaju govorimo o **djelomičnom pivotiranju**, a u drugom o **potpunom pivotiranju**. Kod potpunog pivotiranja potrebno je još i pamtiti vezu između izvornog redoslijeda nepoznanica i redoslijeda stupaca nakon pivotiranja.

4.5.2 LU dekompozicija

Kao što smo vidjeli kod Gaussove metode, transformacija matrice sustava u trokutastu matricu omogućuje jednostavno rješavanje sustava supstitucijom. Pri tome matrica može biti i gornje-trokutasta i donje-trokutasta; u prvom slučaju rješavamo sustav supstitucijom unatrag, a u drugom supstitucijom unaprijed.

Ovo svojstvo trokutastih matrica sustava koristi postupak LU dekompozicije, koji rješavanje linearnog sustava svodi na rješavanje sustava sa trokutastim matricama. Pri tome se provodi postupak eliminacije samo na matrici sustava, što je posebno pogodno ako imamo slučaj u kojem se mijenja vektor desne strane, dok matrica sustava ostaje ista (konstantna), što nije rijetkost u primjeni. Načelno, LU dekompozicija se može promatrati kao oblik Gaussove eliminacije, što ćemo pokazati u nastavku.

Riješimo jednostavan sustav od tri linearne jednadžbe koristeći Gaussovu eliminaciju:

$\epsilon_1:$	$2x_1$	$-2x_2$	$+3x_3$	$=1$
$\epsilon_2:$	$6x_1$	$-7x_2$	$+14x_3$	$=5$
$\epsilon_3:$	$4x_1$	$-8x_2$	$+30x_3$	$=14$
$\epsilon_1' = \epsilon_1:$	$2x_1$	$-2x_2$	$+3x_3$	$=1$
$\epsilon_2' = \epsilon_2 - 3\epsilon_1:$		$-x_2$	$+5x_3$	$=2$
$\epsilon_3' = \epsilon_3 - 2\epsilon_1:$		$-4x_2$	$+24x_3$	$=12$
$\epsilon_1'' = \epsilon_1':$	$2x_1$	$-2x_2$	$+3x_3$	$=1$
$\epsilon_2'' = \epsilon_2':$		$-x_2$	$+5x_3$	$=2$
$\epsilon_3'' = \epsilon_3' - 4\epsilon_2':$			$+4x_3$	$=4$
Supstitucija			x_3	$=1$
unatrag:		$-x_2$	$+5$	$=2 \Rightarrow x_2 = 3$
	$2x_1$	-6	$+3$	$=1 \Rightarrow x_1 = 2$

Napišimo gornji sustav jednadžbi u matričnom obliku:

$$\underbrace{\begin{pmatrix} 2 & -2 & 3 \\ 6 & -7 & 14 \\ 4 & -8 & 30 \end{pmatrix}}_{\mathbf{A}} \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}_{\mathbf{x}} = \underbrace{\begin{pmatrix} 1 \\ 5 \\ 14 \end{pmatrix}}_{\mathbf{b}} \tag{4.148}$$

Prvi korak eliminacije unaprijed podrazumijeva *oduzimanje* prve jednadžbe pomnožene multiplikatorom $a_{21}/a11 = 6/2 = 3$ od druge jednadžbe. Ovaj korak se može zapisati kao umnožak matrica $\mathbf{E}_{21}$ i $\mathbf{A}$:

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 \\ -3 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{\mathbf{E}_{21}} \underbrace{\begin{pmatrix} 2 & -2 & 3 \\ 6 & -7 & 14 \\ 4 & -8 & 30 \end{pmatrix}}_{\mathbf{A}} = \underbrace{\begin{pmatrix} 2 & -2 & 3 \\ 0 & -1 & 5 \\ 4 & -8 & 30 \end{pmatrix}}_{\mathbf{E}_{21}\mathbf{A}}. \tag{4.149}$$

Ako nastavimo s procesom eliminacije, dobit ćemo sljedeće relacije:

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -2 & 0 & 1 \end{pmatrix}}_{\mathbf{E}_{31}} \underbrace{\begin{pmatrix} 2 & -2 & 3 \\ 0 & -1 & 5 \\ 4 & -8 & 30 \end{pmatrix}}_{\mathbf{E}_{21}\mathbf{A}} = \underbrace{\begin{pmatrix} 2 & -2 & 3 \\ 0 & -1 & 5 \\ 0 & -4 & 24 \end{pmatrix}}_{\mathbf{E}_{31}\mathbf{E}_{21}\mathbf{A}}, \tag{4.150}$$

i

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -4 & 1 \end{pmatrix}}_{\mathbf{E}_{32}} \underbrace{\begin{pmatrix} 2 & -2 & 3 \\ 0 & -1 & 5 \\ 0 & -4 & 24 \end{pmatrix}}_{\mathbf{E}_{31}\mathbf{E}_{21}\mathbf{A}} = \underbrace{\begin{pmatrix} 2 & -2 & 3 \\ 0 & -1 & 5 \\ 0 & 0 & 4 \end{pmatrix}}_{\mathbf{E}_{32}\mathbf{E}_{31}\mathbf{E}_{21}\mathbf{A}=\mathbf{U}}. \tag{4.151}$$

Ovdje je matrica **U** gornje trokutasta i odgovara gornje trokutastoj matrici dobivenoj eliminacijom unaprijed. Također. možemo primijetiti da matrice $\mathbf{E}_{21}$, $\mathbf{E}_{31}$ i $\mathbf{E}_{32}$ sadrže *negativne* multiplikatore Gaussove eliminacije unaprijed.

Moguće je rekonstruirati matricu **A** pomoću matrica $\mathbf{E}_{21}$, $\mathbf{E}_{31}$, $\mathbf{E}_{32}$ i **U**.

$$\begin{aligned}\mathbf{E}_{32}\mathbf{E}_{31}\mathbf{E}_{21}\mathbf{A} &= \mathbf{U}\\ \mathbf{A} &= \mathbf{E}_{21}^{-1}\mathbf{E}_{31}^{-1}\mathbf{E}_{32}^{-1}\mathbf{U}\\ \mathbf{A} &= \mathbf{L}\mathbf{U}\end{aligned} \tag{4.152}$$

Inverzi matrica $\mathbf{E}_{ik}$ su jedinične matrice koje dodatno sadrže multiplikator u retku i i stupcu k, odnosno:

$$\begin{aligned}\mathbf{L} &= \mathbf{E}_{21}^{-1}\mathbf{E}_{31}^{-1}\mathbf{E}_{32}^{-1}\\ &= \begin{pmatrix}1 & 0 & 0\\ 3 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 2 & 0 & 1\end{pmatrix}\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 4 & 1\end{pmatrix} = \begin{pmatrix}1 & 0 & 0\\ 3 & 1 & 0\\ 2 & 4 & 1\end{pmatrix}.\end{aligned} \tag{4.153}$$

Zaključno, matricu **A** možemo zapisati kao produkt dviju matrica, donje-trokutaste **L** i gornje-trokutastu **U**, ili,

$$\mathbf{A} = \mathbf{L}\mathbf{U}. \tag{4.154}$$

Ovdje su koeficijenti gornje trokutaste matrice **U** jednaki konačnom obliku nakon Gaussove metode eliminacije unaprijed, a koeficijenti donje trokutaste matrice **L** na glavnoj dijagonali se sastoje od jedinica ($l_{ii} = 1$) i multiplikatora

$$l_{ik} = \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}}, \tag{4.155}$$

upotrijebljenim u izrazu (4.137).

LU dekompozicija se provodi često upotrebljavanim Doolittleovim algoritmom, čiji postupak ćemo pokazati na primjeru matrice sustava 3×3:

$$\mathbf{A} = \begin{bmatrix}a_{11} & a_{12} & a_{13}\\ a_{21} & a_{22} & a_{23}\\ a_{31} & a_{32} & a_{33}\end{bmatrix}. \tag{4.156}$$

Na ovoj matrici sada provodimo postupak Gaussove eliminacije:

- Pomnožiti prvi red matrice s $l_{21} = \frac{a_{21}}{a_{11}}$ i oduzeti od drugog reda matrice da se eliminira a_{21},
- Pomnožiti prvi red matrice s $l_{31} = \frac{a_{31}}{a_{11}}$ i oduzeti od trećeg reda matrice da se eliminira a_{31},
- Pomnožiti modificirani drugi red matrice s $l_{32} = \frac{a'_{32}}{a'_{22}}$ i oduzeti od trećeg reda matrice da se eliminira a'_{32}.

Time smo dobili matricu koja je gornje-trokutasta i koju ćemo nazvati **U**:

$$\mathbf{U} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & 0 & a''_{33} \end{bmatrix}, \tag{4.157}$$

dok ćemo izračunate koeficijente l složiti u donje-trokutastu matricu **L**:

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix}, \tag{4.158}$$

s time da smo joj na glavnoj dijagonali stavili jedinice.

Napomena
Na računalu je uvijek korisno štedjeti memoriju, pa se tako i matrice **L** i **U** mogu kompaktno zapisati u jednoj matrici:

$$\tilde{\mathbf{A}} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ l_{21} & a'_{22} & a'_{23} \\ l_{31} & l_{32} & a''_{33} \end{bmatrix}.$$

Naivna implementacija ovog postupka na primjeru matrice 3×3 prikazana je u nastavku (Python kod 4.18).

Python kod 4.18 LU dekompozicija

```
import numpy as np

A  = np.array([[2, -1, -2],
               [-4, 6, 3],
               [-4, -2, 8]])
n = np.shape(A)[0]

L, U = np.eye(np.shape(A)[0]), np.zeros_like(A)

for i in range(n):
        for k in range(i, n):
            sum = 0;
            for j in range(i):
                sum += (L[i, j] * U[j, k]);
            U[i][k] = A[i][k] - sum;
        for k in range(i, n):
            if i!=k:
                sum = 0;
                for j in range(i):
                    sum += (L[k, j] * U[j, i]);
                L[k, i] = (A[k, i] - sum) / U[i, i];

print('L:\n', L)
print('U:\n', U)
```

Ako za potrebe rješavanja sustava $\mathbf{Ax} = \mathbf{b}$ možemo kreirati donje-trokutastu matricu $\mathbf{L}$ i gornje-trokutastu matricu $\mathbf{U}$ za koje vrijedi $\mathbf{A} = \mathbf{LU}$, npr.:

$$\begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, \tag{4.159}$$

onda vrijede sljedeće relacije:

$$\mathbf{Ax} = (\mathbf{LU})\mathbf{x} = \mathbf{L}(\mathbf{Ux}) = \mathbf{b}. \tag{4.160}$$

Uvedemo li supstituciju $\mathbf{Ux} = \mathbf{d}$, onda vrijedi $\mathbf{Ld} = \mathbf{b}$. Odnosno,

$$\begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}. \tag{4.161}$$

Ovdje lako riješimo nepoznati vektor $\mathbf{d}$ supstitucijom unaprijed. Konačno možemo riješiti izvorni sustav $\mathbf{Ux} = \mathbf{d}$ supstitucijom unatrag, ili

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix}. \tag{4.162}$$

Postavlja se pitanje zašto je bolje riješavati dva sustava jednadžbi ($\mathbf{Ld} = \mathbf{b}$ i $\mathbf{Ux} = \mathbf{d}$) umjesto jednog (inicijalni sustav $\mathbf{Ax} = \mathbf{b}$). Naime, osim što se ovim postupkom omogućuje proizvoljno mjenjanje vektora desne strane $\mathbf{b}$ bez utjecaja na matrice $\mathbf{L}$ i $\mathbf{U}$, sustave $\mathbf{Ld} = \mathbf{b}$ i $\mathbf{Ux} = \mathbf{d}$ se lako (učinkovito) može rješavati jer su matrice $\mathbf{L}$ i $\mathbf{U}$ trokutaste matrice, što znači da ih rješavamo običnom supstitucijom. (S obzirom da je $\mathbf{L}$ donje-trokutasta, sustav $\mathbf{Ld} = \mathbf{b}$ rješavamo supstitucijom unaprijed; analogno, sustav $\mathbf{Ux} = \mathbf{d}$ rješavamo supstitucijom unatrag.)

Da rezimiramo, postupak rješavanja sustava jednadžbi LU dekompozicijom provodi se u tri koraka:

(1) LU dekompozicija matrice sustava, tj. dobivanje matrica $\mathbf{L}$ i $\mathbf{U}$ iz matrice $\mathbf{A}$,

(2) Rješavanje sustava $\mathbf{Ld} = \mathbf{b}$ supstitucijom unaprijed, čime dobijemo vektor $\mathbf{d}$,

(3) Rješavanje sustava $\mathbf{Ux} = \mathbf{d}$ supstitucijom unatrag, čime dobijemo konačni vektor rješenja $\mathbf{x}$.

4.5.3 Iterativne metode

Za razliku od pristupa metoda Gaussove eliminacije i LU dekompozicije koje u načelu daju egzaktno rješenje (tj. savršeno točno rješenje, ako zanemarimo točnost zapisa decimalnih brojeva na računalu), problemu rješavanja sustava linearnih jednadžbi možemo pristupiti i na način da pokušamo doći do nekih približnih rješenja.

Svaku pojedinu jednadžbu linearnog sustava

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \ldots + a_{3n}x_n &= b_3 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \ldots + a_{nn}x_n &= b_n, \end{aligned} \tag{4.163}$$

možemo iskoristiti da eksplicitno izrazimo po jednu nepoznanicu:

$$\begin{aligned} x_1 &= \frac{b_1 - (a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n)}{a_{11}} \\ x_2 &= \frac{b_2 - (a_{21}x_1 + a_{23}x_3 + \ldots + a_{2n}x_n)}{a_{22}} \\ x_3 &= \frac{b_3 - (a_{31}x_1 + a_{32}x_2 + a_{34}x_4 + \ldots + a_{3n}x_n)}{a_{33}} \\ &\vdots \\ x_n &= \frac{b_n - (a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{n-1,n}x_{n-1})}{a_{nn}} \end{aligned} \tag{4.164}$$

odnosno općenito:

$$x_i = \frac{b_i - \sum\limits_{j=1, j\neq i}^{n} a_{ij}x_j}{a_{ii}} \quad i = 1,2,\ldots,n \tag{4.165}$$

Sada jednostavno u ove eksplicitne jednadžbe ubacimo neka početna pretpostavljena rješenja, npr.

$$x_1^{(1)} = x_2^{(1)} = \ldots = x_n^{(1)} = 0, \tag{4.166}$$

ili, još bolje, neka druga, realnim problemom uvjetovana pretpostavljena rješenja $x_1^{(1)}, x_2^{(1)}, \ldots, x_n^{(1)}$ (u inženjerstvu uvijek imamo barem nekakve grube pretpostavke o očekivanim rješenjima), čime ćemo dobiti nova rješenja $x_1^{(2)}, x_2^{(2)}, \ldots, x_n^{(2)}$. Ova nova rješenja bi, općenito govoreći, trebala biti točnija od početnih što znači da bi kroz nekoliko ciklusa (iteracija) ovog postupka mogli doći do rješenja zadovoljavajuće točnosti.

Ovakav postupak, gdje se u svakom (k-tom) koraku za izračun novih vrijednosti $x_i^{(k)}$ uvrštavaju aproksimacije nepoznanica iz prethodnog $(k-1)$ koraka, tj.

$$x_i^{(k)} = \frac{b_i - \sum\limits_{j=1, j\neq i}^{n} a_{ij}x_j^{(k-1)}}{a_{ii}} \quad i = 1,2,\ldots,n \tag{4.167}$$

zove se **Jacobijeva metoda**.

Nešto racionalnije bi bilo u svakom koraku izračuna $x_i^{(k)}$ uvrštavati nove aproksimacije za nepoznanice već izračunate u tekućem koraku $(x_1^{(k)}, \ldots, x_{i-1}^{(k)})$, a stare aproksimacije samo za one nepoznanice do kojih postupak još nije stigao $x_{i+1}^{(k-1)}, \ldots, x_n^{(k-1)}$

odnosno

$$x_i = \frac{b_i - a_{i,1}x_1^{(k+1)} - \ldots - a_{i,j-1}x_{i-1}^{(k+1)} - a_{i,j+1}x_{i+1}^{(k)} - \ldots - a_{i,n}x_n^{(k)}}{a_{ii}} \quad i = 1,2,\ldots,n. \tag{4.168}$$

Ova varijanta postupka zove se **Gauss-Seidelova metoda** i rezultira bržom konvergencijom.

Proces iterativnog približavanja rješenju zaustavljamo u trenutku kada se rješenja stabiliziraju na željenu točnost, npr. kada relativna razlika aproksimacije rješenja iz posljednje dvije iteracije postane zanemarivo mala:

$$\left| \frac{x_i^{(k)} - x_i^{(k-1)}}{x_i^{(k)}} \right| < \varepsilon, \; i = 1,2,\ldots,n, \tag{4.169}$$

gdje je ε dopušteno relativno odstupanje u rezultatu koje smatramo zanemarivim.

Python kod 4.19 prikazuje programsku implementaciju ovih iterativnih metoda za rješavanje sustava linearnih jednadžbi koji je definiran u vanjskim tekstualnim datotekama.

Python kod 4.19 Jacobijeva i Gauss-Seidelova metoda

```
"""
format ulaznih datoteka:

A.txt:
a11 a12 a13 ... a1n
a21 a22 a23 ... a2n
an1 an2 an3 ... ann

B.txt:
b1
b2
...
bn
"""

import numpy as np

A = np.loadtxt('A.txt')
B = np.loadtxt('B.txt')

n = np.size(B)

# početno pretpostavljeno rješenje
X = np.zeros(np.shape(B))
Xpr = np.zeros(np.shape(B))

maxit = 100
eps = 0.001

it = 0
err = eps + 1
while err > eps and it < maxit:
    it += 1
    for i in range(n):
        suma = 0.0
        for j in range(n):
```

```
            if j != i:
                suma = suma + A[i, j] * Xpr[j] # Jacobijeva metoda
                # suma = suma + A[i, j] * X[j] # Gauss-Seidelova metoda
        X[i] = (B[i] - suma) / A[i, i]
    err = np.average(np.abs((X - Xpr) / X))
    Xpr = np.copy(X)

print(it)
print(X)

# NumPy rješenje
print(np.linalg.solve(A, B))
```

Prirodno se postavlja pitanje ima li ikakvih garancija za konvergenciju ovih metoda, tj. da li smo uopće sigurni da ćemo na ovaj način doći do rješenja. Naime, može se pokazati da je konvergencija sigurna, ali samo za sustave za koje vrijedi:

$$|a_{ii}| > \sum_{j=1, i\neq j}^{n} |a_{ij}|,\ i = 1,2,\ldots,n. \tag{4.170}$$

Takve sustave zovemo **dijagonalno dominantnima**, jer imaju elemente na glavnoj dijagonali veće od svih ostalih elemenata u datom redu zajedno (po apsolutnoj vrijednosti).

Sustavi koji proizlaze iz inženjerskih problema su ponekad dijagonalno dominantni, što iteracijske metode čini posebno pogodnima za takve probleme. Kod nekih drugih sustava, dijagonalna se dominantnost može postići pivotiranjem. No, ponekad nemamo tako jednostavno rješenje za problem nedovoljno dobre konvergencije iterativnih metoda.

Ako se desi da iterativna metoda proizvodi aproksimacije koje jako osciliraju oko točnog rješenja, ili ako metoda presporo konvergira prema točnom rješenju, možemo na kraju svakog koraka metode provesti još i kombiniranje rezultata dobivenih u dva posljednja koraka:

$$x_i^{(k)} = \lambda \tilde{x}_i^{(k)} + (1-\lambda) x_i^{(k-1)}, \tag{4.171}$$

gdje je $\tilde{x}_i^{(k)}$ rješenje iz zadnjeg koraka dobiveno osnovnim postupkom iterativnog rješavača. Ova procedura zove se **relaksacija**, gdje parametar λ predstavlja težinski faktor kojim upravljamo procesom konvergencije na sljedeći način:

- $\lambda \in \langle 0,1 \rangle$ podrelaksacija – smanjuje se utjecaj nove aproksimacije (koristi se ako rješenja osciliraju).
- $\lambda \in \langle 1,2 \rangle$ nadrelaksacija – povećava se utjecaj nove aproksimacije (koristi se ako metoda presporo konvergira).

4.6 Fourierova analiza

Stefan Ivić

Fourierova analiza je naziv za tehnike kojima se vremensku funkciju $f(t)$ (često nazivana signal) rastavlja na jednostavne periodičke funckije različitih frekvencija koje kumulativno čine izvorni signal. Korištenjem uobičajene terminologije može se reći da se funkciju vremenske domene pretvara u funkciju spektralne domene.

U nastavku su dane neke matematičke definicije koje se koriste u daljem tekstu ovog poglavlja.

Periodičnost

Funkcija $y(t)$ je periodična, s periodom T, ako postoji broj $T > 0$ tako da je

$$y(t+T) = y(t)$$

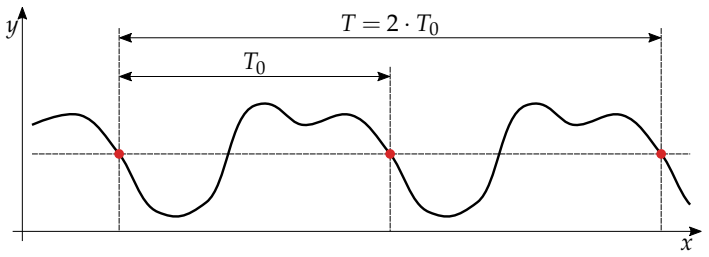
za svaki t. Najmanji period T_0 za koji vrijedi uvjet periodičnosti naziva se **temeljni period** funkcije y, a svaki njegov višekratnik $T = nT_0$ je period funkcije y (Slika 4.42):

$$y(t+nT_0) = y(t),\ n = \pm 1, \pm 2, \pm 3 \ldots$$

Recipročna vrijednost perioda funkcije naziva se **frekvencija**:

$$f = \frac{1}{T}$$

a frekvencija koja odgovara temeljenom periodu T_0 naziva se **temeljna frekvencija** f_0.



Slika 4.42
Periodična funkcija

Parne i neparne funkcije

Funkcija $y(t)$ naziva se **parnom funkcijom** ukoliko vrijedi

$$y(-t) = y(t).$$

Za **neparnu funkciju** $y(t)$ vrijedi

$$y(-t) = -y(t).$$

Graf parne funkcije je osno simetričan s obzirom na os ordinate dok je graf neparne funkcije centralno simetričan s obzirom na ishodište koordinatnog sustava.

Eulerova formula

Eulerova formula postavlja vezu između trigonometrijskih funckija i kompleksnih brojeva tj. preciznije kompleksne eksponencijalne funckije. Za realnu varijablu x vrijedi

$$e^{ix} = \cos x + i \sin x$$

gdje je i imaginarna jedinica. Iz Eulerove formule, lako je pokazati da vrijedi:

$$e^{ix} + e^{-ix} = 2\cos x$$

odnosno

$$\cos x = \mathrm{Re}\left(e^{ix}\right) = \frac{e^{ix} + e^{-ix}}{2}.$$

Analogno, sinus funkcija se može zapisati kao

$$\sin x = \mathrm{Im}\left(e^{ix}\right) = \frac{e^{ix} - e^{-ix}}{2i}$$

gdje $\mathrm{Re}(z)$ predstavlja relani dio a $\mathrm{Im}(z)$ imaginarni dio nekog kompleksnog broja z.

4.6.1 Fourierov red

Razmotrimo funkciju $y(\tau)$ koja je periodična sa periodom T. Skaliranjem varijable τ možemo mijanjati period funkcije. Ukoliko definiramo nezavisnu varijablu $t = \frac{2\pi}{T}\tau$, funkcija $y(t)$ je tada periodična sa periodom 2π:

$$y(t + 2\pi) = y(t).$$

Ovakvu manipulaciju možemo napraviti sa bilo kojom periodičkom funkcijom, a pošto je funkcija periodična sa periodom 2π dovoljno je promatrati funkciju na intervalu $[-\pi, \pi]$ ili na nekom drugom intervalu duljine 2π.

Proizvoljnu periodičku funkciju $y(t)$ na intervalu $[-\pi, \pi]$ možemo zamijeniti beskonačnom sumom sinus i kosinus funkcija

$$y(t) = \frac{1}{2}a_0 + \sum_{n=1}^{\infty} a_n \cos(nt) + \sum_{n=1}^{\infty} b_n \sin(nt)$$

koju nazivamo Fourierovim redom. Koeficijenti a_n i b_n su konstante koje nazivamo Fourierovim koeficijentima a definirani kao

$$\begin{aligned} a_0 &= \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)\mathrm{d}t \\ a_n &= \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)cos(nt)\mathrm{d}t \\ b_n &= \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)sin(nt)\mathrm{d}t. \end{aligned}$$

Napomena
Ukoliko je podintegralna funkcija neparna a granice integracije su simetrične, određeni integral je jednak nuli. Dodatno, umnožak parne i neparne funkcije je neparna funckija. S obzirom na navedeno, ukoliko je $y(t)$ parna funkcija, tada je $y(t)\sin(nt)$ neparna funkcija pa je $b_n = 0$ za svaki n. Analogno, za neparane funkcije $y(t)$, $y(t)\cos(nt)$ je neparna funkcija, pa vrijedi da je $a_n = 0$ za svaki n.

Razvoj periodične funkcije u Fourierov red moguć je i za kompleksne funkcije $y : \mathbb{R} \to \mathbb{C}$. Tada, koristeći Eulerovu formulu, dolazimo da elegantnog i kompaktnog zapisa Fourierovog reda:

$$y(t) = \sum_{n=-\infty}^{+\infty} A_n e^{in(2\pi f_0)t}$$

gdje je f_0 temeljna frekvencija funkcije y, a Fourierovi koeficijenti A_n su kompleksni brojevi.

Napomena
Razvojem relane funkcije $y : \mathbb{R} \to \mathbb{C}$ u kompleksni Fourierov red, dobivaju se kompleksni keoficijenti A_n. Međutim, vrijedi da je

$$A_n = A_{-n}^*$$

gdje * označava kompleksno konjugirani par. Zbog navedenog se u sumi reda imaginarni dijelovi Fourierovih koeficijenata A_n i A_{-n} međusobno poništavaju te ostaje samo realni dio pa je i rekonstruirana funkcijska vrijednost realana.

4.6.2 Fourierova transformacija

Razvoj periodičke funkcije u Fourierov red definiran je diskretnom sumom kompleksnih eksponencijalnih funkcija. Fourierova transformacija omogućuje nam spektralnu reprezentaciju općenitih (ne nužno periodičkih) funkcija pomoću kontinuirane superpozicije (integrala) kompleksnih eksponencijalnih funkcija. Fourierovu transformaciju se može sagledavati kao limes Fourierov niz kod kojih se neperiodičke funkcije razmatraju kao funkcije sa periodom $T \to \infty$. Shodno tome, razmak između dviju susjednih temeljnih frekvencija

$$(n+1)f_0 - nf_0 = f_0 = \frac{1}{T} \to \mathrm{d}f$$

pa suma u definiciji Fourierovog reda postaje integral a granice integracije kod izraza za koeficijente se mijenja sa intervala perioda $[-\pi, \pi]$ na $[-\infty, \infty]$.

$$y(t) = \int_{-\infty}^{+\infty} \left[\int_{-\infty}^{+\infty} x(\tau) e^{-i2\pi f\tau} \mathrm{d}\tau \right] e^{i2\pi ft} \mathrm{d}f$$

Fourierova transformacija kompleksne (ili samo realne) integrabilne funckije $y : \mathbb{R} \to \mathbb{C}$ definirana je kao

$$\hat{y}(f) = \int_{-\infty}^{\infty} y(t)\ e^{-i2\pi tf}\, \mathrm{d}t$$

gdje t predstavlja vrijeme a f frekvenciju. Funkcija $\hat{y}(f)$ je Fourierova transformacija funkcije $y(t)$ te predstavlja amplitude baznih funkcija u ovisnosti o frekvenciji f.

Funckija $y(t)$ se može rekonstruirati iz $\hat{y}(f)$ pomoću **inverzne Fourierove transformacije**:

$$y(t) = \int_{-\infty}^{\infty} \hat{y}(f) e^{i2\pi tf}\, \mathrm{d}f.$$

4.6.3 Diskretna Fourierova transformacija

U praktičnoj primjeni rijeđe se analiziraju poznate analitičke funckije, već dolazi do potrebe za analiziranje signala dobivenomg mjerenjima ili simulacijama koji dolazi u obliku diskretnih podataka. Za skup podataka (x_j, y_j), $j = 0, \ldots, n-1$ uz uniformno diskretiziranu domenu $x_j = j \cdot \Delta x$, **diskretnu Fourierovu transformaciju** (DFT) možemo definirati na sljedeći način:

$$A_k = \sum_{j=0}^{n-1} y_j e^{-i \cdot 2\pi \cdot f_k \cdot x_j}, \qquad k = 0, 1, \ldots, n-1 \tag{4.172}$$

Inverzna diskretna Fourierova transformacija (IDFT) omogućuje rekonstrukciju signala iz poznatih Fourierovih koeficijenata A:

$$y_j = \sum_{k=0}^{n-1} A_k e^{i \cdot 2\pi \cdot f_k \cdot x_j}, \qquad j = 0, 1, \ldots, n-1.$$

Napomena
Nyquistov teorem uzorkovanja kaže da frekvencija uzorkovanje signala mora biti barem dvostruko veća od najviše frekvencije toga signala. Uz ovaj limit moguća je točna detekcija prisutnih frekvencija (DFT) te točna rekonstrukcija signala (IDFT). Ovako definirana minimalna potrebna frekvencija uzorkovanja naziva se **Nyquistovom frekvencijom**.

4.6.4 Brza Fourierova transformacija

Brza Fourierova transformacija (eng. *Fast Fourier transform* ili FFT) je numerički algoritam za učinkovito izvođenje diskretne Fourierove transformacije. Za razliku od izraza (4.172) za diskretnu Fourierovu transformaciju, koji zahtjeva red veličine n^2 operacija, FFT omogućuje dobivanje istih rezultata uz red veličine $n \log_2 n$ računalnih operacija.

Cooley i Tukey (1965)

4.6.5 Numpy FFT modul

Numpy sadrži modul `numpy.fft` koji nudi funkcije i FFT algoritme za diskretnu Fourierovu transformaciju (DFT). Navedeni modul omogućuje rad s kompleksnom vremenskom domenom i kompleksnom frekvnecijskom domenom, pri čemu je diskretna Fourierova transformacija diskretnog signala $(y_0, y_1, \ldots, y_n - 1)$ definirana kao:

$$A_k = \sum_{j=0}^{n-1} y_j e^{-2\pi i j k / n} \qquad k = 0, \ldots, n-1.$$

Inverzna Fourierova transformacija definirana je preko

$$y_j = \frac{1}{n} \sum_{k=0}^{n-1} A_k e^{2\pi i j k / n} \qquad m = 0, \ldots, n-1.$$

Za skup realnih ulaznih podataka $y_m \in \mathbb{R}, m = 0, \ldots, n$, dostupna je funkcija `rfft` koja izračunava (kompleksne) Fourierove koeficijente A_k a čija je sintaksa:

```
numpy.fft.rfft(y, m=None)
```

gdje je `y` polje (1D `numpy.ndarray`) ulaznog signala a `n` je broj željenih Fourierovih koeficijenata $A_k, k = 0, \ldots, m-1$. Ako `m` nije zadan, onda se koriti $m = n$, dok se za zadani m manji od n, `rfft` vraća "odrezani" vektor Fourierovih koeficijenata a za m veći od n koeficijenti $A_n, A_{n+1}, \ldots, A_{m-1}$ se postavljaju na nulu.

Ako ulazne podatke promatramo kao vremensko-prostorni signal $y_j = y(t_j)$ zadan poljima `y` i `t` (pri čemu je `n = y.size()`) na kojima je provedena diskretna Fourierova transformacija `A = np.fft.rfft(y)`, tada vektor `np.abs(A)/n` predstavlja spektar amplituda, dok `np.angle(A)` predstavlja fazni spektar.

Kako bi potpuno analizirali signal u frekventnom prostoru, uz amplitude i pomake faze dobivene sa `rfft`, potrebno je odrediti i pripadajuće frekvnecije. Iako se frekvencije jednostavno mogu računati pomoću jednostavnog izraza $f_i = \frac{i}{\Delta x n}$ (gdje je $i = 0, 1, \ldots, \frac{n}{2}$ za parni n, a $i = 0, 1, \ldots, \frac{n-1}{2}$ za neparni n) u `numpy.fft` modulu postoji pomoćna funkcija koja računa frekvencije za dobivene spektre amplituda i faza:

```
numpy.fft.rfftfreq(n, d=1.0)
```

gdje je `n` duljina ulaznog signala (`n = y.size()`) a `d` je korak uzorkovanja (`d = t[1] - t[0]`).

U primjeru Python koda 4.20 izvorni signal sastavljan od konstante, 2 kosinusna signala s različitim amplitudama i pomacima u fazi te nasumičnog šuma. Korištenjem `numpy.fft` modula napravljena je spektralna analiza i detekcija sastavnih signala odnsono njihovih frekvencija, amplituda i pomaka u fazi.

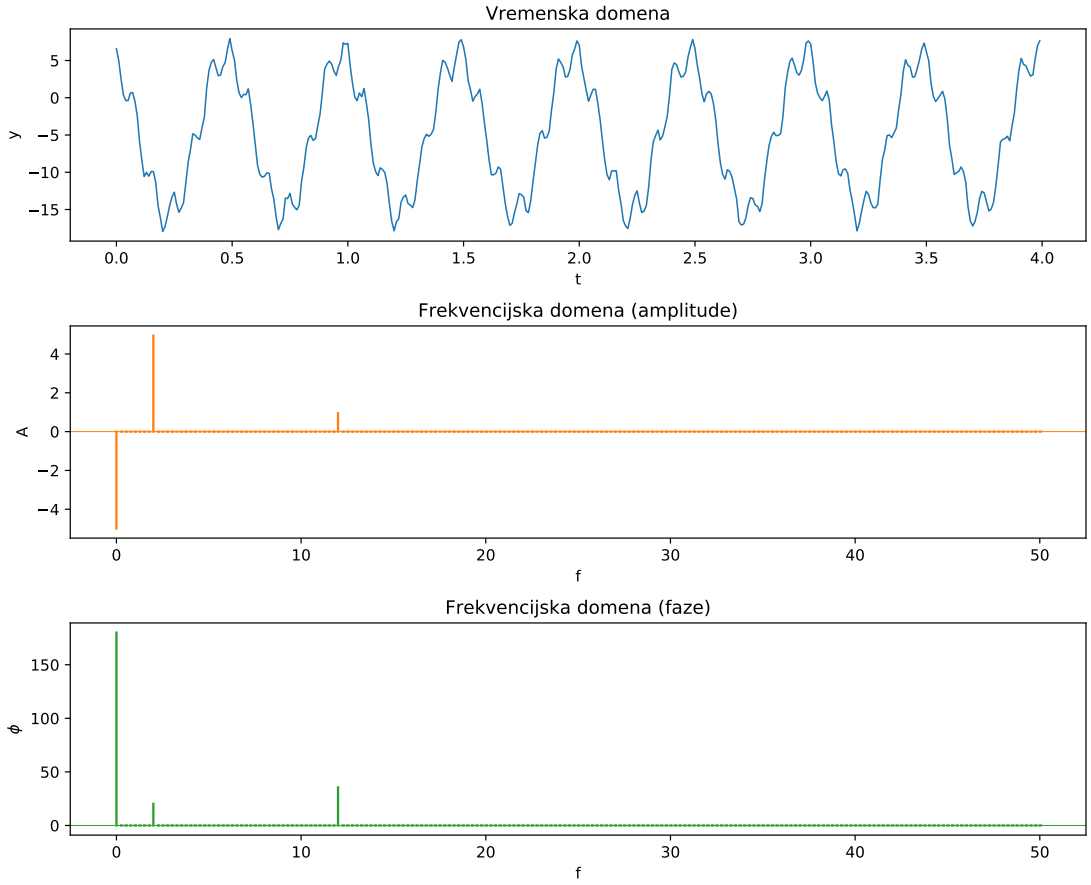
Python kod 4.20 Iterativno rješavanje sustava linearnih jednadžbi

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

"""
Konstrukcija signala y(t) koji je sačinjen od:
    - konstante A0
    - kosinusnog signala amplitude A1, frekvencije f1 i pomaka u fazi phi1
    - kosinusnog signala amplitude A2, frekvnecije f2 i pomaka u fazi phi2
    - te nasumičnog šuma amplitude A3.
"""

# Vrijeme uzorkovanja
dt = 0.01
# Vrijeme
t = np.arange(0, 4, dt)
# Broj uzoraka
n = t.size

# Frekvencije
f1, f2 = 2, 12
# Amplitude
A1, A2 = 10.5, 2.3
# Pomak u fazi
phi1, phi2 = np.deg2rad(20), np.deg2rad(35)
# Konstantna amplituda i amplituda šuma
A0, A3 = -5, 0.6

# Signal y(t)
```

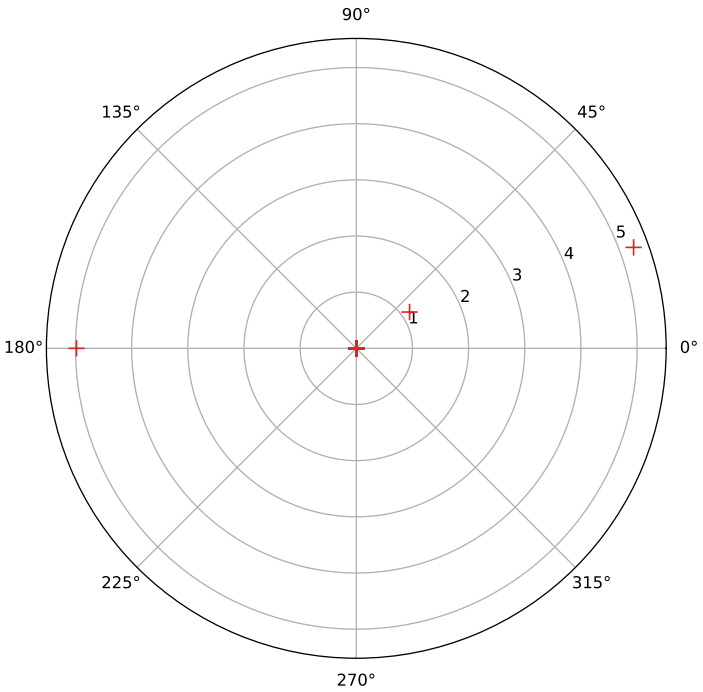
```
y = A0 + \
    A1 * np.cos(f1 * t * 2 * np.pi + phi1) + \
    A2 * np.cos(f2 * t * 2 * np.pi + phi2) + \
    A3 * np.random.uniform(-1,1, n)


"""
FFT
"""

# Fourierovi koeficijenti
Y = np.fft.rfft(y)
#Y = np.fft.fft(y)
# Frekvencije
F = np.fft.rfftfreq(y.size, dt)
#F = np.fft.fftfreq(y.size, dt)

# Izračun pravih amplituda
A = Y / n

# Postavljanje malih amplituda na nulu (čišći graf faza)
A[np.abs(A) < 0.1] = 0.0

"""
Vizualizacija
"""

plt.figure(figsize=[10, 8])
gs = gridspec.GridSpec(3, 1,
                       hspace=0.4,
                       left=0.07, right=0.99,
                       bottom=0.055, top=0.97)

ax1 = plt.subplot(gs[0])
ax2 = plt.subplot(gs[1])
ax3 = plt.subplot(gs[2])

# Vremenska domena
ax1.plot(t, y, lw=1)
ax1.set_title('Vremenska domena')
ax1.set_xlabel('t')
ax1.set_ylabel('y')

# Frekvencijska domena
ax2.set_title('Frekvencijska domena (amplitude)')
ax2.set_xlabel('f')
ax2.set_ylabel('A')
ax3.set_title('Frekvencijska domena (faze)')
ax3.set_xlabel('f')
ax3.set_ylabel('$\phi$')
for f, a in zip(F, A):
    ax2.plot([f, f], [0, a], c='C1')
    ax3.plot([f, f], [0, np.angle(a)/np.pi*180], c='C2')
ax2.axhline(0, c='C1', lw=0.5)
ax3.axhline(0, c='C2', lw=0.5)

# Prikaz amplituda i faza u kompleksnoj ravnini
plt.figure(figsize=[6, 6])
ax = plt.subplot(1, 1, 1, projection='polar')
ax.plot(np.angle(A), np.abs(A), 'C3+', ms=10)
plt.subplots_adjust(left=0.06, right=0.94,
```

```
                    bottom=0.06, top=0.94)

# Ispis
for f in [0, f1, f2]:
    k = np.argmin(np.abs(F - f)) # Pronađi indeks najbliže frekvencije
    phi_ang = np.angle(A[k])
    phi_atan = np.arctan2(A.imag[k], A.real[k])
    print(f'Frekvencija: {F[k]:.4f}')
    print(f'Amplituda: {np.sign(A.real[k])*np.abs(A[k]):.4f}')
    print(f'Faza (angle): {np.rad2deg(phi_ang):.4f}')
    print(f'Faza (arctan2): {np.rad2deg(phi_atan):.4f}')
    print()
```

```
Frekvencija: 0.0000
Amplituda: -4.9890
Faza (angle): 180.0000
Faza (arctan2): 180.0000

Frekvencija: 2.0000
Amplituda: 5.2699
Faza (angle): 20.1467
Faza (arctan2): 20.1467

Frekvencija: 12.0000
Amplituda: 1.1639
Faza (angle): 35.5009
Faza (arctan2): 35.5009
```



Slika 4.43 Graf vremenske i frekvencijske domene iz Python koda 4.20



Slika 4.44
Prikaz amplituda i faza u kompleksnoj ravnini iz Python koda 4.20

4.7 Rastav na singularne vrijednosti

Stefan Ivić

4.7.1 Rang matrice

Rang matrice jednak je maksimalnom broju linearno nezavisnih stupaca odnosno maksimalnom broju nezavisnih redaka. Maksimalan broj linearno nezavisnih stupaca jednak je maksimalnom broju linearno nezavisnih redaka matrice. Iz toga slijedi da je

$$\text{rang}(\mathbf{A}) = \text{rang}(\mathbf{A}^T).$$

Ako je matrica $\mathbf{A}$ dimenzija $m \times n$, tada za rang matrice $\mathbf{A}$ vrijedi:

$$\text{rang}(\mathbf{A}) \leq \min(m,n),$$

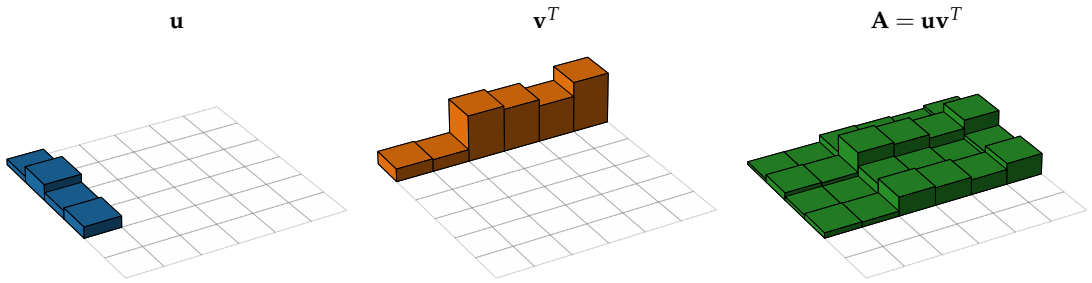
tj. rang matrice ne može biti veći od broja redaka ili od broja stupaca matrice.

Matrice **ranga 0** su nul-matrice.

Matrice čiji su svi retci višekratnici ostalih redaka i svi stupci su višekratnici ostalih stupaca imaju **rang 1**. Matrica $m \times n$ ranga 1 je jednaka vektorskom produktu $\mathbf{u}\mathbf{v}^T$ vektora $\mathbf{u}$ veličine m i vektora $\mathbf{v}$ veličine n:

$$\mathbf{A} = \mathbf{u}\mathbf{v}^T = \begin{bmatrix} \text{—} & u_1\mathbf{v}^T & \text{—} \\ \text{—} & u_2\mathbf{v}^T & \text{—} \\ & \vdots & \\ \text{—} & u_m\mathbf{v}^T & \text{—} \end{bmatrix} = \begin{bmatrix} | & | & & | \\ v_1\mathbf{u} & v_2\mathbf{u} & \cdots & v_n\mathbf{u} \\ | & | & & | \end{bmatrix}$$

što je grafički prikazano na Slici 4.45.



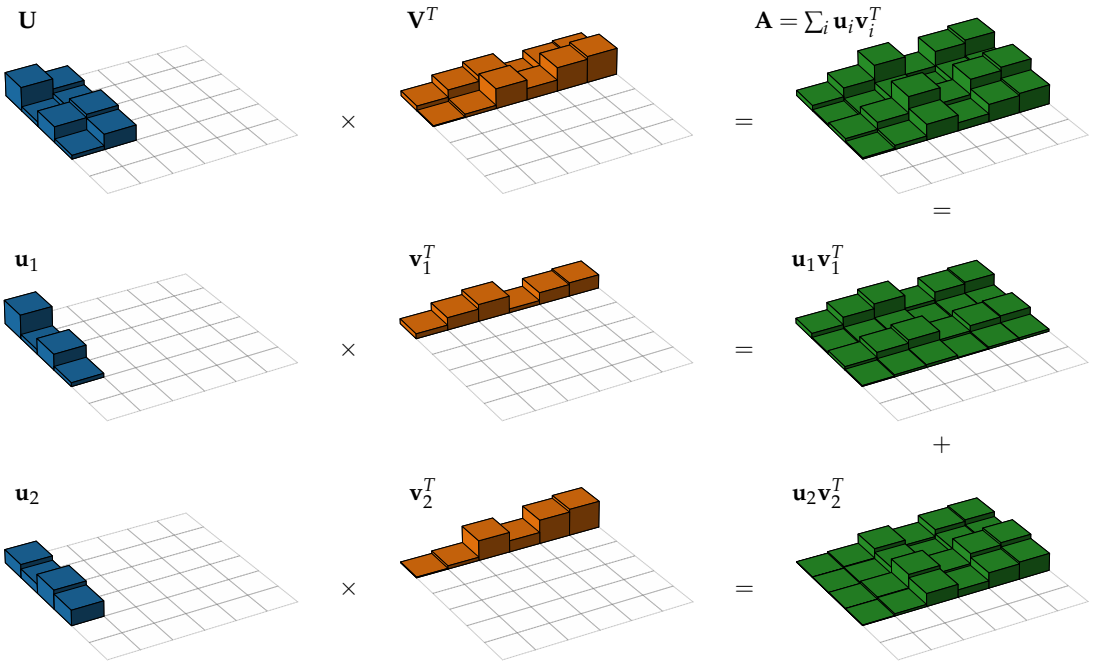
Slika 4.45 Matrica ranga 1 može se prikazati kao umnožak dva vektora

Kod matrice ranga 2 vrijedi superpozicija dvije matrice ranga 1:

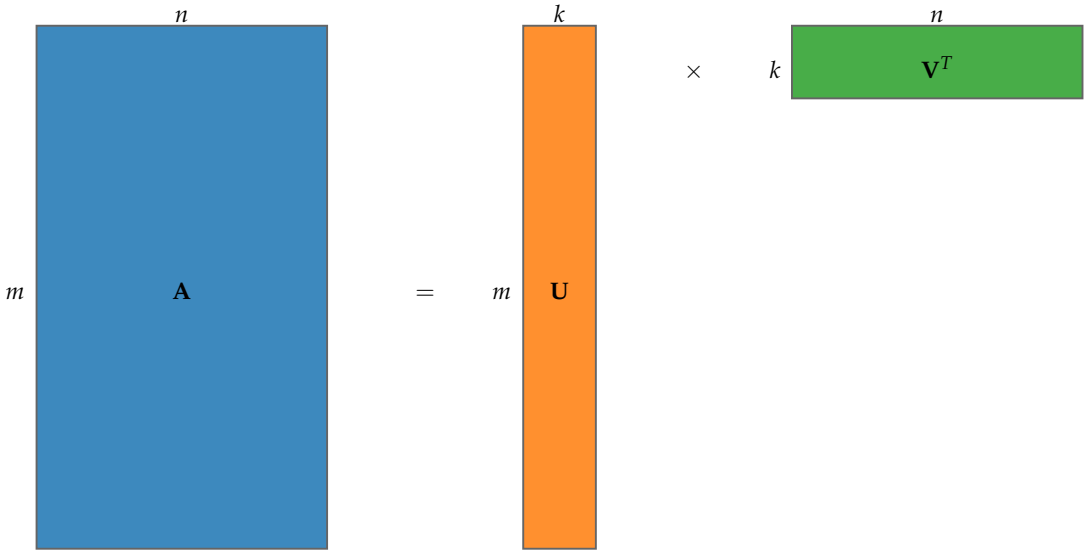
$$\mathbf{A} = \mathbf{u}\mathbf{v}^T + \mathbf{w}\mathbf{z}^T = \begin{bmatrix} | & | \\ \mathbf{u} & \mathbf{w} \\ | & | \end{bmatrix} \cdot \begin{bmatrix} \text{—} & \mathbf{v}^T & \text{—} \\ \text{—} & \mathbf{z}^T & \text{—} \end{bmatrix}$$

Dekompozicija matrice ranga k

$$\begin{array}{ccccc} \mathbf{A} & = & \mathbf{U} & \times & \mathbf{V}^T \\ m \times n & & m \times k & & k \times n \end{array}$$



Slika 4.46 Matrica ranga 2 može se prikazati kao zbroj dvije matrice ranga 0



Slika 4.47 Rastav matrice ranga k

4.7.2 Rastav na singularne vrijednosti

Dekompozicija na singularne vrijednosti matrice $\mathbf{A}$, veličine $m \times n$, izražava matricu kao umnožak "jednostavnijih" matrica:

$$\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T$$

gdje je:

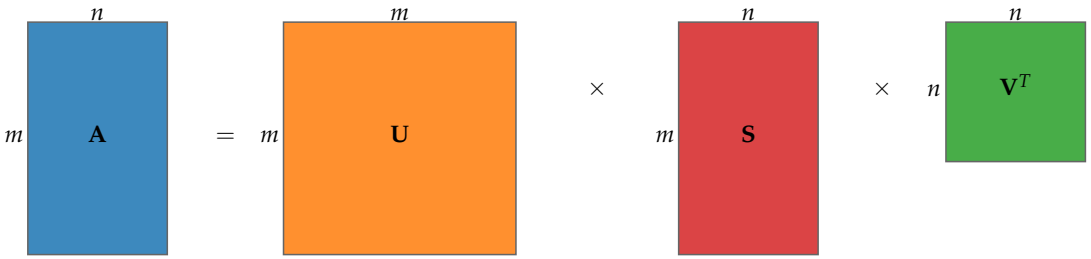
- $\mathbf{U}$ je $m \times m$ ortogonalna matrica, **lijevi singularni vektori**
- $\mathbf{V}$ je $n \times n$ ortogonalna matrica, **desni singularni vektori**

- **S** je $m \times n$ dijagonalna matrica, **singularne vrijednosti**, sa sortiranim nenegativnim vrijednostima

$$s_1 \geq s_2 \geq \ldots \geq s_{\min(m,n)} \geq 0$$

"Puni" rastav na singularne vrijednosti

$$\underset{m \times n}{\mathbf{A}} = \underset{m \times m}{\mathbf{U}} \times \underset{m \times n}{\mathbf{S}} \times \underset{n \times n}{\mathbf{V}^T}$$

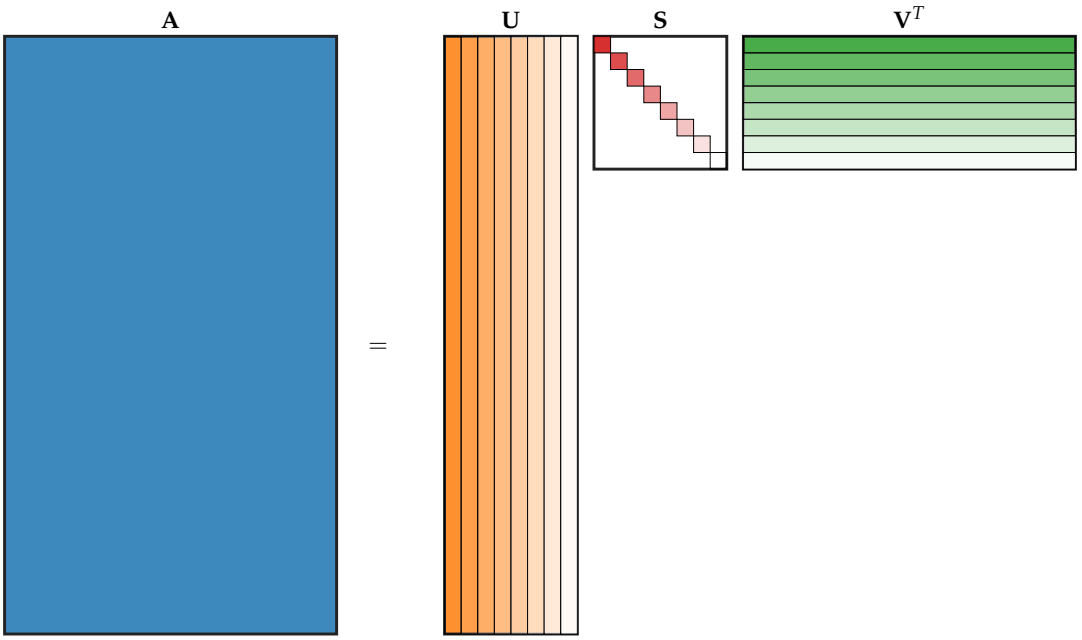


Slika 4.48 Rastav matrice na singularne vrijednosti

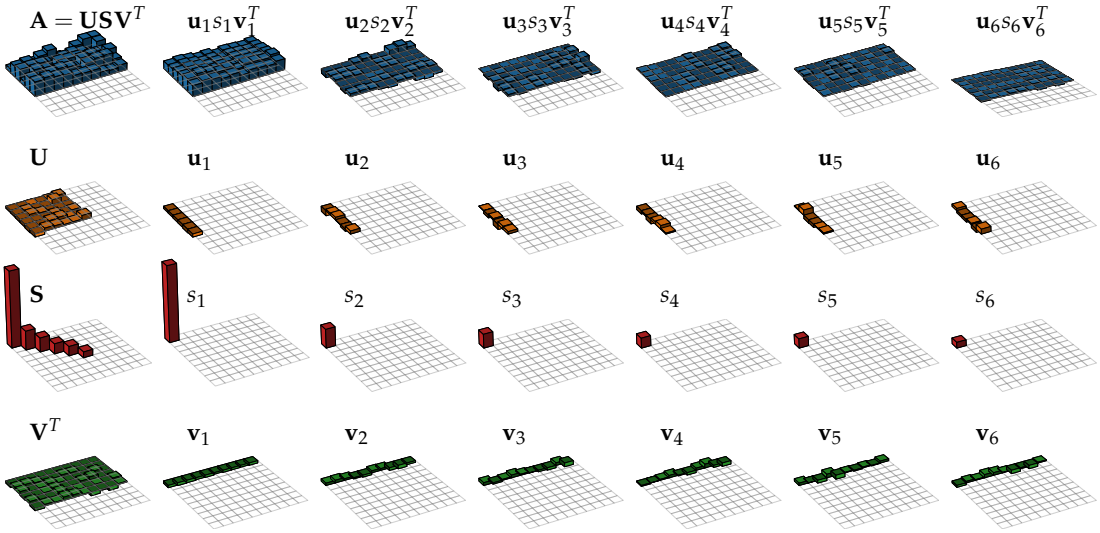
Ali **S** je dijagonalna matrica - ima puno nula!

$$\mathbf{A} = \sum_{i=1}^{\min(m,n)} s_i \cdot \mathbf{u}_i \mathbf{v}_i^T$$

Matrice mogu biti manjih dimenzija (zbog množenja s nulama matrice **S**):



Slika 4.49 Rastav matrice na singularne vrijednosti



Slika 4.50 Rastav matrice na singularne vrijednosti

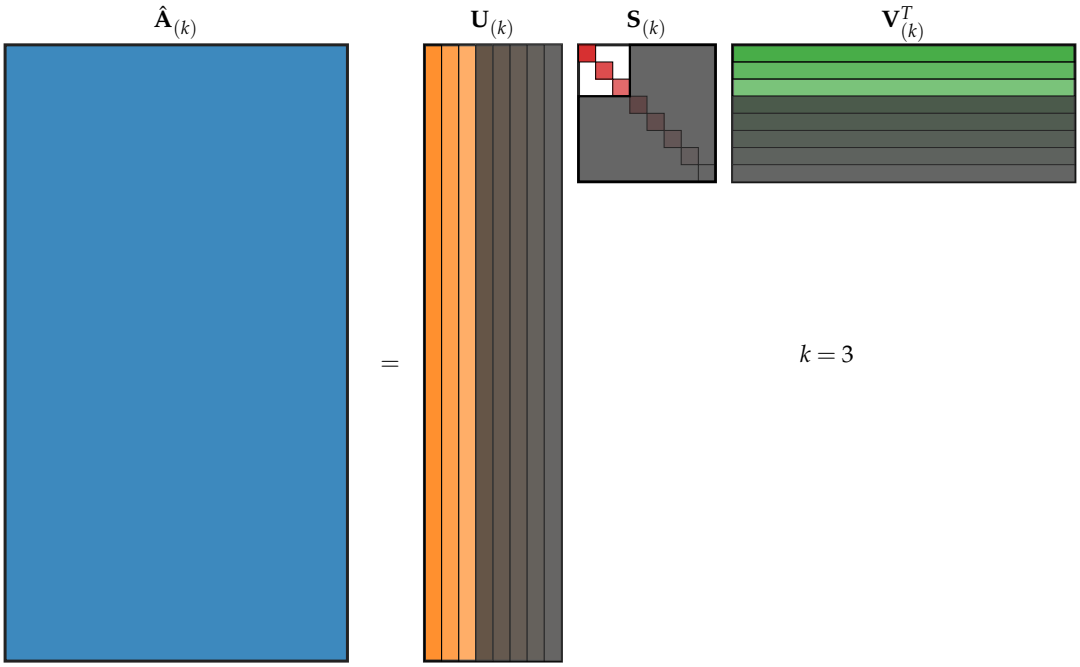
4.7.3 Aproksimacija manjim rangom

Pošto s_i opada s rastom i-a, viši rangovi sve manje doprinose u rekonstrukciji podataka A. Možemo napraviti aproksimaciju $\hat{\mathbf{A}}_{(k)}$ pomoću prvih k vlastitih vektora:

$$\mathbf{A} \approx \hat{\mathbf{A}}_{(k)} = \sum_{i=1}^{k} s_i \cdot \mathbf{u}_i \mathbf{v}_i^T$$

Zapisano s reduciranim matricama $\mathbf{U}_{(k)}$, $\mathbf{S}_{(k)}$ i $\mathbf{V}_{(k)}^T$:

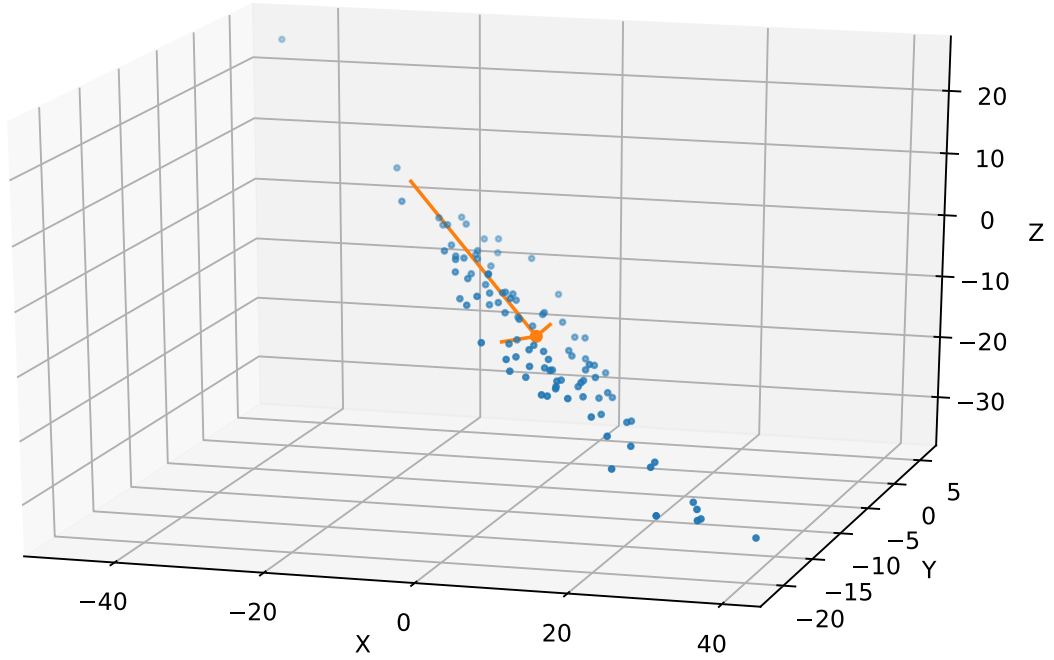
$$\hat{\mathbf{A}}_{(k)} = \mathbf{U}_{(k)} \mathbf{S}_{(k)} \mathbf{V}_{(k)}^T$$



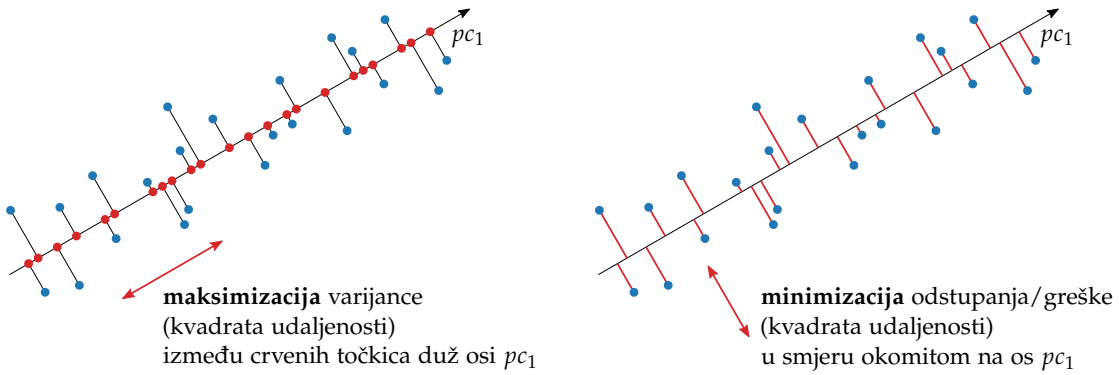
Slika 4.51 Rekonstrukcija matrice s reduciranim rangom

4.7.4 Analiza glavnih komponenata

- Slični rastav kao i SVD (PCA se može u potpunosti svesti na SVD)
- Dobiva se "jedna po jedna" glavna komponenta (prvih k)
- U nekim slučajevima može biti brže od SVD-a
- Drukčija interpretacija
- Temelji se na računanju vlastitih vrijednosti i vlastitih vektora ($\mathbf{Av} = \lambda\mathbf{v}$)



Slika 4.52 Glavne osi $pc1$, $pc2$ i $pc3$ (narančaste linije) su bolja parametrizacija podataka nego izvorne koordinate x, y i z.



Slika 4.53 Dva pogleda na određivanje glavnih osi

5. Modeliranje pomoću diferencijalnih jednadžbi

Obične diferencijalne jednadžbe – početni problem
Obične diferencijalne jednadžbe – rubni problem
Parcijalne diferencijalne jednadžbe
Metoda linija

5.1 Obične diferencijalne jednadžbe – početni problem

Jerko Škifić

Diferencijalne jednadžbe koje sadrže jednu ili više funkcija jedne neovisne varijable i njenih derivacija nazivaju se običnim diferencijalnim jednadžbama (ODJ).

Sustav N međusobno zavisnih običnih diferencijalnih jednadžbi y_i, $i = 0,1,\ldots,N-1$ se može zapisati u obliku

$$\frac{\mathrm{d}y_i}{\mathrm{d}x} = f_i(x, y_0, y_1, \ldots, y_{N-1}), \quad i = 0,1,\ldots,N-1 \tag{5.1}$$

gdje su funkcije f_i zadane. Ako se ograničimo na jednu funkciju, ODJ su oblika

$$y' = \frac{\mathrm{d}y}{\mathrm{d}x} = f(x,y). \tag{5.2}$$

Radi jednostavnosti, promotrimo jednostavnu ODJ čija funkcija f ovisi samo o x:

$$\frac{\mathrm{d}y}{\mathrm{d}x} = f(x), \tag{5.3}$$

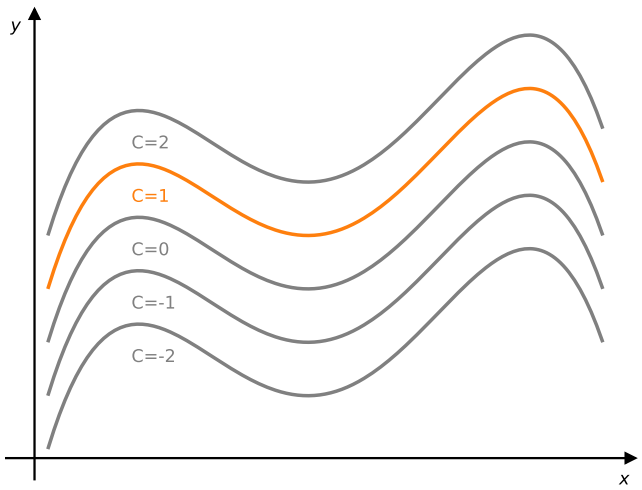
gdje je

$$f(x) = -2x^3 + 12x^2 - 20x + 8.5. \tag{5.4}$$

Potrebno je izračunati vrijednost funkcije y za $x = 4$. Analitičko rješenje se može izraziti izrazom

$$y = -\frac{1}{2}x^4 + 4x^3 - 10x^2 + 8.5x + C, \tag{5.5}$$

gdje je C konstanta integracije, što nam jasno govori da rješenje nije jedinstveno, odnosno da ovisi o vrijednosti konstante C (Slika 5.1).



Slika 5.1
Moguća rješenja za $\mathrm{d}y/\mathrm{d}x = -2x^3 + 12x^2 - 20x + 8.5$

Kako odabrati C? Potrebno je navesti još jedan, dodatni uvjet, koji se još naziva i **početnim uvjetom**. U našem, gornjem primjeru, odabrali smo $x = 0$ i $y = 1$. Skraćeno, kao početni uvjet zadali smo točku $T_0(0,1)$. Time je rješenje jedinstveno za $C = 1$.

Napomena
Početni uvjeti običnih diferencijalnih jednadžbi u inženjerskoj praksi imaju fizikalno značenje, npr. početna brzina tijela u vremenu $t = 0$.

Dakle, za rješavanje problema $y = y(x) =?$, potrebno je zadati ODJ i početni uvjet. Matematički zadatak određen ODJ (ili sustavom ODJ) i početnim uvjetom zove se **početni problem** i za njegovo rješavanje postoji više numeričkih metoda.

5.1.1 Runge-Kutta metode

Derivaciju iz izraza (5.2) svakako možemo približno izračunati kao:

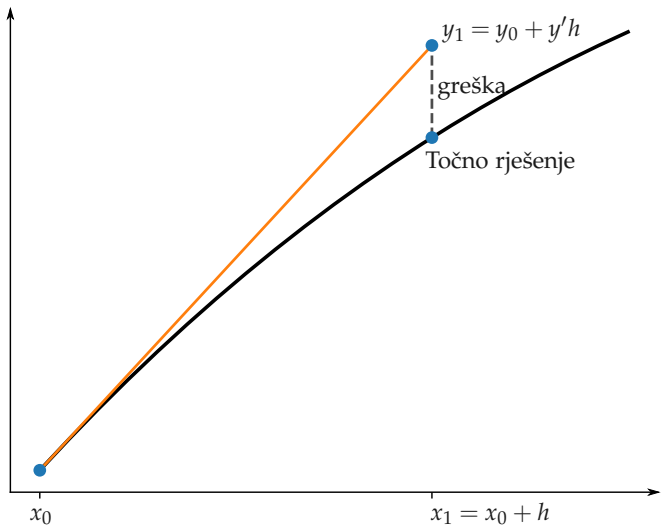
$$\frac{\mathrm{d}y}{\mathrm{d}x} \approx \frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0}, \tag{5.6}$$

gdje su y_0 i x_0 zadani iz početnog uvjeta. Sada ako sami izaberemo x_1 (kao x-koordinatu sljedeće točke koja leži na krivulji rješenja izvorne diferencijalne jednadžbe), možemo izračunati y_1:

$$y_1 = y_0 + \frac{\mathrm{d}y}{\mathrm{d}x}(x_1 - x_0) = y_0 + y'h, \tag{5.7}$$

gdje se $h = (x_1 - x_0)$ naziva još i **korakom** metode. Time smo numeričko rješavanje obične diferencijalne jednadžbe (5.2) sveli na aproksimaciju rješenja pravcem unutar koraka h.

Promatrajući ovaj izraz vidimo da je vrijednost y_1 moguće dobiti zbrojem **stare**, tj. **početne** vrijednosti y_0 i **promjene** (tj. prve derivacije) pomnožene korakom h. Ovdje derivacija y' ima značenje koeficijenta nagiba pravca, što se može i vidjeti na Slici 5.2.



Slika 5.2
Jedan korak numeričkog rješavanja jednadžbe $\mathrm{d}y/\mathrm{d}x = f(x,y)$ s početnim uvjetom $T_0(x_0,y_0)$ i korakom h

Usporedimo li izraz (5.7) s Taylorovim redom (kojim aproksimiramo funkciju u okolici neke točke),

$$y_1 = y_0 + y'h + \frac{y''}{2!}h^2 + \frac{y'''}{3!}h^3 \ldots, \tag{5.8}$$

vidjet ćemo da smo pri izračunu vrijednosti y_1 uzeli u obzir samo prva dva člana reda, čime nesumnjivo činimo grešku pri izračunu. Grešku možemo smanjiti na više načina:

- Uvođenjem dodatnih članova Taylorovog reda, što je u najmanju ruku neprak-tično,
- Smanjenjem veličine koraka h,

Veličina koraka h je ograničena računalnim resursima te ne može biti po volji malena. U pravilu, poželjno je odabrati što je moguće veći korak, da bi uz minimum računanja došli do rješenja potrebne točnosti.

Općenito, rješavanje običnih diferencijalnih jednadžbi Runge-Kutta metodama svodi se na izračunavanje niza točaka po sistemu

$$(\text{nova vrijednost}) = (\text{stara vrijednost}) + (\text{koeficijent pravca}) \times \text{korak},$$

odnosno

$$y_{i+1} = y_i + \phi h, \tag{5.9}$$

gdje je ϕ nagib pravca kojim duž koraka h aproksimiramo funkciju $y(x)$. Konkretne se metode razlikuju po načinu određivanja nagiba pravca ϕ i eventualno načinu određivanja veličine koraka h.

Eulerova metoda

Primijetimo da smo već u izrazu (5.7) zapravo iskoristili desnu stranu diferencijalne jednadžbe za procjenu nagiba pravca u točki x_i odnosno:

$$\phi = f(x_i, y_i), \tag{5.10}$$

čime smo dobili izraz za **Eulerovu metodu**:

$$y_{i+1} = y_i + f(x_i, y_i)h. \tag{5.11}$$

Ovdje se nova vrijednost izračunava evaluacijom nagiba pravca na *početku intervala* i ekstrapolira duž cijelog intervala h (Slika 5.2).

Python implementacija Eulerove metode dana je u programskom kodu 5.1:

Python kod 5.1 Eulerova metoda

```
import numpy as np

def df(x, y=0):
    # desna strana ODJ
    return -2*x**3 + 12*x**2 - 20*x + 8.5


x0, y0 = 0, 1   # početni uvjeti
x_end = 4       # krajnji x
n = 9           # broj koraka

xs, h = np.linspace(x0, x_end, n, retstep=True)
ys = np.zeros_like(xs)
for i in range(1, n):
    ys[i] = ys[i - 1] + df(xs[i - 1]) * h

print('y({})={}'.format(xs[-1], ys[-1]))
```

Greška Eulerove metode

Općenito, greška odrezivanja numeričkog rješenja ODJ sastoji se od dva dijela: **lokalne greške odrezivanja**, što je rezultat aproksimacije funkcije $y(x)$ tangentom tijekom jednog koraka, i **propagirajuće greške odrezivanja**, koja je rezultat aproksimacija tijekom prethodnih koraka. Kombinacija obje greške daje ukupnu, **globalnu grešku odrezivanja**.

Drugim riječima, s jedne strane javlja se greška zbog toga što traženu funkciju aproksimiramo tangentom na dužini $h > 0$ (lokalna greška, Slika 5.2), a s druge strane tako dobivena sljedeća točka (x_{i+1}, y_{i+1}) ne leži točno na traženoj funkciji $y(x)$, što znači da ni tangenta u novootkrivenoj točki nije jednaka pravoj tangenti, tj. $f(x_{i+1}, y_{i+1}) \neq y'(x_{i+1})$, što rezultira greškom koja propagira kroz sljedeće korake metode.

Procjena lokalne greške odrezivanja Eulerove metode može se odrediti relativno lako. Zamijenimo li y' sa $f(x_0, y_0)$ u izrazu (5.8), dobit ćemo

$$y_1 = y_0 + f(x_0, y_0)h + \frac{f'(x_0, y_0)}{2!}h^2 + \ldots + \frac{f_i^{(n-1)}(x_0, y_0)}{n!}h^n + O(h^{n+1}), \tag{5.12}$$

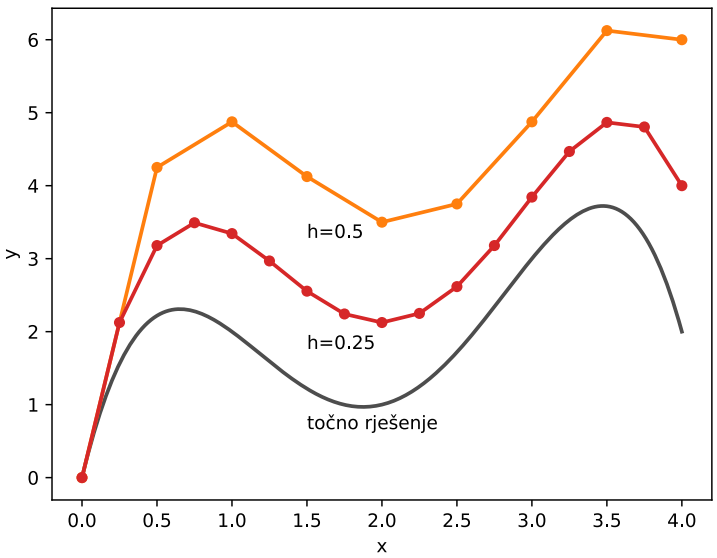
gdje prva dva člana desne strane odgovaraju Eulerovoj metodi, dok ostatak odgovara lokalnoj grešci metode:

$$E_t = \frac{f'(x_0, y_0)}{2!}h^2 + \ldots + \frac{f_i^{(n-1)}(x_0, y_0)}{n!}h^n + O(h^{n+1}). \tag{5.13}$$

Ovdje član $O(h^{n+1})$ predstavlja ostatak reda h^{n+1}. Viši članovi reda se mogu zanemariti za dovoljno mali korak h, čime se aproksimirana lokalna greška odrezivanja može izraziti kao:

$$E_t \approx \frac{f'(x_i, y_i)}{2!}h^2. \tag{5.14}$$

Dakle, lokalna greška odrezivanja Eulerove metode je proporcionalna kvadratu koraka h, a utjecaj veličine koraka h na točnost Eulerove metode je grafički ilustriran na Slici 5.3.



Slika 5.3
Usporedba točnog rješenja s rješenjima dobivenim Eulerovom metodom za $dy/dx = -2x^3 + 12x^2 - 20x + 8.5$, početnim uvjetom $T_0(0,0)$ i koracima $h = 0.25, 0.5$

Poboljšanja Eulerove metode

Izvor greške Eulerove metode proizlazi iz pretpostavke da se derivacija na početku intervala uzima kao **reprezentativna** duž cijelog intervala h. Točnost metode možemo poboljšati jednostavnim modifikacijama kojima ćemo postići primjereniji odabir reprezentativnog nagiba pravca na intervalu h.

Heunova metoda određuje reprezentativni nagib pravca ϕ iz izraza (5.9) kao linearnu kombinaciju nagiba pravaca na početku i kraju intervala (Slika 5.4). Kao i kod Eulerove metode, odredimo nagib na početku intervala:

$$y_i' = f(x_i, y_i) \tag{5.15}$$

i izračunamo preliminarni y na kraju intervala:

$$y_{i+1}^{(p)} = y_i + f(x_i, y_i)h. \tag{5.16}$$

Umjesto da se zadovoljimo dobivenim $y_{i+1}^{(p)}$, izračunajmo još i nagib na kraju intervala:

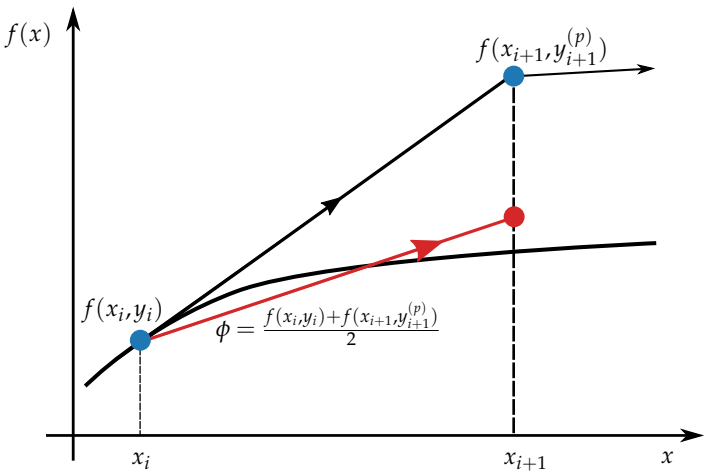
$$y_{i+1}' = f(x_{i+1}, y_{i+1}^{(p)}) \tag{5.17}$$

i kao reprezentativni nagib na cijelom intervalu uzmemo aritmetičku sredinu dva dobivena nagiba, onog na početku i onog na kraju intervala:

$$\phi = \frac{y_i' + y_{i+1}'}{2} = \frac{f(x_i, y_i) + f(x_{i+1}, y_{i+1}^{(p)})}{2} \tag{5.18}$$

Konačno, koristeći usrednjeni nagib ϕ, dobijemo y_{i+1} osnovnom Eulerovom metodom:

$$y_{i+1} = y_i + \phi h = y_i + \frac{f(x_i, y_i) + f(x_{i+1}, y_{i+1}^{(p)})}{2} h. \tag{5.19}$$



Slika 5.4
Primjer jednog koraka Heunove metode

Napomena
U slučajevima gdje derivacija y' ovisi samo o varijabli x, tj. rješava se jednadžba oblika $y' = f(x)$, nije potrebno računati $y_{i+i}^{(p)}$, već se reprezentativni nagib pravca ϕ može zapisati kao

$$\phi = \frac{f(x_i) + f(x_{i+1})}{2}, \tag{5.20}$$

odnosno vrijednost funkcije u koraku $i+1$ može se dobiti sljedećim izrazom:

$$y_{i+1} = y_i + \frac{f(x_i) + f(x_{i+1})}{2} h. \tag{5.21}$$

Metoda srednje točke kao reprezentativni nagib pravca ϕ iz izraza (5.9) uzima onaj dobiven na sredini intervala (Slika 5.5). Kao i kod Eulerove metode, odredimo nagib na početku:

$$y_i' = f(x_i, y_i) \tag{5.22}$$

i pomoću njega izračunamo y na sredini intervala:

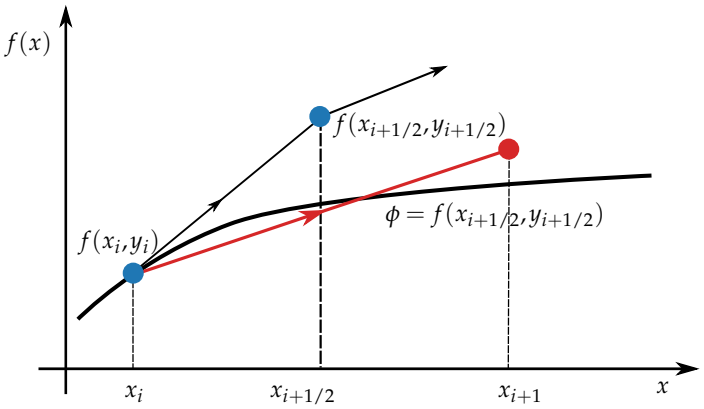
$$y_{i+1/2} = y_i + f(x_i, y_i)\frac{h}{2}. \tag{5.23}$$

Sada izračunamo nagib za dobivenu točku na sredini intervala:

$$\phi = f(x_{i+1/2}, y_{i+1/2}), \tag{5.24}$$

kojeg sada smatramo aproksimacijom nagiba pravca na čitavom intervalu h. Konačno, koristeći ovako dobiveni nagib ϕ, dobijemo sljedeću točku y_{i+1} Eulerovom metodom:

$$y_{i+1} = y_i + \phi h = y_i + f(x_{i+1/2}, y_{i+1/2})h \tag{5.25}$$



Slika 5.5
Primjer jednog koraka metode srednje točke

Runge-Kutta metode višeg reda

Do sada opisane metode (Eulerova, Heunova i metoda srednje točke) pripadaju puno većoj grupi metoda poznatim pod nazivom **Runge-Kutta metode**. One se općenito mogu zapisati već navedenim izrazom (5.9), gdje je $\phi = \phi(x_i, y_i, h)$ inkrementalna funkcija, koja predstavlja **reprezentativni nagib** na promatranom intervalu. Inkrementalna funkcija ϕ zapisuje se u obliku:

$$\phi(x_i, y_i, h) = a_1 k_1 + a_2 k_2 + \ldots + a_n k_n = \sum_{i=1}^{n} a_i k_i, \tag{5.26}$$

gdje su a_i konstante (uz uvjet $\sum_{i=1}^{n} a_i = 1$), n označava red metode, a nagibi se izračunavaju po sljedećem sistemu:

$$\begin{aligned}
k_1 &= f(x_i, y_i), \\
k_2 &= f(x_i + p_1 h, y_i + q_{1,1} k_1 h), \\
k_3 &= f(x_i + p_2 h, y_i + q_{2,1} k_1 h + q_{2,2} k_2 h), \\
&\ldots \\
k_n &= f(x_i + p_{n-1} h, y_i + q_{n-1,1} k_1 h + q_{n-1,2} k_2 h + \ldots + q_{n-1,n-1} k_{n-1} h),
\end{aligned} \tag{5.27}$$

gdje su p_i i $q_{i,j}$ konstante.

Iz ovakve definicije slijedi da je Eulerova metoda zapravo Runge-Kutta metoda prvog reda, uz $n = 1$ i $a_1 = 1$.

Metode drugog reda ($n = 2$) zapisuju se kao

$$y_{i+1} = y_i + (a_1 k_1 + a_2 k_2)\, h, \tag{5.28}$$

gdje se k_1 i k_2 dobivaju kako je prikazano u (5.27). Pažljivim odabirom koeficijenata možemo dobiti Heunovu metodu:

$$\begin{aligned} &a_1 = a_2 = \frac{1}{2}, \;\; p_1 = q_{1,1} = 1, \\ &k_1 = f(x_i, y_i), \;\; k_2 = f(x_i + h, y_i + k_1 h), \\ &y_{i+1} = y_i + \frac{k_1 + k_2}{2} h, \end{aligned} \tag{5.29}$$

i metodu srednje točke:

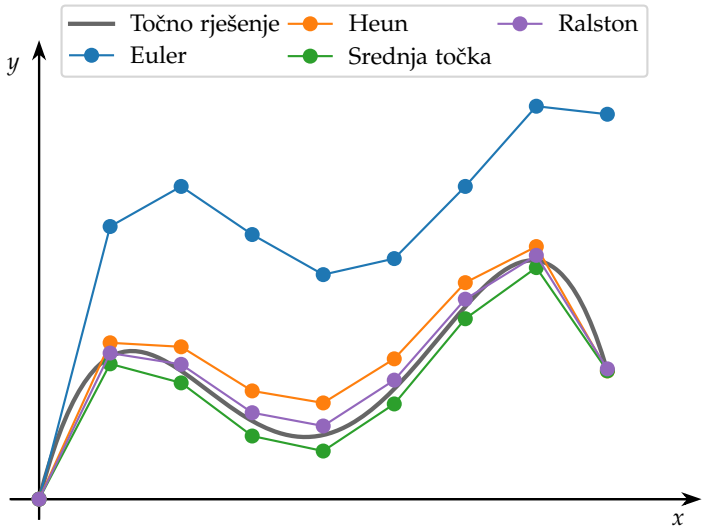
$$\begin{aligned} &a_1 = 0, \;\; a_2 = 1, \;\; p_1 = q_{1,1} = \frac{1}{2}, \\ &k_1 = f(x_i, y_i), \;\; k_2 = f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1 h), \\ &y_{i+1} = y_i + k_2 h. \end{aligned} \tag{5.30}$$

Ovdje valja navesti još jednu popularnu metodu drugog reda, koja je definirana na sljedeći način:

$$\begin{aligned} &a_1 = \frac{1}{3}, \;\; a_2 = \frac{2}{3}, \;\; p_1 = q_{11} = \frac{3}{4} \\ &k_1 = f(x_i, y_i), \;\; k_2 = f(x_i + \frac{3}{4}h, y_i + \frac{3}{4}k_1 h) \\ &y_{i+1} = y_i + (\frac{1}{3}k_1 + \frac{2}{3}k_2)h, \end{aligned} \tag{5.31}$$

a naziva se **Ralstonovom metodom**.

Na Slici 5.6 grafički je prikazana usporedba Eulerove metode s obrađenim Runge-Kutta metodama drugog reda za diferencijalnu jednadžbu $\mathrm{d}y/\mathrm{d}x = f(x)$, zadanu izrazom (5.4). Python implementacija navedenih Runge-Kutta metoda drugog reda dana je u programskom kodu 5.2.



Slika 5.6
Usporedba Eulerove, Heunove, Ralstonove metode i metode srednje točke za jednaki korak h

Python kod 5.2 Runge-Kutta metoda drugog reda

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

def df(x, y=0):
    # desna strana ODJ
    return -2*x**3 + 12*x**2 - 20*x + 8.5

x0, y0 = 0, 1   # početni uvjeti
x_end = 4       # krajnji x
n = 9           # broj koraka

xs, h = np.linspace(x0, x_end, n, retstep=True)
ys = np.zeros_like(xs)

# Heun
ys = np.zeros_like(xs)
ys[0] = y0
for i in range(1, 9):
    k1 = df(xs[i - 1])
    k2 = df(xs[i - 1] + h)
    ys[i] = ys[i - 1] + (k1 + k2) / 2 * h
plt.plot(xs, ys, label='Heun')

# Midpoint
ys = np.zeros_like(xs)
ys[0] = y0
for i in range(1, 9):
    k2 = df(xs[i - 1] + 0.5 * h)
    ys[i] = ys[i - 1] + k2 * h
plt.plot(xs, ys, label='Metoda srednje točke')

# Ralston
ys = np.zeros_like(xs)
ys[0] = y0
for i in range(1, 9):
    k1 = df(xs[i - 1])
    k2 = df(xs[i - 1] + 0.75 * h)
    ys[i] = ys[i - 1] + h * (k1 + 2 * k2) / 3.
plt.plot(xs, ys, label='Ralston')
```

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

Uzmemo li $n = 3$, dobivamo Runge-Kutta metode trećeg reda, koje karakteriziraju manje greške odrezivanja u odnosu na metode drugog reda. Od tih je najpopularnija varijanta:

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 4k_2 + k_3)h, \tag{5.32}$$

gdje je:

$$\begin{aligned} k_1 &= f(x_i, y_i), \\ k_2 &= f(x_i + \frac{1}{2}h, y_i + k_1\frac{1}{2}h), \\ k_3 &= f(x_i + h, y_i - k_1 h + 2k_2 h). \end{aligned} \tag{5.33}$$

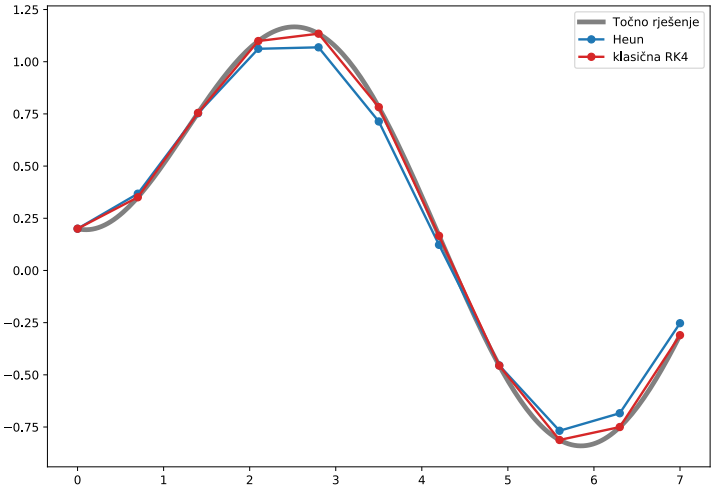
Od svih Runge-Kutta metoda, najčešće se upotrebljava metoda četvrtog reda ($n = 4$), i to sljedeća verzija:

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)h, \tag{5.34}$$

gdje je:

$$\begin{aligned} k_1 &= f(x_i, y_i), \\ k_2 &= f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1 h), \\ k_3 &= f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2 h), \\ k_4 &= f(x_i + h, y_i + k_3 h). \end{aligned} \tag{5.35}$$

Na Slici 5.7 grafički je prikazana usporedba Heunove metode s opisanom Runge-Kutta metodom četvrtog reda.



Slika 5.7
Usporedba točnog rješenja s rješenjima dobivenim Runge-Kutta metodom 4. reda i Heunovom metodom, za $\mathrm{d}y/\mathrm{d}x = -1/2y + \sin x$, uz početni uvjet $T_0(0, 0.2)$ i korak $h = 0.7$

Jednostavna implementacija navedenih Runge-Kutta metoda trećeg i četvrtog reda u programskom jeziku Python dana je u programskom kodu 5.3.

Python kod 5.3 Runge-Kutta metode trećeg i četvrtog reda

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

def df(x, y=0):
    # desna strana ODJ
    return -0.5*y + np.sin(x)

x0, y0 = 0, 0.2 # početni uvjeti
x_end = 7       # krajnji x
n = 9           # broj koraka

xs, h = np.linspace(x0, x_end, n, retstep=True)
ys = np.zeros_like(xs)

# RK 3. reda
ys = np.zeros_like(xs)
ys[0] = y0
for i in range(1, n):
    k1 = df(xs[i - 1], ys[i - 1])
    k2 = df(xs[i - 1] + 0.5*h, ys[i-1]+k1*0.5*h)
    k3 = df(xs[i - 1] + h, ys[i-1]+(-k1+2*k2) * h)
    ys[i] = ys[i - 1] + h * (k1 + 4 * k2 + k3) / 6.
plt.plot(xs, ys, label='RK3')

# RK 4. reda
ys = np.zeros_like(xs)
ys[0] = y0
for i in range(1, n):
    k1 = df(xs[i - 1], ys[i - 1])
    k2 = df(xs[i - 1] + 0.5 * h, ys[i - 1] + k1 * 0.5 * h)
    k3 = df(xs[i - 1] + 0.5 * h, ys[i - 1] + k2 * 0.5 * h)
    k4 = df(xs[i - 1] + h, ys[i - 1] + k3 * h)
    ys[i] = ys[i - 1] + h * (k1 + 2 * k2 + 2 * k3 + k4) / 6.
plt.plot(xs, ys, label='RK4')

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

5.1.2 Sustavi običnih diferencijalnih jednadžbi

Mnogi inženjerski problemi zahtijevaju rješavanje sustava običnih diferencijalnih jednadžbi (5.1), koji se može zapisati kao:

$$\begin{aligned}
\frac{\mathrm{d}y_1}{\mathrm{d}x} &= f_1(x, y_1, y_2, \ldots, y_n) \\
\frac{\mathrm{d}y_2}{\mathrm{d}x} &= f_2(x, y_1, y_2, \ldots, y_n) \\
&\ldots \\
\frac{dy_n}{dx} &= f_n(x, y_1, y_2, \ldots, y_n),
\end{aligned} \tag{5.36}$$

ili u vektorskom obliku:

$$\frac{\mathrm{d}}{\mathrm{d}x}\begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix} = \begin{pmatrix} f_1(x, y_1, y_2, \dots, y_n) \\ f_2(x, y_1, y_2, \dots, y_n) \\ \dots \\ f_n(x, y_1, y_2, \dots, y_n), \end{pmatrix} \tag{5.37}$$

odnosno:

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}x} = \mathbf{f}(x, \mathbf{y}) \tag{5.38}$$

Njihovo rješavanje se u osnovi ne razlikuje od rješavanja jedne jednadžbe, odnosno, rješava se svaka jednadžba zasebno za svaki korak.

Riješimo jednostavan model kretanja populacije u poznatom modelu grabežljivca i plijena (*predator prey*, ili *Lotka Volterra* model). Opisan je sustavom ODJ koji se često koristi za opisivanje dinamike bioloških sustava u kojima djeluju međusobno dvije vrste, grabežljivac i plijen.

Promjena populacije obje vrste opisuje se sljedećim izrazima:

$$\begin{aligned} \frac{dy_1}{dt} &= \alpha y_1 - \beta y_1 y_2 \\ \frac{dy_2}{dt} &= -\gamma y_2 + \delta y_1 y_2,, \end{aligned} \tag{5.39}$$

gdje su

- y_1 populacija plijena,
- y_2 populacija grabežljivaca,
- t vrijeme,
- α stopa rasta populacije plijena u odsutnosti grabežlivaca,
- β mortalitet plijena kao posljedice djelovanja grabežljivca,
- γ mortalitet grabežljivca čiji uzrok nije ovisan o populaciji plijena i
- δ faktor koji opisuje koliko uhvaćenog plijena rezultira novim grabezljivcima.

Rješenje modela Eulerovom metodom prikazano je u programskom kodu 5.4. Potrebno je za svaku jednadžbu sustava zadati posebnu funkciju te rješenja spremati u dva odvojena `numpy` vektora.

Python kod 5.4 Rješavanje modela grabežljivca i plijena pomoću Eulerove metode

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

# definicija parametara
a, b, c, d = 1., 0.1, 1.5, 0.75

# sustav ODJ
def df1(y_1, y_2, t=0):
    return a*y_1 - b*y_1*y_2

def df2(y_1, y_2, t=0):
    return -c*y_2 + d*b*y_1*y_2
```

```
t, h = np.linspace(0, 15,  1000, retstep=True)
y1 = np.zeros_like(t)
y2 = np.zeros_like(t)
y1[0], y2[0] = 10, 5  # početni uvjet

for i in range(1, t.size):
   y1[i] = y1[i-1] + df1(y1[i-1], y2[i-1])*h
   y2[i] = y2[i-1] + df2(y1[i-1], y2[i-1])*h

# crtanje
plt.plot(t, y1, label='plijen')
plt.plot(t, y2, label='grabežljivac')
plt.legend(loc='best')
plt.xlabel('vrijeme')
plt.ylabel('populacija')
plt.show()
```

Programsko rješenje navedeno u programskom kodu 5.4 je u najmanju ruku nepraktično. Nije teško zamisliti scenarij gdje je potrebno riješiti sustav s većim brojem ODJ. Tada je potrebno nadopisati dodatne definicije funkcija, deklarirati dodatne liste za rješenja, itd. Općenito, programski kod postaje kompliciraniji, teže čitljiv i podložniji greškama. Zapišemo li opisani model (5.39) u vektorskom obliku, vektori nepoznanica i funkcije gornjeg sustava su

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \quad \text{i} \quad \mathbf{f} = \begin{pmatrix} \alpha y_1 - \beta y_1 y_2 \\ -\gamma y_2 + \delta y_1 y_2 \end{pmatrix}, \tag{5.40}$$

čime smo definirali sustav ODJ u izrazu (5.38). Rješenje vektoriziranog modela Eulerovom metodom je prikazano u programskom kodu 5.5. Potrebno je primijetiti da je osnovna petlja Eulerove metode za vektoriziranu sustav ODJ svedena na jednu evaluaciju funkcije u odnosu na nevektoriziranu verziju (programski kod 5.4). Razlog tomu leži u činjenici što smo rješenja spremilli u dvodimenzionalno polje, gdje je broj stupaca jednak broju jednadžbi sustava. Također, funkcija `df` vraća `numpy.array` od dva elementa, čime programski kod postaje čitljiviji i kompaktniji.

Python kod 5.5 Rješavanje vektoriziranog modela grabežljivca i plijena pomoću Eulerove metode

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

# definicija parametara
a, b, c, d = 1., 0.1, 1.5, 0.75

# sustav ODJ
def df(y_t, t=0):
    return np.array([a*y_t[0] -   b*y_t[0]*y_t[1] ,
                     -c*y_t[1] + d*b*y_t[0]*y_t[1] ])

t, h = np.linspace(0, 15,  1000, retstep=True)
y = np.zeros([t.size, 2])
y[0,:] = np.array([10, 5]) # početni uvjet

# Eulerova metoda
for i in range(1, t.size):
   y[i,:] = y[i-1,:] + df(y[i-1,:])*h
```

```
# crtanje
plt.plot(t, y[:,0], label='plijen')
plt.plot(t, y[:,1], label='grabežljivac')
plt.legend(loc='best')
plt.xlabel('vrijeme')
plt.ylabel('populacija')
plt.show()
```

5.1.3 Rješavanje početnog problema pomoću SciPy modula

SciPy modul sadrži programsku funkciju `odeint` za rješavanje početnog problema običnih diferencijalnih jednadžbi, koja se nalazi u podmodulu `scipy.integrate`. Funkcija `odeint` koristi poznati FORTRAN rješavač *odepack*.

U svom osnovnom obliku, `odeint` se koristi na sljedeći način:

```
y = scipy.integrate.odeint(f, y0, x)
```

gdje je:

- `f` Programska funkcija `f(y,x)` koja definira običnu diferencijalnu jednadžbu, zadanu kao $dy/dx = f(y,x)$, mora biti napisana tako da vraća vektor evaluiranih vrijednosti svih jednadžbi sustava, u slučaju rješavanja sustava ODJ
- `y0` Početni uvjet, zadaju se vektorom (ili listom) vrijednosti
- `x` Lista točaka za koje se traži rješenje i
- `y` Lista rješenja za pripadajuće `x` točke.

Napomena
Za razliku od zapisa korištenog u matematičkim izrazima u tekstu, `odeint` zahtijeva da programska funkcija koja izračunava desnu stranu diferencijalne jednadžbe $dy/dx = f$ bude zadana na način da joj u argumentima ide prvo nezavisna varijabla, a nakon toga vektor zavisnih varijabli, tj. $f = f(\mathbf{y},x)$. U slučaju nepoštivanja ovog redoslijeda, `odeint` neće dati ispravno rješenje!

Kao primjer, u programskom kodu 5.6, prikazana je Python implementacija već opisanog modela kretanja populacije:

Python kod 5.6 Rješavanje modela grabežljivca i plijena pomoću SciPy funkcije `odeint`

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

# definicija parametara
a, b, c, d = 1, 0.1, 1.5, 0.75

# sustav ODJ
def df(y_t, t=0):
    return np.array([a*y_t[0] - b*y_t[0]*y_t[1] ,
                     -c*y_t[1] + d*b*y_t[0]*y_t[1] ])

t = np.linspace(0, 15,  1000)
y0 = np.array([10, 5]) # početni uvjet

# rješavanje
y = odeint(df, y0, t)
```

```
# crtanje
plt.figure(figsize=(10, 7))
plt.plot(t, y[:,0], label='plijen')
plt.plot(t, y[:,1], label='grabežljivac')
plt.legend(loc='best')
plt.xlabel('vrijeme')
plt.ylabel('populacija')
plt.show()
```

Alternativna, novija i preporučena SciPy funckija za rješavanje početnog problema je `solve_ivp`:

```
scipy.integrate.solve_ivp(f, t_span y0, x)
```

Važno je napomenuti da se argumenti funckije bitno razlikuju od `odeint` funkcije:

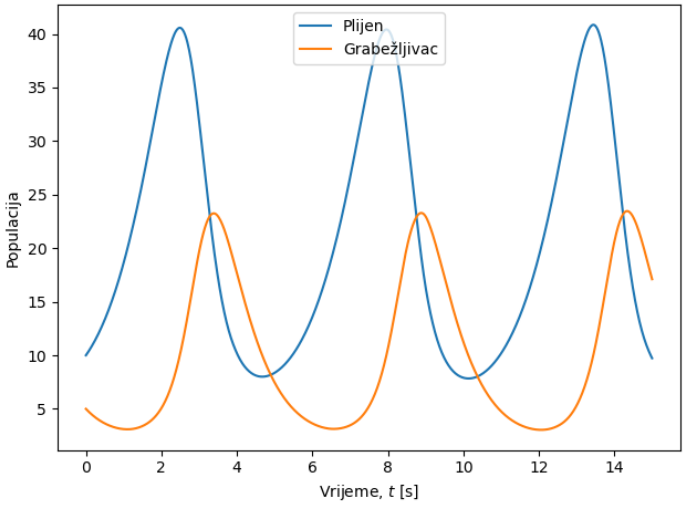
- `f` Programska funkcija `f(t,y)` koja definira običnu diferencijalnu jednadžbu, zadanu kao $dy/dt = f(t,y)$, mora biti napisana tako da vraća vektor evaluiranih vrijednosti svih jednadžbi sustava, u slučaju rješavanja sustava ODJ
- *t_span* raspon rješavanja (tuple koji sadrži početno i krajnje vrijeme)
- `y0` Početni uvjet, zadaju se vektorom (ili listom) vrijednosti
- `t_eval` Lista točaka (vremena) za koje se traži rješenje.

U Izvornom kodu 5.7 dan je primjer rješavanja istog modela kao i kod `odeint` primjera 5.6. Rezultat je prikazan na Slici 5.8.

Python kod 5.7 Rješavanje modela grabežljivca i plijena pomoću SciPy funkcije `solve_ivp`

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

# Definicija parametara
a, b, c, d = 1, 0.1, 1.5, 0.75

# Sustav ODJ
def df(t, y_t):
    return np.array([a*y_t[0] - b*y_t[0]*y_t[1] ,
                     -c*y_t[1] + d*b*y_t[0]*y_t[1] ])

t_start, t_end = 0, 15
t = np.linspace(t_start, t_end,  1000)
y0 = np.array([10, 5]) # početni uvjet

# Rješavanje
rjesenje = sp.integrate.solve_ivp(df,
                                  t_span=[t_start, t_end],
                                  y0=y0,
                                  t_eval=t)
print(f'{rjesenje.y.shape=}')

# Vizualizacija
fig, ax = plt.subplots(tight_layout=True)
plt.plot(rjesenje.t, rjesenje.y[0, :], label='Plijen')
plt.plot(rjesenje.t, rjesenje.y[1, :], label='Grabežljivac')
plt.legend(loc='best')
plt.xlabel('Vrijeme, $t$ [s]')
```

```
plt.ylabel('Populacija')
plt.show()
```



Slika 5.8
Rezultat *predator-prey* (grabežljivca i plijena) modela rješenog u Izvornom kodu 5.7

5.2 Obične diferencijalne jednadžbe – rubni problem

Stefan Ivić

Općenito, rubni problem je diferencijalna jednadžba promatrana na domeni uz određena ogranmičenja postavljena na rubu domene. Kod običnih diferencijalnih jednadžbi ta ograničenja se postavljaju na početku i kraju intervala na kojem se jednadžba rješava te se nazivaju **rubni uvjeti** (eng. *boundary conditions*).

Kao i kod početnog problema, broj ograničenja potrebnih za definiranje i jednoznačno rješavanje ovisi o redu obične diferencijalne jadnadžbe tj. o broju jednadžbi u sustavu diferencijalnih jednadžbi. Tako npr. diferencijalna jednadžba drugog reda

$$\frac{\mathrm{d}^2 y}{\mathrm{d}x^2} = f(x,y)$$

može imati zadane uvjete na rubovima domene $x \in [a,b]$:

$$\begin{aligned} y(a) &= \alpha \\ y(b) &= \beta \end{aligned} \tag{5.41}$$

gdje su α i β zadane skalarne vrijednosti. Rubni uvjet u kojem je zadana funkcijska vrijednost tražene funkcije $y(x)$, kao u (5.41), nazivamo **Dirichletov rubni uvjet**.

Uvjeti postavljeni na rubove domene mogu biti i drukčije formulirani. Zadavanje derivacije tražene funkcije, u rubnoj točki, naziva se **Neumannov rubni uvjet**. Na prethodnom primjeru, možemo postaviti Neumannov rubni uvjet u točki b:

$$\frac{\mathrm{d}y}{\mathrm{d}x}(b) = \beta. \tag{5.42}$$

Ograničenja zadana rubnim uvjetom mogu biti i složenija. Primjerice, može se kombinirati funkcijska vrijednost i derivacija tražane funkcije u rubnoj točki:

$$y(b) + \gamma \frac{\mathrm{d}y}{\mathrm{d}x}(b) = \beta.$$

Ovaj tip rubnog uvjeta nazivamo **Robinov rubni uvjet**.

> **Napomena**
> Rubni uvjeti definiraju ponašanje diferencijalne jednadžbe na rubu promatrane domene ali time ujedno i određuju rješenje na cijeloj domeni. U modelim gdje diferencijalna jednadžba opisuju razne fizikalne (ili druge) pojave i rubni uvjeti imaju fizikalnu interpretaciju i značenje koja zapravo slijedi iz samog modela. Primjerice, Neumannov rubni uvjet $\frac{\mathrm{d}T}{\mathrm{d}x} = 0$ kod provođenja topline kroz štap predstavlja izolaciju dok kod modela elastične linije grede $\frac{\mathrm{d}u}{\mathrm{d}x} = 0$ predstavlja uklještenje tj. spriječava rotaciju/nagib grede u rubnoj točki.

5.2.1 Metoda gađanja

Metoda gađanja (eng. *shooting method*) temelji se na riješvanju početnog problema gdje se pokušavaju postaviti nepoznati rubni uvjeti na lijevom rubu (početni uvjeti za riješavanje početnog problema) tako da se "pogode" zadani rubni uvjeti na desnom rubu.

Linearnu običnu diferencijalnu jednadžbu drugog reda koju rješavamo na intervalu $[0,l]$, možemo rastaviti na sustav n običnih diferencijalnih jednadžbi prvog reda:

$$\begin{aligned} y_1' &= f_1(x,y_1,y_2,\ldots,y_n) \\ y_2' &= f_2(x,y_1,y_2,\ldots,y_n) \\ &\vdots \\ y_n' &= f_3(x,y_1,y_2,\ldots,y_n). \end{aligned} \tag{5.43}$$

Za običnu diferencijalnu jednadžbu n-tog reda, odnosno za sustav (5.43) potrebno je postaviti ukupno n uvjeta $(y_1,y_2,\ldots,y_n)$ u točkama $x=0$ i/ili x=l. Ukoliko je su svih n uvjeta zadano u točki $x=0$ tada je riječ o početnom problemu, no ukoliko je bar jedan uvjet postavljen u $x=l$ tada govorimo o rubnom problemu. Međutim, rješavanju rubnog problema možemo pristupiti kao da je riječ o početnom problemu, ukoliko nepoznata početna stanja pretpostavimo, pa je određeno stanje u lijevom rubu $y_i(0)$ za svaki $i=1,\ldots,n$.

Radi jednostavnosti pretpostavimo da su na lijevom rubu zadani svi uvjeti osim y_j dok je na desnom rubu zadan samo jedan rubni uvjet y_k. Pošto kod početnog problema sva stanja ovise o početnom uvjetu, tako i konačno stanje tj. stanje u desnoj rubnoj točki (označeno kao $|_0$) ovisi o onom u lijevoj (označeno kao $|_l$):

$$\begin{aligned} y_1|_l &= g_1\left(y_1|_0, y_1|_0,\ldots, y_n|_0\right) \\ y_2|_l &= g_2\left(y_1|_0, y_1|_0,\ldots, y_n|_0\right) \\ &\vdots \\ y_n|_l &= g_n\left(y_1|_0, y_1|_0,\ldots, y_n|_0\right). \end{aligned} \tag{5.44}$$

Cilj metode gađanja je pronaći početno stanje koje će "pogoditi" krajnje stanje. Pošto su za dani primjer sva početna stanja poznata osim $y_j|_0$ a jedino stanje koje možemo "pogađati" (jer je jedino zadano) na desnom rubu je $y_k|_l$, nije potrebno rješavati sustav (5.44), već samo jednu nelinearnu jednadžbu:

$$y_k|_l = g_k\left(y_j|_0\right). \tag{5.45}$$

Treba napomenuti da funkcija g_k zapravo predstavlja cijelokupno rješavanje (5.43) kao početnog problema, uz poznato početno stanje koje se sastoji od zadanih y_i za $i=1,\ldots,n$, $i\neq j$ i pretpostavljenog $y_k|_0$.

Nelinearnu jednadžbu (5.45) možemo rješavati nekom od metoda predstavljenih u poglalvju 4.1.

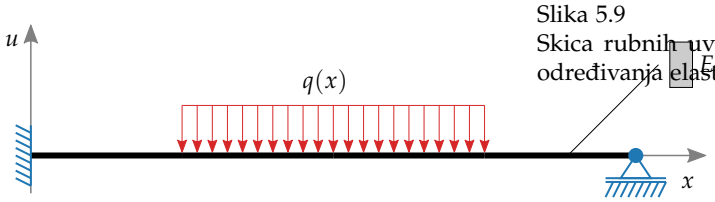
Napomena
Ukoliko je diferencijalna jednandžba koju rješavamo linearna, tada rješenje (5.45) možemo pronaći u samo jednom koraku metode sekante!

U općenitom slučaju, proizvoljan broj rubnih uvjeta može biti zadan na desnom rubu. Definirajmo skup L za koji su zadani $y_i|_0, \forall i \in L$ te skup D za koji su zadani $y_i|_l, \forall i \in D$. U tom slučaju možemo definirati sustav nelinearnih jednadžbi koje je potrebno riješiti:

$$y_k|_l = g_1\left(\ldots, y_j|_0, \ldots\right), \quad \forall k \in D, j=1,\ldots,n, j \notin L. \tag{5.46}$$

Primjer metode gađanja na rješavanju elastične linije grede

Raspisati uvod u problem

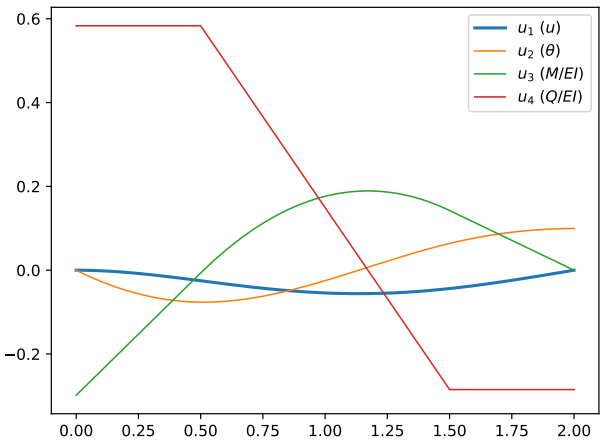


Slika 5.9
Skica rubnih uvjeta kod primjera određivanja elastične linije grede

Python kod 5.8 Lagrangeova interpolacija

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate as intg
import scipy.optimize as opt


"""
Euler-Bernoulli model grede

d^4 u / d x^4 = q(x) / (E * I)

d u_1 / dx = u_2
d u_2 / dx = u_3
d u_3 / dx = u_4
d u_4 / dx = q(x)

B.C.
u_1(0) = 0
u_2(0) = 0
u_1(L) = 0
u_3(L) = 0

Uklještenje na lijevom rubu (x=0) i zglob (bez pomaka) na desnom rubu (x=L)
"""

# Parametri
L = 2.0
E = 270e9
I = 0.02 * 0.04**4 / 12  # b*h^3/12

# Kontinuirano opterećenje N/m
def q(x):
    if 0.5 <= x <= 1.5:
        return -1000.0
    else:
        return 0.0

# Diferencijalna jednadžba
def du_dx(u, x):
    u1, u2, u3, u4 = u

    du1_dx = u2
    du2_dx = u3
    du3_dx = u4
    du4_dx = q(x) / (E * I)
```

```
    dudx = [du1_dx, du2_dx, du3_dx, du4_dx]

    return dudx

# Diskretizacija
x = np.linspace(0, 2, 101)

# Poznati rubni uvjeti
u1_start = 0
u2_start = 0
u1_end = 0
u3_end = 0

# Rješenje diff. jed. ovisno o zadanim rubnim uvjetima na lijevom rubu
# (samo oni koji su inače nepozanti)
def sol(U_init, full_solution=False):
    u3_start, u4_start = U_init
    U = intg.odeint(du_dx, [u1_start, u2_start, u3_start, u4_start], x)

    if full_solution:
        # Vraća kompletno rješenje jednadžbe
        return U
    else:
        # Vraća odstupanje rješenja od zadanih rubnih uvjeta na desnom rubu
        return np.array([u1_end - U[-1,0], u3_end - U[-1,2]])

# Rješava sustav nelinearnih jednadžbi (sol)
# uz početno pretpostavljene u3(0)=0 i u4(0)=0
U_init = opt.fsolve(sol, [0, 0])
print(U_init) # Ispis pronađenih (u startu nepozantih) rubnih uvjeta na lijevom rubu

# Računanje kompletnog riješenja
U = sol(U_init, True)

# Vizualizacija rješenja
plt.plot(x, U[:,0], lw=2, label=r'$u_1$ ($u$)')          # progib (pomak)
plt.plot(x, U[:,1], lw=1, label=r'$u_2$ ($\theta$)')     # kut nagiba
plt.plot(x, U[:,2], lw=1, label=r'$u_3$ ($M/EI$)')       # moment savijanja
plt.plot(x, U[:,3], lw=1, label=r'$u_4$ ($Q/EI$)')       # poprečna sila
plt.legend()
```



Slika 5.10
Rješenje Euler-Bernoullijevog modela grede (Izvorni kod 5.8)

5.2.2 Metoda konačnih razlika

5.3 Parcijalne diferencijalne jednadžbe

Bojan Crnković

Fizikalne pojave i sustavi kod dinamike fluida, elektriciteta, magnetizma, mehanike, termodinamike, bioloških i kemijskih interakcija može se opisati parcijalnim diferencijalnim jednadžbama. U nekim slučajevima moguće je određene pojave opisati i običnim diferencijalnim jednadžbama.

Parcijalnim diferencijalnim jednadžbama opisuju se vremenske i prostorne promjene nekih veličina u zatvorenom sustavu. Upravo je to osnovni razlog zašto se javljaju tako često kod opisivanja dinamičkih sustava i pojava i zato je mnogim znanstvenicima i inženjerima važno poznavanje tehnika i metoda kojima se rješavaju.

5.3.1 Osnovni pojmovi

Općenito parcijalna diferencijalna jednadžba (PDJ) je jednadžba u kojoj se pojavljuju parcijalne derivacije nepoznate zavisne funkcije.

Definicija 5.3.1 — Parcijalna derivacija. Neka je $\Omega \subset \mathbb{R}^n$ otvoren. Funkcija $u : \Omega \to \mathbb{R}$ ima parcijalnu derivaciju po i-toj varijabli x_i u točki $\mathbf{x} = (x_1, \ldots, x_n) \in \Omega$ ako postoji limes

$$\lim_{\Delta x_i \to 0} \frac{u(\mathbf{x} + \Delta x_i \, \mathbf{e}_i) - u(\mathbf{x})}{\Delta x_i} = \frac{\partial u}{\partial x_i}(\mathbf{x}) = u_{x_i}(\mathbf{x})$$

gdje je $\mathbf{e}_i$ koordinatni jedinični vektor baze vektorskog prostora $\mathbb{R}^n$

Posebno za funkciju dvije varijable $u = u(x,y)$, parcijalne derivacije možemo definirati na sljedeći način

$$\lim_{\Delta x \to 0} \frac{u(x + \Delta x, y) - u(x,y)}{\Delta x} = \frac{\partial u}{\partial x}(x,y) = u_x(x,y) \tag{5.47}$$

i

$$\lim_{\Delta y \to 0} \frac{u(x, y + \Delta y) - u(x,y)}{\Delta y} = \frac{\partial u}{\partial y}(x,y) = u_y(x,y) \tag{5.48}$$

Parcijalna derivacija funkcije je funkcija koja može biti neprekidna pa tada ima smisla tražiti njene parcijalne derivacije. Koristit ćemo i oznaku za funkciju čije su sve parcijalne derivacije neprekidne odnosno C^1 funkcije.

Definicija 5.3.2 — C^1. Neka je $\Omega \subset \mathbb{R}^n$ otvoren. Funkcija $u : \Omega \to \mathbb{R}$ je klase $C^1(\Omega)$ ako je neprekidna na Ω i sve njene parcijalne derivacije prvog reda su neprekidne na Ω.

Parcijalnu derivaciju m-tog reda dobijemo uzastopnim parcijalnim derivacijama funkcije f m-puta gdje je $m = k_1 + \ldots + k_n$ i pišemo:

$$\frac{\partial^m u}{\partial x_1^{k_1} \ldots \partial x_n^{k_n}}(\mathbf{x})$$

Definicija 5.3.3 — C^k. Neka je $\Omega \subset \mathbb{R}^n$ otvoren. Funkcija $u : \Omega \to \mathbb{R}$ je klase $C^k(\Omega)$ ako je klase $C^{k-1}(\Omega)$ je neprekidna na Ω i sve njene parcijalne derivacije reda k su neprekidne na Ω.

Ako su druge parcijalne derivacije $\frac{\partial^2 u}{\partial x_i \partial x_j}$ neprekidne na Ω tada su one i jednake odnosno:

$$\frac{\partial^2 u}{\partial x_i \partial x_j}(\mathbf{x}) = \frac{\partial^2 u}{\partial x_j \partial x_i}(\mathbf{x}), \quad \forall \mathbf{x} \in \Omega$$

5.3.2 Klasifikacija

Parcijalne diferencijalne jednadžbe možemo klasificirati na različite načine. Osnovna podjela je prema redu jednadžbe.

Definicija 5.3.4 — Red jednadžbe. Parcijalna diferencijalna jednadžba reda k je jednadžba koja ovisi o nepoznatoj funkciji u i njenim parcijalnim derivacijama tako da je najviši red parcijalne derivacije u jednadžbi jednak k.

Primjer 5.3.1 Jednadžba

$$u_{xx} + u_{yy} = 0$$

je homogena parcijalna diferencijalna 2. reda. Najviši red derivacije je 2 i slobodni član koji ne ovisi o u je 0. Lako je provjeriti da su neka od rješenja te jednadžbe $u_1(x,y) = x + y$ i $u_2(x,y) = xy$, ali i bilo koja njihova linearna kombinacija.

Primjer (5.3.1) pripada linearnim PDJ jer nepoznata funkcija u i njezine derivacije pojavljuju linearno, s koeficijentima koji ovise samo o nezavisnim varijablama x i y. Često je korisno klasificirati i kvazilinearne i polulinearne PDJ. Kod polulinearnih član s najvišim redom derivacije je linearan s koeficijentima koji ovise o nezavisnim varijablama, a ostali članovi s nižim derivacijama mogu biti nelinearni. Kvazilinearne su linearne za najviše derivacije, ali koeficijenti mogu ovisiti o nižim derivacijama nezavisne varijable.

Primjer 5.3.2 Jednadžba

$$3u_{xxy} + uu_{yy} - u^2u_x + \cos(u) = 1$$

je nehomogena polulinearna parcijalna diferencijalna 3. reda jer je najviši red derivacije 3 i slobodni član koji ne ovisi o u je različit od nule. Lako je provjeriti da su neka od rješenja te jednadžbe $u_k(x,y) = 2k\pi$, $k \in \mathbb{Z}$.

U primjeru (5.3.1) se pojavljuju i nelinearni članovi poput člana u^2u_x, ali član najvećeg reda je linearan.

Naglasimo da polulinearne jednadžbe nisu linearne, nego nelinearne kod kojih je nelinearnost slabije izražena pa se metode za linearne PDJ mogu prilagoditi za rješavanje polulinearnih. Nelinearne jednadžbe su uglavnom preteške i danas nemamo dobrih alata i tehnika za njihovo sustavno rješavanje i razumijevanje.

5.3.3 Linearne PDJ drugog reda

Posebno su zanimljive linearne parcijalne diferencijalne jednadžbe drugog reda jer se ovaj tip jednadžbi često javlja u prirodnim i tehničkim znanostima. Neke od klasičnih jednadžbi su valna jednadžba, jednadžba provođenja topline i Laplaceova jednadžba.

Linearna PDJ drugog reda može se zapisati u općenitom obliku:

$$\sum_{i,j=1}^{n} a_{i,j}(\mathbf{x})u_{x_ix_j} + \sum_{i=1}^{n} b_i(\mathbf{x})u_{x_i} + c(\mathbf{x})u = d(\mathbf{x})$$

Uvedimo oznaku za linearni operator

$$L = \sum_{i,j=1}^{n} a_{i,j}(\mathbf{x})\frac{\partial^2}{\partial x_i \partial x_j} + \sum_{i=1}^{n} b_i(\mathbf{x})\frac{\partial}{\partial x_i} + c(\mathbf{x}). \tag{5.49}$$

Operator L je linearan jer vrijedi

$$L(\alpha u_1 + \beta u_2) = \alpha L(u_1) + \beta L(u_2).$$

Tada možemo linearna PDJ drugog reda pisati u sažetom obliku:

$$Lu = d(\mathbf{x}). \tag{5.50}$$

Zbog linearnosti jednadžbe, možemo koristi princip superpozicije odnosno ako su $u_i, i = 1,\ldots,n$ rješenja jednadžbe (5.50) tada vrijedi:

$$L\sum_i^n \alpha_i u_i = d(\mathbf{x})\sum_i^n \alpha_i. \tag{5.51}$$

U slučaju homogene linearne PDJ linearna kombinacija rješenja je također rješenje odnosno:

$$L\sum_i^n \alpha_i u_i = 0. \tag{5.52}$$

Princip superpozicije rješenja je važan prilikom rješavanja separacijom varijabli. Ovom metodom opće rješenje se zapisuje kao red $u = \sum_i^n \alpha_i u_i$, gdje su u_i rješenja homogene jednadžbe odnosno $Lu_i = 0$. Separacija varijabli zajedno i superpozicija rješenja se obično naziva Fourierovom metodom.

5.3.4 Linearne PDJ drugog reda s dvije nezavisne varijable

U slučaju dvije nezavisne varijable linearne PDJ možemo zapisati pomoću linearnog operatora

$$L = A\frac{\partial^2}{\partial x^2} + 2B\frac{\partial^2}{\partial x \partial y} + C\frac{\partial^2}{\partial y \partial y} + D\frac{\partial}{\partial x} + E\frac{\partial}{\partial y} + F. \tag{5.53}$$

Želimo odrediti opće rješenje problema

$$Lu = G, \tag{5.54}$$

gdje su $A = A(x,y)$, $B = B(x,y)$, $C = C(x,y)$, $D = D(x,y)$, $E = E(x,y)$, $F = F(x,y)$ i $G = G(x,y)$ funkcije koje ovise samo o nezavisnim varijablama x i y, $u = u(x,y)$ je nepoznata zavisna funkcija.

Glavni dio linearnog operatora čine članovi najvišeg reda

$$A\frac{\partial^2}{\partial x^2} + 2B\frac{\partial^2}{\partial x \partial y} + C\frac{\partial^2}{\partial y \partial y}. \tag{5.55}$$

O glavnom dijelu operatora će ovisiti općenito ponašanje rješenja PDJ pa s obzirom na svojstvene vrijednosti odnosno diskriminantu glavnog dijela linearnog operatora možemo klasificirati ove jednadžbe na sljedeći način.

Tip jednadžbe (5.53,5.54) određen je predznakom diskriminirate $\Delta(x,y) = B^2(x,y) - 4A(x,y)C(x,y)$. Parcijalna diferencijalna jednadžba (5.53,5.54) je

- eliptička u (x,y) za $\Delta(x,y) < 0$,
- parabolička u (x,y) za $\Delta(x,y) = 0$,
- hiperbolička u (x,y) za $\Delta(x,y) > 0$.

Ako $\Delta(x,y)$ na mijenja tip na cijelom skupu Ω tada kažemo da je eliptička, parabolička ili hiperbolička. Nazivi za ove tipove dolaze iz geometrije jer jednadžbe nalikuju na jednadžbe elipse, parabole i hiperbole.

Zbog raširene pojave jednadžbi ovakvog tipa u matematičkom modeliranju fizikalnih pojava, a time i u inženjerstvu, upoznat ćemo neke od numeričkih metoda koje se koriste pri rješavanju jednadžbi oblika (5.53,5.54). Numeričke se metode bitno razlikuju u zavisnosti o tipu jednadžbe.

Uvedimo sljedeće oznake koje se često koriste

$$\nabla = [\frac{\partial}{\partial x}, \frac{\partial}{\partial y}]^T$$

$$\nabla u = [\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}]^T = grad(u)$$

te također uvedimo oznaku za Laplaceov operator

$$\Delta = \nabla \cdot \nabla = \nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2},$$

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2},$$

Klasični predstavnici ovih klasa linearnih PDJ s dvije nezavisne varijable su

- Laplaceova jednadžba
$$\Delta u = 0$$
je eliptička jer je $\Delta < 0$. Laplaceove jednadžba opisuje potencijalno polje odnosno stacionarnu distribuciju polja.
- Jednadžba provođenja u
$$u_t - k\Delta u = 0,$$
gdje je konstanta $k > 0$ vodljivost materijala. Jendadžba je parabolička jer je $\Delta < 0$. Ovom jednadžbom se opisuju difuzijski procesi u prirodi.
- Valna jednadžba u
$$u_{tt} - c^2\Delta u = 0$$
je hiperbolička jer je $\Delta < 0$. Ovom jednadžbom se opisuje širenje valova kroz elastični medij s konačnom brzinom širenja.
- Tricomi jednadžba $u_{tt} + xu_{xx} = 0$ je eliptička za $x < 0$, parabolička za $x = O$ te hiperbolička za $x < O$.

Primjer 5.3.3 PDJ ne mora biti u jednoj od danih kategorija već se klasifikacija može mijenjati prostorno i vremenski. Tricomi jednadžba $u_{tt} + xu_{xx} = 0$ je eliptička za $x < 0$, parabolička za $x = O$ te hiperbolička za $x < O$.

5.3.5 Rubni i početni uvjeti

Kod svakog fizikalnog problema koji se opisuje PDJ ne interesiraju nas opća rješenja već partikularna rješenje na zadanoj domeni Ω na čijem su rubu $\partial\Omega$ propisani rubni uvjeti koje mora zadovoljiti partikularno rješenje $u(t,x)|_{\partial\Omega} = f(x,t,u,u_x,u_{xx},\ldots)$. Posebno, ako je $u(t,x)$ odnosno vremenski ovisno tada zadajemo početni uvjet odnosno $u(0,x)$ koje mora biti usklađeno s rubnim uvjetima.

Iako postoji veliki broj rubnih uvjeta koji se mogu koristiti, postoji relativno maleni broj koji se redovito koristi:

- Dirichletov uvjet, propisuje se funkcijska vrijednost na dijelu ruba domene $\partial\Omega$.

$$u = g(x,t) \qquad \text{na } \partial\Omega_1$$

- Neumannov uvjet, propisuje derivaciju u smjeru normale na rubu domene

$$\frac{\partial u}{\partial n} = g(x,t) \qquad \text{na } \partial\Omega_2$$

- Robinov rubni uvjet, na rubu propisuje

$$a\,u + b\,\frac{\partial u}{\partial n} = g \qquad \text{na } \partial\Omega_3$$

Rubni uvjeti mogu biti kombinirani, odnosno na jednom dijelu domene zavisna varijabla u može zadovoljavati Dirichletov uvjet u jednom jednom dijelu granice a Neumannov uvjet na ostatku domene.

Primjer 5.3.4 Na primjeru jednadžbe provođenja topline možemo demonstrirati zadavanje različitih uvjeta:

$$T_t = k\,T_{xx},$$

gdje je $x \in \Omega = [0,1] \subset \mathbb{R}$ i $t \in [0,\tau]$ gdje je koeficijent termalne vodljivosti $k = 0.5$. Ova PDJ može predstavljati model koji prati promjenu temperature u tankom štapu jedinične duljine.

U jednom kraju štapa $x = 0$ možemo staviti Dirichletov rubni uvjet na sljedeći način

$$T(t,0) = \sin(10t).$$

Ovakav rubni uvjet propisuje oscilaciju na početnoj točki štapa i ta vrijednost ne ovisi o drugim točkama i samoj PDJ.

Na drugom kraju štapa možemo zadati Neumannov uvjet izolacije odnosno:

$$\frac{\partial T}{\partial x}(t,1) = 0.$$

Vrijednost u desnom kraju ovisi o unutarnjim točkama domene i početnom uvjetu.

Da bi ova jednadžba bila zadana u potpunosti, moramo postaviti početni uvjet. Potrebno je obratiti pozornost na usklađenost rubnih i početnih uvjeta:

$$T(0,x) = x$$

Lako možemo provjeriti da su u lijevom kraju rubni uvjeti usklađeni te da u desnom kraju domene nije potrebno posebno namještati rubne probleme.

Dobro postavljen problem

Francuski matematičar Jacques Hadamard dao je karakterizaciju dobro postavljenog problema. On je smatrao da modeli kojima se opisuju fizikalne pojave moraju imati

1. rješenje
2. rješenje mora biti jedinstveno
3. rješenje mora neprekidno ovisiti o početnim i rubnim uvjetima i ostalim parametrima sustava.

Dobro postavljeni problemi su jednadžbe širenja topline ili Laplaceove ili Poissonova jednadžba koje se smatraju prirodno postavljenim problemima. Iako je problem dobro postavljen, on može biti loše uvjetovan odnosno, njegov kondicijski broj je visok. Kod loše postavljenih ili loše uvjetovanih problema, numeričke aproksimacije pravog rješenja mogu imati velike pogreške i često nije moguće postići konvergenciju ka pravom rješenju.

Loše postavljeni problemi su često inverzni problemi kod kojih je osjetljivost na pogreške ili perturbacije u rješenju velika.

Jedinstvenost nije uvijek osigurana kod nekih jednadžbi, ali u tom slučaju mora postojati jasan kriterij koji daje jedinstvenost rješenja. Taj kriterij je često neki fizikalno opravdani dodatni uvjet kao što je uvjet povećanja entropije.

5.3.6 Eliptičke PDJ

Laplaceova jednadžba

$$\Delta u = 0$$

je jednostavani i tipičan primjer eliptičke jednadžbe koja se pojavljuje u primjenama. Rješenje Laplacove jednadžbe se može opisati pomoću harmonijskih funkcije i javljaju se kao rješenja problema u teorijama koja se bave skalarnim i vektorskim poljima kao što je mehanika fluida, termodinamika i elektromagnetizam.

U termodinamici, Laplaceova jednadžba je stacionarno rješenje jednadžbe provođenja topline.

Osim ove homogene, često se pojavljuje i Poissonova jednadžba

$$\Delta u = f.$$

Teorem 5.3.1 Princip maksimuma Ako je $\Delta u = 0$ na Ω i $u \in C^2(\Omega) \cap C^1(\bar{\Omega})$ tada je

$$\min_{y\in\partial\Omega} u(y) \leq u(x) \leq \max_{y\in\partial\Omega} u(y), \quad x \in \Omega \tag{5.56}$$

Ako vrijedi $\Delta u \geq 0$ na Ω tada je u konstanta ili je

$$u(x) < \max_{y\in\partial\Omega} u(y), \quad x \in \Omega \tag{5.57}$$

Korolar 5.3.1 — Jedinstvenost. Dirichletov problem Laplaceove jednadžbe može imati najviše jedno rješenje.

Korolar 5.3.2 — Jedinstvenost. Rješenja Neumannovog problema za Laplaceovu jednadžbe se razlikuju na konstantu.

Teorem 5.3.2 — Sferna simetrija. Neka je $B(x,r) \subset \Omega$ kugla radijusa r oko x tada za rješenje Laplaceove jednadžbe vrijedi da je $u(x)$ jednako srednjoj vrijednosti na rješenja u kugli odnosno:

$$u(x) = \frac{1}{P(\partial B(x,r))nr^{n-1}} \int_{\partial B(x,r)} u\, d\sigma = \frac{1}{P(\partial B(x,r))r^n} \int_{B(x,r)} u\, dV, \tag{5.58}$$

gdje je $P(\partial B(x,r))$ površina sfere.

5.4 Metoda linija

Stefan Ivić

Metoda linija je opća tehnika za rješavanje parcijalnih diferencijalnih jednadžbi (PDE) koje obično koriste konačne razlike za aproksimaciju prostornih derivacija dok se u vremenu riješava kao sustav običnih diferencijalnih jednadžbi (po jedna za svaku točku prostorne diskretizacije).

Za ilustraciju i detaljnije pojašnjenje ove metode koristit će se primjena metode na problem nestacionarnog progrijavanja štapa. Parcijalna diferencijalna jednadžvba koja opisuje promjenu temperature duž osi štapa i u vremenu je:

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2} \tag{5.59}$$

Lijevi rub je izoliran i u toj točki vrijedi Neumannov rubni uvjet:

$$\left.\frac{\partial T}{\partial x}\right|_0 = 0 \tag{5.60}$$

Na desnom rubu zadana je konstantna temperatura i vrijedi Dirichleov rubni uvjet:

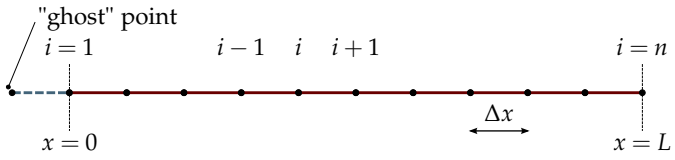
$$T|_L = 0 \tag{5.61}$$

.

Osnovna ideja metode linija je da se parcijalnu diferencijalnu jednadžbu (5.59) svede na sustav običnih diferencijalnih jednadžbi.

Diskretizacijom prostorne koordinate (Slika 5.11) te aproksimacijom konačnim razlikama, vrijedi:

$$\frac{\partial^2 T}{\partial x^2}(x) = \frac{T(x+\Delta x) - 2T(x) + T(x-\Delta x)}{\Delta x^2} \tag{5.62}$$



Slika 5.11
Prostorna diskretizacija metodom konačnih razlika

Jednadžba (5.62) vrijedi za svaku unutarnju diskretizacijsku točku i ovisna je o susjednim točkama:

$$\frac{\partial T_i}{\partial t} = D\frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2}, \quad i = 2,\ldots,n-1. \tag{5.63}$$

Na lijevom rubu, iz (5.60) vrijedi:

$$T_1 = T_0 \tag{5.64}$$

pa se može korigirati jednadžbu za prvu točku:

$$\frac{\partial T_1}{\partial t} = D\frac{T_2 - 2T_1 + T_0}{\Delta x^2} = D\frac{T_2 - T_1}{\Delta x^2}. \tag{5.65}$$

Desni rub se ne mjenja u vremenu, pa u krajnjoj točki vrijedi:

$$\frac{\partial T_n}{\partial t} = 0. \tag{5.66}$$

Jednadžbe (5.62), (5.65) i (5.66) tvore sustav običnih diferencijalnih jednadžbi koje se može rješiti prikladinim rješavačem.

Python kod 5.9 Primjer rješavanje provođenja topline kroz štap pomoću metode linija

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from scipy import *
from scipy.integrate import *
from matplotlib.pyplot import *
from matplotlib.animation import *

nx = 50
x = linspace(0, 1, nx)
A = diag(ones(nx)*(-2.0), k=0) + \
    diag(ones(nx-1)*1.0, k=1) + \
    diag(ones(nx-1)*1.0, k=-1)

"""
Resetiraj jednadžbu za prvu i zadnju točku
"""
A[0,:] = 0
A[-1,:] = 0

"""
Prva točka: izolacija dT0/dx = 0 ==> T_-1 = T_0
dT0/dt = k*(T_1 - 2*T_0 +T_-1)/dx^2
       = k*(T_1 - T_0)/dx^2
"""
A[0,0] = -1
A[0,1] = 1

"""
Zadnja točka: fiksna temperatura dT/dt = 0
"""
A[-1,-1] = 0


dx=0.1;
k=0.01;
nt = 11
t = linspace(0.0, 100., nt)

T0 = sin(x*pi)
print shape(dot(A,T0))

"""
Postavi sustav ODJ
"""
```

```
def dT_dt(T, t):
    dT = (k/dx**2.0)*dot(A,T)
    return dT

"""
Rješi sustav ODJ
"""
rez = odeint(dT_dt, T0, t)

"""
Crtanje i animacija rješenja
"""
fig1 = figure()
lineGeom, = plot([], [], 'r-', lw=2)
titl = title('')

def update_line(i):
    lineGeom.set_data(x, rez[i,:])
    titl.set_text('t=%f' % t[i])
    #fig1.savefig('img_%03d.png' % i)
    return

anim = FuncAnimation(fig1, update_line, nt, interval=1000, repeat_delay=0)
show()
```

6. Optimizacijske metode

Optimizacija funkcije jedne varijable
Linearno programiranje
Analiza u uvjetima neizvjesnosti
Određivanje najkraćeg puta
Pretraživanje uzorkom
Optimizacije pomoću scipy modula
Optimizacija rojem čestica

6.1 Optimizacija funkcije jedne varijable

Stefan Ivić

Zadatak jednodimenzionalne optimizacije je da za zadanu funkciju $f(x)$ pronađe optimum funkcije odnosno ekstrem, koji može biti minimum ili maksimum. S obzirom da se minimizacijski problem lako može pretvoriti u maksimizacijski i obrnuto, ako znamo da vrijedi:

$$\max f(\mathbf{x}) = -\min\left(-f(\mathbf{x})\right), \tag{6.1}$$

daljnji tekst u ovom poglavlju uglavnom će se baviti samo problemom minimizacije.

Kod definicije optimizacijskog problema, osim funkcije $f(x)$ potrebno je i zadati prostor pretraživanja Ω. To je područje na kojem tražimo optimalni x: $x \in \Omega$. Kod funkcija realnih varijabli, razlikujemo:

- Ograničeni prostor pretraživanja $\Omega = [x_a, x_b]$,
- Neograničeni prostor pretraživanja $\Omega = \mathbb{R}$.

Napomena
Prostor pretraživanja ovisi o promatranom problemu odnosno o tome što optimizacijska varijabla x predstavlja. Za lakše razumijevanje, navodimo nekoliko inženjerskih primjera:

- Ukoliko je varijabla x debljina lima, tada njenu vrijednost tražimo u smislenim tehnološki uvjetovanim granicama (npr. $x \in [0.1, 50]$ mm).
- Ako varijabla x predstavlja relativni tlak, čija vrijednost predstavlja razliku od nekog referentnog tlaka i može biti pozitivna ili negativna, tada npr. možemo postaviti prostor pretraživanja $x \in [-10, 10]$ bar.
- Ako je predmet optimizacije zakrivljenost nekog luka možemo formulirati problem tako da x predstavlja polumjer R nekog kružnog luka. S obzirom da polumjer ne može biti negativan, vrijedi $x \in [0, \infty]$.

Razlikujemo globalni i lokalni minimum, a analogno vrijedi i za maksimum (Slika 6.1). Globalni minimum, je točka za koju je funkcijska vrijednost najmanja na cijelom promatranom prostoru pretraživanja Ω, a može se definirati kao x^* za koji vrijedi:

$$f(x^*) \leq f(x),\ \forall x \in \Omega. \tag{6.2}$$

Lokalni minimum ima vrlo sličnu definiciju, ali funkcijska vrijednost x^* je najmanja samo u svojoj okolini $[x^* - \varepsilon, x^* + \varepsilon]$:

$$f(x^*) \leq f(x),\ \forall x \in [x^* - \varepsilon, x^* + \varepsilon] \tag{6.3}$$

gdje ta okolina, definirana sa ε, može (ali ne mora) biti vrlo mala. Isti uvjet je zadovoljen i za globalni minimum, pa je svaki globalni minimum ujedno i lokalni.

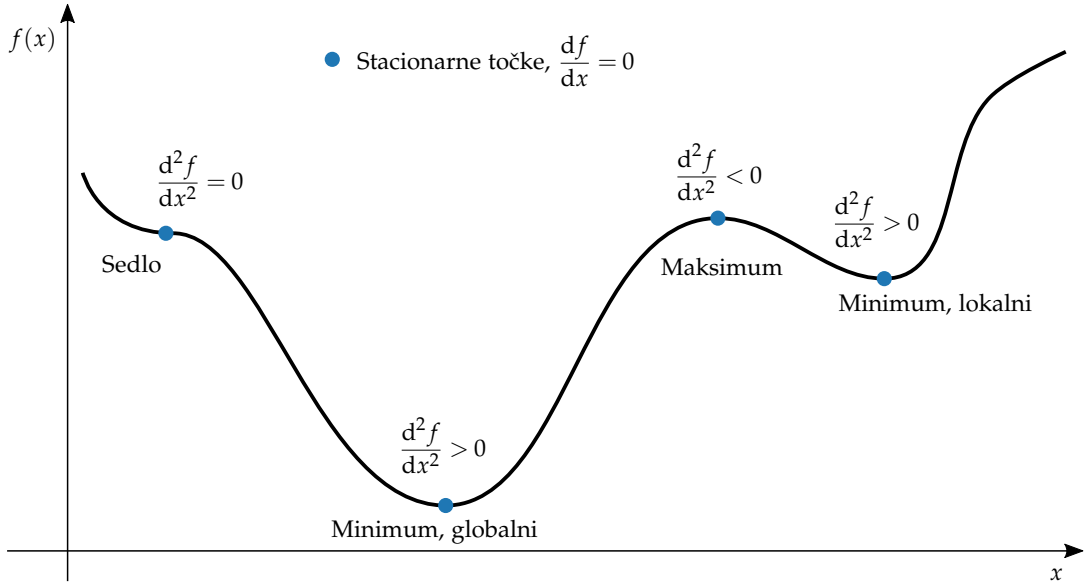
Iz matematičke analize poznato je da je vrijednost derivacije funkcije u ekstremu jednaka nuli. Međutim, to ne znači da je točka u kojoj je derivacija funkcije jednaka nuli nužno ekstrem. Takve točke nazivamo stacionarne točke, koje mogu biti sedla (točke infleksije) ili ekstremi (minimum ili maksimum).

Dakle, stacionarne točke su sve točke koje zadovoljavaju tzv. nužan uvjet ekstrema:

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = 0 \tag{6.4}$$

a pomoću druge derivacije možemo preciznije odrediti što stacionarne točke predstavljaju (dovoljan uvjet ekstrema):

$$\frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2} > 0 \implies \text{minimum funkcije,}$$
$$\frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2} = 0 \implies \text{potrebno testirati vrijednosti viših derivacija,}$$
$$\frac{\mathrm{d}^2 f(x)}{\mathrm{d}x^2} < 0 \implies \text{maksimum funkcije.}$$



Slika 6.1 Stacionarne točke

Upravo ove teoretske postavke mogu se iskoristiti u numeričkim metodama za optimizaciju 1D funkcija. No, često upotreba derivacije u optimizaciji predstavlja prejak zahtjev jer je derivacija nepoznata ili ju je "preskupo" približno izračunavati. Stoga se numeričke metode za 1D optimizaciju često oslanjaju na neke druge informacije, slično kao i kod rješavanja nelinearnih jednadžbi. Inicijalna točka x_0 za koju se pretpostavlja da je blizu rješenja (minimuma) x^* često se koristi u optimizacijskim metodama. Neke pak optimizacijske metode koriste unaprijed zadani prostor pretraživanja odnosno interval $[x_a, x_b]$ u kojem traže minimum. Pod pretpostavkom da je evaluacija funkcije računalno zahtjevan zadatak, cilj je uvijek da optimizacijska metoda u što manje koraka pronađe minimum ili točku u bliskoj okolini minimuma.

6.1.1 Metoda zlatnog reza

Kao i kod rješavanja nelinearnih jednadžbi, najjednostavnije metode za optimizaciju realne funkcije jedne varijable su metode ograđivanja. Jedna od najjednostavnijih

metoda ograđivanja je metoda zlatnog reza, koja može uspješno pronaći lokalni optimum neprekidne funkcije na zadanom intervalu $[x_a, x_b]$, a zahtjeva samo mogućnost evaluacije funkcije $f(x)$ za bilo koju točku na intervalu $[x_a, x_b]$.

Metoda zlatnog reza je iterativna metoda koja u svakoj iteraciji racionalno dijeli interval pretraživanja te ga potom sužava i na taj način postiže konvergenciju k ekstremu funkcije.

Promatrajmo minimizaciju funkcije $f(x)$ kako bi se definirala metoda zlatnog reza. Pošto na temelju rubnih točaka $[x_a, x_b]$ i jedne točke djeljenja ne možemo odrediti način sužavanja promatranog podintervala, potrebno je koristiti dvije točke djeljenja $[x_c, x_d]$. Tada se za sljedeću $(i+1)$ iteraciju može zadržati onaj podinterval koji "na sredini" sadrži onu točku djeljenja koja ima manju funkcijsku vrijednost odnosno:

$$[x_a^{(i+1)}, x_b^{(i+1)}] = \begin{cases} [x_a^{(i)}, x_d^{(i)}] & \text{ako je } f(x_c^{(i)}) < f(x_d^{(i)}) \\ [x_c^{(i)}, x_b^{(i)}] & \text{u suprotnom.} \end{cases} \tag{6.5}$$

Kod unimodalne funkcije (tj. funkcije koja ima samo jedan ekstrem – u ovom slučaju minimum) ovakav nam način sužavanja promatranog intervala omogućuje konvergenciju ka globalnom minimumu funkcije, dok kod multimodalnih funkcija (funkcije sa više ekstrema) ne garantira pronalazak globalnog minimuma, ali osigurava konvergenciju ka nekom od lokalnih minimuma.

Nakon što je definiran mehanizam ograđivanja tj. sužavanja intervala $[x_a^{(i)}, x_b^{(i)}]$, potrebno je odrediti način određivanja međutočaka $[x_c^{(i)}, x_d^{(i)}]$. Pošto postoji nebrojno mogućnosti kako da se podjeli promatrani interval, razmotrit će se neki zahtjevi koji utječu na određivanje međutočaka, a omogućavaju bržu konvergenciju metode. Ti zahtjevi su:

- U svim podjelama, nakon prvog koraka, iskoristiti postojeću međutočku te uvoditi samo jednu novu međutočku u svakoj sljedećoj iteraciji.
- Osigurati simetričnu raspodjelu međutočaka u svakoj iteraciji.
- Neka su omjeri razmaka između točaka isti u svakoj iteraciji.

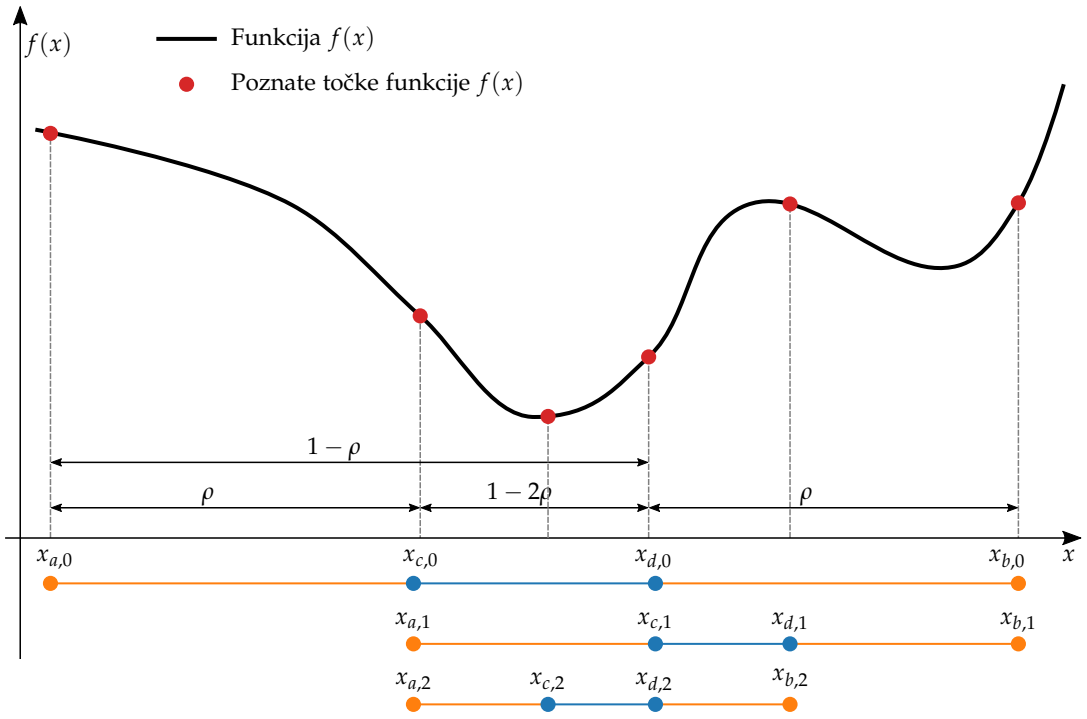
Pretpostavimo da je zadani početni interval $[x_a^{(0)}, x_b^{(0)}]$. Kako bi zadovoljili uvjet simetrije, središnje točke moraju jednako biti udaljene od bliže im rubne točke. Relativnu udaljenost središnjih točaka od bližih rubnih (narančaste linije na Slici 6.2) možemo dobiti dijeljenjem s obzirom na ukupnu širinu početnog intervala $(x_b^{(0)} - x_a^{(0)})$:

$$\rho = \frac{x_c^{(0)} - x_a^{(0)}}{x_b^{(0)} - x_a^{(0)}} = \frac{x_b^{(0)} - x_d^{(0)}}{x_b^{(0)} - x_a^{(0)}}. \tag{6.6}$$

Slijedi da relativni razmak između dvije srednje točke (plave linije na Slici 6.2) mora biti:

$$1 - 2\rho = \frac{x_d^{(0)} - x_c^{(0)}}{x_b^{(0)} - x_a^{(0)}}. \tag{6.7}$$

Ako pretpostavimo da je inicijalni interval ograđivanja definiran kao $[x_c^{(0)}, x_b^{(0)}]$, onda za funkciju na Slici 6.2 uzimamo nove granice $x_a^{(1)} = x_c^{(0)}$, $x_b^{(1)} = x_b^{(0)}$, a pošto



Slika 6.2 Dijeljenje intervala pretraživanja u metodi zlatnog reza

želimo zadržati postojeće točke, središnja točka je $x_c^{(1)} = x_d^{(0)}$. Nova točka dijeljenja $x_d^{(1)}$ nalaziti će se negdje između točaka $x_c^{(1)}$ i $x_b^{(1)}$.

Točke u prvoj iteraciji moraju zadovoljavati iste omjere udaljenosti središnje točke od bliže rubne i ukupne širine intervala. To možemo iskazati pomoću relativnih razmaka:

$$\frac{\rho}{1} = \frac{1-2\rho}{1-\rho} \tag{6.8}$$

što je zapravo kvadratna jednadžba

$$\rho^2 - 3\rho + 1 = 0. \tag{6.9}$$

čije je riješenje

$$\rho = \frac{3 \pm \sqrt{5}}{2} \tag{6.10}$$

odnosno $\rho_1 = 2.611348$ i $\rho_2 = 0.381966$. Pošto se tražena točka nalazi u promatranom intervalu, jedinična udaljenost ρ mora biti manja od 1, pa u obzir uzimamo samo rješenje $\rho = 0.381966$, koje predstavlja relativnu udaljenost srednje točke od bliže rubne točke intervala ograđivanja.

Napomena
Za relativnu udaljenost točke dijeljenja od udaljenije rubne točke $1-\rho = 0.618034$ vrijedi:

$$\varphi = \frac{1}{1-\rho} = \frac{1-\rho}{\rho} = \frac{1+\sqrt{5}}{2} = 1.618039887\ldots \tag{6.11}$$

gdje je φ tzv. konstanta zlatnog reza ili omjer zlatnog reza.

Konačno, možemo odrediti izraze za računanje središnjih točaka. Ukoliko je $f(x_c^{(i)}) < f(x_d^{(i)})$, tada za sljedeću iteraciju preuzimamo postojeće točke iz intervala $[x_a^{(i)}, x_d^{(i)}]$:

$$\begin{aligned} x_a^{(i+1)} &= x_a^{(i)} \\ x_b^{(i+1)} &= x_d^{(i)} \\ x_d^{(i+1)} &= x_c^{(i)} \end{aligned} \tag{6.12}$$

dok se nova lijeva središnja točka izračunava kao:

$$x_c^{(i+1)} = x_a^{(i+1)} + \rho \cdot (x_b^{(i+1)} - x_a^{(i+1)}). \tag{6.13}$$

Analogno, kada je $f(x_c^{(i)}) > f(x_d^{(i)})$ preuzimamo postojeće točke iz intervala $[x_b^{(i)}, x_d^{(i)}]$:

$$\begin{aligned} x_a^{(i+1)} &= x_c^{(i)} \\ x_b^{(i+1)} &= x_b^{(i)} \\ x_c^{(i+1)} &= x_d^{(i)} \end{aligned} \tag{6.14}$$

dok se nova desna središnja točka dobiva po izrazu:

$$x_d^{(i+1)} = x_b^{(i+1)} - \rho \cdot (x_b^{(i+1)} - x_a^{(i+1)}). \tag{6.15}$$

U svakoj iteraciji metode zlatnog reza duljina intervala pretraživanja se smanjuje za faktor $(1 - \rho)$. Vrlo se jednostavno može prikazati da je širina intervala ograđivanja nakon k koraka jednaka

$$x_b^{(k)} - x_b^{(k)} = (1 - \rho)^k \cdot (x_b^{(0)} - x_a^{(0)}), \tag{6.16}$$

što znači da je konvergencija metode zlatnog reza **linearna** i to sa konstantom $(1 - \rho) = 0.61803$.

6.1.2 Bisekcija

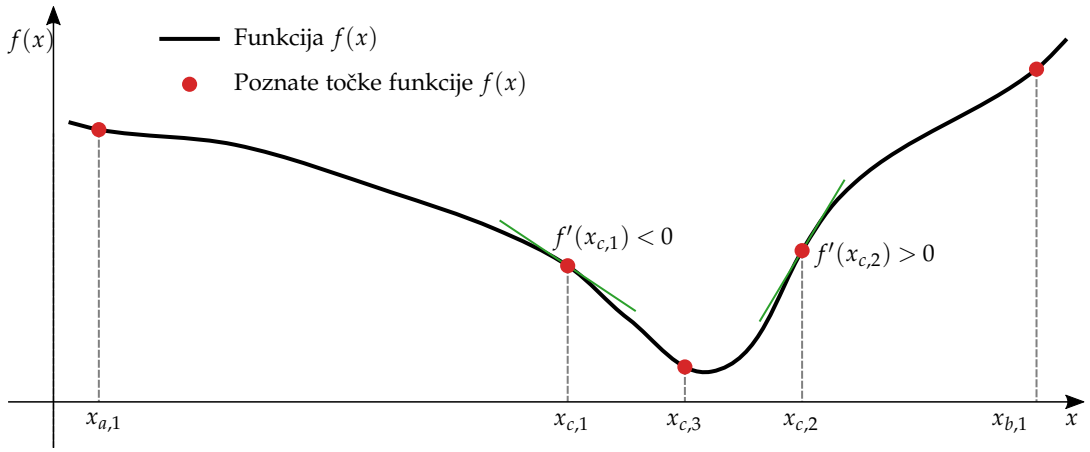
Jednostavna metoda za traženje nul-točke funkcije, kao što je metoda bisekcije (poglavlje 4.1.1), može se prilagoditi da rješava minimizacijski problem ukoliko se u nju "ugradi" korištenje derivacije funkcije.

Djeljenjem intervala $[x_a^{(i)}, x_b^{(i)}]$ na pola, točkom dijeljenja $x_c^{(i)}$, na raspolaganju imamo tri točke. Međutim, ne može se donijeti ispravna odluka o sužavanju promatranog intervala samo na temelju koordinata te tri točke. Korištenje točaka iz prethodnih iteracija (čime bi dobili ukupno četiri točke) svelo bi metodu na nešto slično metodi zlatnog reza (poglavlje 6.1.1), ali uz puno manju učinkovitost zbog neracionalnog dijeljenja intervala.

Međutim, poznavanjem derivacije funkcije u točki dijeljenja znamo stranu prema kojoj funkcija pada te možemo pouzdano odrediti koji od dva podintervala bisekcije odabrati:

$$[x_a^{(i+1)}, x_b^{(i+1)}] = \begin{cases} [x_a^{(i)}, x_c^{(i)}] & \text{za } f'(x_c^{(i)}) > 0 \\ [x_c^{(i)}, x_b^{(i)}] & \text{u suprotnom.} \end{cases} \tag{6.17}$$

Ovakva odluka glede sužavanja intervala ograđivanja je legitimna, pošto zbog derivacije funkcije pouzdano znamo u kojem smjeru funkcija pada, iz čega mora slijediti da u odabranom podintervalu postoje točke koje su niže od rubnih (Slika 6.3).



Slika 6.3 Korištenje metode bisekcije i derivacije funkcije za rješavanje problema minimizacije

6.1.3 Optimizacija pomoću modula SciPy

Modul `scipy.optimize` sadrži pregršt funkcija za rješavanje jednodimenzionalnih optimizacijskih problema (optimizaciju funkcije jedne varijable) ali i za višedimenzionalne probleme.

Osnovna funkcija namijenjena isključivo za 1D optimizaciju je `minimize_scalar`:

```
scipy.optimize.minimize_scalar(f, bracket=None)
```

gdje je `f` funkcija čiji se minimum traži, a neobavezni argument `bracket` su granice intervala ograđivanja $[x_a, x_b]$ zadane kao lista ili tri inicijalne točke $x_a < x_b < x_c$ isto zadane kao lista.

Naprednija varijanta namijenjena višedimenzionalnoj optimizaciji je funkcija `minimize` koja podržava desetak različitih optimizacijskih metoda koje se biraju argumentom `method`:

```
scipy.optimize.minimize(f, x0, method=None)
```

Funkcija `minimize` ne prima početni interval nego početno odnosno približno pretpostavljeno rješenje `x0`.

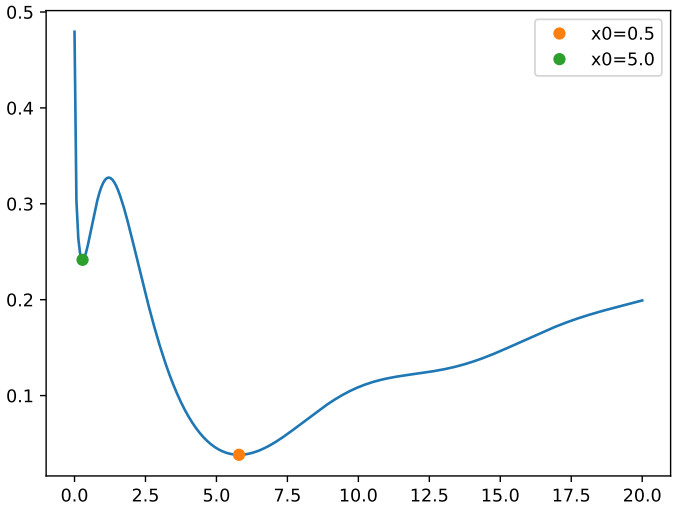
Primjer korištenja naredbi `minimize_scalar` i `minimize` prikazan je u izvornom kodu 6.1, a dobiveni rezultati vizualizirani su na Slici 6.4.

Python kod 6.1 Traženje minimuma funkcije pomoću funkcija iz modula SciPy

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize as opt

# Funkcija
def f(x):
    return np.sin(x - np.sqrt(x) + 0.5) / (x**2 + np.cos(x)) + x / 100.0

# Crtanje funkcije
x = np.linspace(0, 20, 300)
plt.plot(x, f(x))


# Traženje minimuma različitim metodama
```

```
print('minimize_scalar(f)')
min1 = opt.minimize_scalar(f) # bez definiranog intervala ograđivanja
print(min1)

print('\nminimize_scalar(f, [0, 0.5])')
min2 = opt.minimize_scalar(f, [0, 0.5]) # sa definiranim intervalom ograđivanja
print(min2)

print('\nminimize(f, 0.5)')
min3 = opt.minimize(f, 0.5) # u okolini točke x=0.5
print(min3)

print('\nminimize(f, 5.0)')
min4 = opt.minimize(f, 5.0) # u okolini točke x=5.0
print(min4)


# Crtanje minimuma
plt.plot(min1.x, min1.fun, 'o', label='x0=0.5')
plt.plot(min3.x, min3.fun, 'o', label='x0=5.0')
plt.legend()
```

```
minimize_scalar(f)
fun: 0.038254785470055071
nfev: 10
nit: 9
success: True
x: 5.7944748536953847

minimize_scalar(f, [0, 0.5])
fun: 0.24159953818268828
nfev: 21
nit: 20
success: True
x: 0.28288315393167446

minimize(f, 0.5)
fun: 0.24159953818271054
hess_inv: array([[ 0.75702895]])
jac: array([  2.47731805e-07])
message: 'Optimization terminated successfully.'
nfev: 21
nit: 5
njev: 7
status: 0
success: True
x: array([ 0.28288333])

minimize(f, 5.0)
fun: 0.03825478554541462
hess_inv: array([[ 50.21686664]])
jac: array([ -1.71968713e-06])
message: 'Optimization terminated successfully.'
nfev: 21
nit: 5
njev: 7
status: 0
success: True
x: array([ 5.7943872])
```



Slika 6.4
Traženje minimuma funkcije pomoću metoda iz SciPy modula

6.2 Linearno programiranje

Siniša Družeta

Linearno programiranje (LP) je matematička metoda pronalaska optimuma linearne funkcije više varijabli poštujući ograničenja zadana linearnim nejednakostima i jednakostima.

Linearno programiranje je razvijeno kao disciplina u 1940-ima, u početku motivirano zbog potrebe za rješavanje složenih problema planiranja u ratnim operacijama. Njegov razvoj ubrzano raste u poslijeratnom razdoblju, kad mnoge industrije pronalaze korisne primjene linearnog programiranja.

Iako je problem linearnog programiranja formulirao još Fourier, jedna od prvih praktičnih upotreba imao je ruski matematičar Kantorovič (1960) u pokušaju poboljšanja ekonomskih planiranja 1939. godine u SSSR-u.

6.2.1 Definicija LP problema

Problem linearnog programiranja može se razlučiti na **funkciju cilja** te na **ograničenja**. Funkcija cilja je linearna jednadžba n varijabli $(x_1, x_2, ..., x_n)$ definirana koeficijentima $c_1, c_2, ..., c_n$.

$$f(x_1, x_2, ..., x_n) = \sum_{i=1}^{n} c_i \cdot x_i \tag{6.18}$$

Jednadžba (6.18) može se zapisati u vektorskom obliku:

$$f(\mathbf{x}) = \mathbf{c} \cdot \mathbf{x} \tag{6.19}$$

gdje je $\mathbf{x} = [x_1\ x_2\ ...\ x_n]^{\mathrm{T}}$ vektor varijabli, a $\mathbf{c} = [c_1\ c_2\ ...\ c_n]$ vektor koeficijenata linearne jednadžbe. Vektor $\mathbf{x} \in \Omega \subset \mathbb{R}^n$ predstavlja optimizacijske varijable za koje se traži optimalno rješenje $\mathbf{x}^*$ za koje vrijedi

$$f(\mathbf{x}^*) \leq f(\mathbf{x}),\ \forall \mathbf{x} \in \Omega \tag{6.20}$$

Ograničenja problema se mogu zadati kao sustav linearnih nejednadžbi:

$$\begin{array}{rcl} a_{11} \cdot x_1 + a_{12} \cdot x_2 + \ldots + a_{1n} \cdot x_n & \leq & b_1 \\ a_{21} \cdot x_1 + a_{22} \cdot x_2 + \ldots + a_{2n} \cdot x_n & \leq & b_2 \\ \vdots & & \vdots \\ a_{m1} \cdot x_1 + a_{m2} \cdot x_2 + \ldots + a_{mn} \cdot x_n & \leq & b_m \end{array} \tag{6.21}$$

gdje je m broj ograničenja. Nejednadžbe se mogu zadavati sa različitim operatorima uspoređivanja ($\geq$ i $\leq$), a promjena operatora se može postići množenjem cijele nejednandžbe sa -1.

Sustav linearnih nejednadžbi (6.21) se može jednostavnije zapisati vektorski

$$\mathbf{A} \cdot \mathbf{x} \leq \mathbf{b} \tag{6.22}$$

Osim ograničenja zadanih nejednakostima, ograničenja se mogu zadati i linearnim jednadžbama:

$$\begin{aligned} d_{11} \cdot x_1 + d_{12} \cdot x_2 + \ldots + d_{1n} \cdot x_n &= e_1 \\ d_{21} \cdot x_1 + d_{22} \cdot x_2 + \ldots + d_{2n} \cdot x_n &= e_2 \\ \vdots & \vdots \\ d_{p1} \cdot x_1 + d_{p2} \cdot x_2 + \ldots + d_{pn} \cdot x_n &= e_p \end{aligned} \tag{6.23}$$

gdje je p broj ograničenja. U vektorskom zapisu sustav linearnih jednadžbi ograničenja (6.23) može se zapisati kao

$$\mathbf{A_{eq}} \cdot \mathbf{x} = \mathbf{b_{eq}} \tag{6.24}$$

gdje su d_{ij} elementi matrice sustava $\mathbf{A_{eq}}$, a e_i elementi vektora desne strane $\mathbf{b_{eq}}$.

Napomena
Ako je potrebno, ograničenja zadana jednadžbama mogu se uvijek svesti na nejednadžbe, jer općenito vrijedi da je zahtjevati $a = b$ isto kao i istovremeno zahtjevati $a \geq b$ i $a \leq b$.

Osobine i logiku LP problema ćemo detaljno pokazati na jednom jednostavnom primjeru, preuzetom iz Winston i Goldberg (2004) i prilagođenom.

Primjer 6.1 Proizvođač igračaka ima dva proizvoda: vojnike i vlakove. Troškovi rada u proizvodnji jednog vojnika iznose 14 €, a troškovi materijala 10 €. Kod vlaka, troškovi rada su 10 €, a troškovi materijala 9 €. Prodajna cijena vojnika je 27 €, a vlaka 21 €.

Proizvodnja ovih igračaka zahtjeva dvije vrste rada: grube radove i završne radove. Jedan vojnik zahtjeva 1 sat grubih radova i 2 sata završnih radova, dok je za jedan vlak potreban 1 sat grubih i 1 sat završnih radova. Proizvođač ima neograničen izvor materijala, ali u jednom tjednu ima kapacitet za samo 80 sati grubih radova i 100 sati završnih radova.

Potražnja na tržištu igračaka za vlakovima daleko prelazi proizvodne kapacitete proizvođača, ali se zato vojnika prodaje ne više od 40 komada tjedno. Proizvođač želi maksimizirati zaradu na tjednoj bazi. ■

Prvi korak u rješavanju LP problema je određivanje varijabli. Varijable su oni parametri problema na temelju kojih se može izraziti konačni cilj (u ovom slučaju profit) i uz to izreći sve važne tvrdnje, tj. zadana ograničenja odnosno veze između varijabli. Imajući to na umu, definirati ćemo varijable na sljedeći način:

x_1 - broj komada vojnika proizvedenih u jednom tjednu

x_2 - broj komada vlakova proizvedenih u jednom tjednu

Funkcija koju u ovom slučaju želimo maksimizirati predstavlja profit i glasi:

$$\begin{aligned} f(x_1, x_2) &= (\text{zarada od vojnika}) + (\text{zarada od vlakova}) \\ &= ((\text{novac od prodaje vojnika}) - (\text{troškovi proizvodnje vojnika})) \\ &\quad + ((\text{novac od prodaje vlakova}) - (\text{troškovi proizvodnje vlakova})) \\ &= (27 - (14 + 10)) \cdot x_1 + (21 - (10 + 9)) \cdot x_2 \end{aligned}$$

što ispada

$$f(\mathbf{x}) = 3x_1 + 2x_2.$$

Ovako formulirana funkcija cilja očito nema maksimum; uvećavanjem x_1 i x_2 dobijemo po želji veliku zaradu. (Sa druge strane, iz zadatka je očito da x_1 i x_2 ne mogu biti negativni, pa je najmanja moguća zarada $\min f(\mathbf{x}) = f(\mathbf{0}) = 0$.) Iz tog razloga ovaj zadatak ne bi imao smisla bez ograničenja, a njih ima nekoliko:

(1) maksimalno 80 sati grubih radova tjedno

(2) maksimalno 100 sati završnih radova tjedno

(3) maksimalno 40 komada vojnika prodano tjedno

Sva tri ograničenja možemo lako formulirati upotrebom varijabli x_1 i x_2. Za ograničenje (1) imamo:

$$\begin{aligned}(\text{sati grubih radova za vojnike}) + (\text{sati grubih radova za vlakove}) &\leq 80\\ 1x_1 + 1x_2 &\leq 80\end{aligned}$$

Analogno, za ograničenje (2) imamo:

$$\begin{aligned}(\text{sati završnih radova za vojnike}) + (\text{sati završnih radova za vlakove}) &\leq 100\\ 2x_1 + 1x_2 &\leq 100\end{aligned}$$

Konačno, ograničenje (3) zapisujemo kao:

$$x_1 \leq 40$$

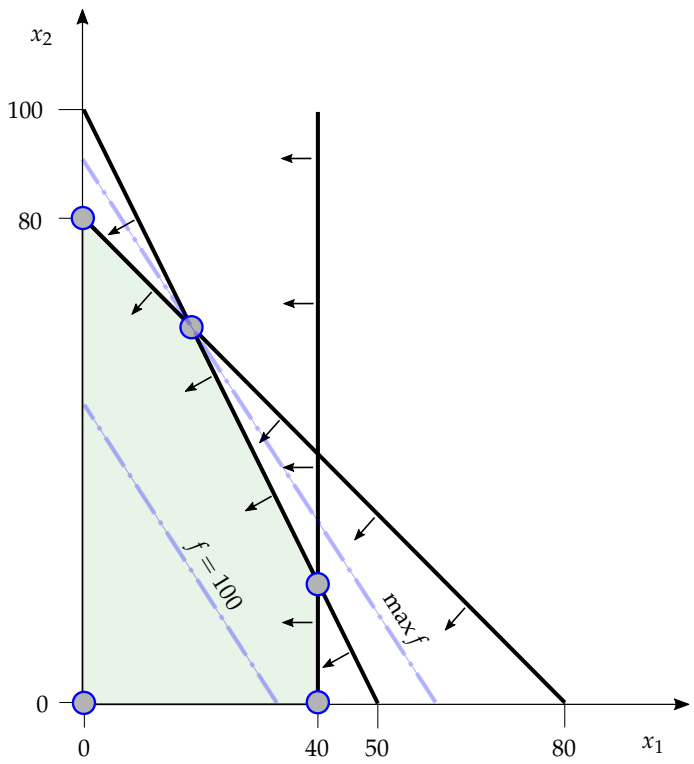
Dodamo li na ovo ograničenje nenegativnosti x_1 i x_2, dobijemo cijeli set ograničenja za promatrani linearni problem:

$$\begin{aligned} x_1 + x_2 &\leq 80\\ 2x_1 + x_2 &\leq 100\\ x_1 &\leq 40\\ x_1 &\geq 0\\ x_2 &\geq 0 \end{aligned}$$

S obzirom da se ovdje radi o prilično jednostavnom LP problemu sa samo dvije varijable, možemo ga čak i grafički riješiti. Za to je potrebno prvo na grafu označiti pravce koji označavaju ograničenja kako je to prikazano na Slici 6.5. Prostor unutar obojanog poligona označava prostor u kojem se nalaze kombinacije vrijednosti varijabli $\mathbf{x} = (x_1, x_2)$ koje ispunjavaju sva ograničenja.

Kod LP problema optimum je uvijek na jednom od vrhova poligona koji opisuje prostor mogućih rješenja. U našem primjeru to su istaknute točke na Slici 6.5. Pitanje je sada koja od ovih točaka daje maksimum funkcije cilja $f(\mathbf{x})$. Odgovor možemo dobiti tako da primijetimo da za neku izabranu zaradu Z funkcija cilja $f(\mathbf{x}) = 3x_1 + 2x_2$ postaje pravac $3x_1 + 2x_2 = Z$, koji se može nacrtati na grafu, kako je to i prikazano za $Z = 100$ na Slici 6.5. Sada možemo ovaj pravac jednostavno "translatirati" u smjeru većeg Z (gore-desno) dok ne ostane samo jedna točka na pravcu koja je ujedno i unutar prostora mogućih rješenja.

Na taj način možemo pronaći da se optimum nalazi u točki $(20, 60)$, što znači da je optimalno rješenje $x_1^* = 20, x_2^* = 60$, za koje se dobije zarada $f(\mathbf{x}^*) = 180$. Drugim riječima, ako proizvođač proizvodi 20 komada vojnika i 60 komada vlakova tjedno, imat će maksimalnu zaradu, koja iznosi 180 € tjedno.



Slika 6.5
Grafički prikaz LP problema proizvodnje igračaka

6.2.2 Rješavanje LP problema pomoću SciPy modula

Naravno, ako bi problem bio složeniji odnosno imao više od dvije varijable, svakako ga ne bismo grafički rješavali, nego bi koristili neki od gotovih računalnih rješavača. Za upotrebu gotovih rješavača najčešće je potrebno matrično zapisati funkciju cilja i ograničenja.

To znači da ako želimo riješiti naš primjer moramo pripremiti vektor koeficijenata funkcije cilja:

$$\mathbf{c} = \begin{bmatrix} 3 & 2 \end{bmatrix},$$

matricu sustava nejednakosti:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 1 & 0 \\ -1 & 0 \\ 0 & -1 \end{bmatrix},$$

te vektor desne strane sustava nejednakosti:

$$\mathbf{b} = \begin{bmatrix} 80 \\ 100 \\ 40 \\ 0 \\ 0 \end{bmatrix}.$$

Ovdje smo posljednja dva retka u sustavu nejednadžbi pomnožili sa (-1), da bi sve nejednadžbe koristile isti znak ($\leq$).

Ovdje ćemo za rješavanje linearnih problema koristiti `linprog` funkciju, koja se nalazi u modulu `scipy.optimize` i u osnovnoj varijanti se koristi na sljedeći način:

```
scipy.optimize.linprog(c, A_ub, b_ub)
```

gdje je `c` vektor koeficijenata funkcije cilja $\mathbf{c}$ (za $f(\mathbf{x}) = \mathbf{c} \cdot \mathbf{x}$), `A_ub` matrica sustava nejednakosti $\mathbf{A}$, a `b_ub` vektor desne strane sustava nejednakosti $\mathbf{b}$ (za $\mathbf{A} \cdot \mathbf{x} \leq \mathbf{b}$). Funkcija provodi minimizaciju, tj. podrazumijeva da se traži $\min f(\mathbf{x})$.

Kao rezultat, `linprog` vraća objekt sa više varijabli, s time da su za nas najvažnije varijable `success` u kojoj imamo informaciju da li je rješavač uspio pronaći rješenje i varijabla `x` u kojoj dobijemo vektor rješenja $\mathbf{x}$.

S obzirom da `linprog` provodi minimizaciju, a naš problem optimalne proizvodnje igračaka zahtjeva maksimizaciju, moramo funkciju cilja transformirati na način da je pomnožimo sa (-1), pri čemu je dovoljno da vektor $\mathbf{c}$ pomnožimo sa (-1). Podsjetimo se da u svakom trenutku možemo maksimizacijski problem pretvoriti u minimizacijski i obrnuto, ako znamo da vrijedi:

$$\max f(\mathbf{x}) = -\min\left(-f(\mathbf{x})\right).$$

Programski kod koji prikazuje rješavanje našeg zadatka pomoću funkcije `linprog` iz modula `scipy.optimize` dan je u nastavku (Python kod 6.2).

Python kod 6.2 Primjer rješavanja LP problema maksimizacije protoka u sustavu

```
import numpy as np
from scipy.optimize import linprog

# Giapetto's Woodcarving (Wayne L. Winston, Operation Research, 2004)
c = -np.array([3, 2]) # minus radi prelaska na minimizaciju
A = np.array([[1, 1],
              [2, 1],
              [1, 0],
              [-1, 0],
              [0, -1]])
b = np.array([80, 100, 40, 0, 0])
# optimum je -f(x) = -c*x, jer max(f(x)) = -min(-f(x))

solution = linprog(c, A_ub=A, b_ub=b)
if solution.success:
    print(solution.x)
    print(-np.dot(c, solution.x)) # minus radi povratka na maksimizaciju
else:
    print('rješavač nije došao do rješenja!')
```

6.2.3 Rješavanje LP problema pomoću PuLP modula

PuLP je Python modul namijenjen za rješavanje problema linearnog programiranja. Izvorni kod je dostupan na `https://github.com/coin-or/pulp` a dokumentacija na `https://pythonhosted.org/PuLP/#`.

Primjer korištenja modula PuLP dan je u izvornom kod 6.3, gdje se vidi jednostavnost i elegancija definiranja problem linearnog programiranja pomoću PuLP modula. Varijabla danog problema definira se pomoću naredbe `LpVariable` gdje se varijabli dodjeljuje ime te gornja i donja granica (eng. *bounds*). Pomoću `LpProblem` definira se

problem, odnosno da li je problem minimizacijski (`LpMinimize`) ili maksimizacijski (`LpMaximize`). Potom se na definirani problem nadodaju (pomoću operatora `+=`) ograničenja i cilj koji su definirani artimetičkim (linearnim!) vezama između optimizacijskih varijabli te pravilima nejednakosti/jednakosti u slučaju ograničenja. Konačno, linearni se problem riješava pomoću `solve()` metode (funkcija u `LpProblem` objektu).

Python kod 6.3 Primjer rješavanja LP problema pomoću modula PuLP

```
from pulp import *

# Varijable
x = LpVariable("x", 0, 3)
y = LpVariable("y", 0, 1)

# LP problem
prob = LpProblem("myProblem", LpMinimize)

# Cilj
prob += -4*x + y

# Ograničenja
prob += x + y <= 2
prob += x == y

# Rješavanje
status = prob.solve()

# Ispis
print(LpStatus[status])
print(value(x))
print(value(y))
```

6.2.4 Praktični primjeri LP problema

Primjer 6.2 Linearno programiranje se tradicionalno koristi za učinkovitu organizaciju posla, odnosno ekonomično zadovoljavanje potreba za radnom snagom. Ovdje ćemo na primjeru pokazati kako izgleda postupak rješavanja problema raspoređivanja posla (*work-scheduling problem*).

Razmotrimo problem poštanskog ureda (iz Winston i Goldberg (2004)) za koji postoje različite potrebe za radnom snagom za svaki dan u tjednu (Tablica 6.1). Pri tome propisi traže da svaki zaposlenik mora raditi pet uzastopnih radnih dana, nakon čega ima pravo na dva dana odmora. Ako su svi radnici zaposleni na puno radno vrijeme, potrebno je odrediti minimalni broj radnika koji mogu zadovoljiti potrebe ureda.

Tablica 6.1
Potrebe poštanskog ureda za radnom snagom

Dan	R. br.	Potrebno radnika
Ponedjeljak	1	17
Utorak	2	13
Srijeda	3	15
Četvrtak	4	19
Petak	5	14
Subota	6	16
Nedjelja	7	11

Kod problema koji se mogu rješavati linearnim programiranjem nerijetko je najteže uopće postaviti problem. To znači ispravno odrediti varijable odnosno odrediti koji parametri problema su temeljni, a koji su samo posljedice ovih prvih. U promatranom primjeru, prva ideja na koju ljudi obično dođu je da se varijable definiraju kao x_i - broj radnika koji rade na i-ti dan. U tom slučaju funkcija cilja glasi:

$$f(\mathbf{x}) = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7,$$

gdje se traži $\min f(\mathbf{x})$, a ograničenja koja slijede iz tablice 6.1 i uvjeta nenegativnosti varijabli glase:

$$\begin{aligned} x_1 &\geq 17 \\ x_2 &\geq 13 \\ x_3 &\geq 15 \\ x_4 &\geq 19 \\ x_5 &\geq 14 \\ x_6 &\geq 16 \\ x_7 &\geq 11 \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

Ovo očito nije dobro definiran linearni problem. Kao prvo, funkcija cilja ne daje ukupni broj zaposlenika, jer je npr. radnik koji radi od ponedjeljka do petka ukupno pet puta zastupljen u cjelokupnoj sumi, jednom u x_1, drugi put u x_2 i tako još u x_3, x_4 i x_5. Nadalje, iz rečenoga slijedi da su varijable međusobno zavisne, a njihove veze nisu definirane u sklopu sustava nejednakosti odnosno ograničenja. U konačnici, za ovako postavljen problem rješenje neizbježno mora biti $x_1 = 17$, $x_2 = 13$, $x_3 = 15$, itd., što znači da optimizacije tu praktički i nema.

Stoga je za ovaj tip problema ključno primijetiti da je, s obzirom da svi radnici u ciklusu 5+2 (tj. rade pet dana nakon čega imaju dva dana odmora), daleko bolje razmišljati ne o broju radnika koji rade na određeni dan, nego o broju radnika koji **počinju** raditi na određeni dan. Tako je x_1 broj radnika koji počinju raditi ponedjeljkom, rade do petka, a subotom i nedjeljom ne rade. Dalje, x_2 predstavlja broj radnika koji počinju raditi utorkom, rade do subote, a nedjeljom i ponedjeljkom ne rade, itd. do

x_7 koja predstavlja radnike koji počinju raditi nedjeljom, rade do četvrtka, a petkom i subotom ne rade.

Uz ovako definirane varijable, funkcija cilja ponovno (ali sada ispravno) glasi:

$$f(\mathbf{x}) = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7,$$

gdje oni radnici koji svoj radni ciklus počinju npr. srijedom (x_3) nisu istovremeno prisutni u preostalim varijablama (x_1, x_2, x_4, x_5, x_6, x_7).

Sada je jednostavno ustvrditi da npr. srijedom rade oni koji svoj radni ciklus počinju ponedjeljkom, utorkom, srijedom, subotom i nedjeljom, tj.

$$x_1 + x_2 + x_3 + x_6 + x_7 \geq 15.$$

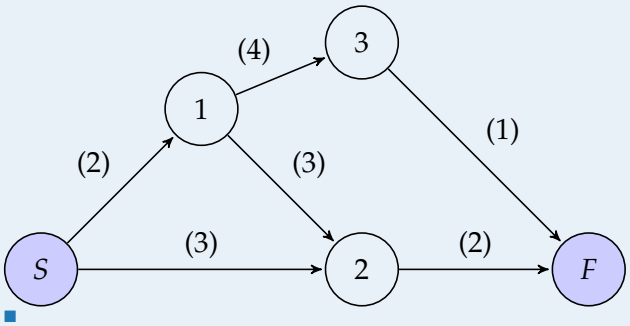
Shodno tome, cjeloviti sustav nejednakosti ograničenja za problem raspoređivanja posla u poštanskom uredu glasi:

$$\begin{aligned}
x_1 \quad &+x_4+x_5+x_6+x_7 \geq 17 \\
x_1+x_2 \quad &+x_5+x_6+x_7 \geq 13 \\
x_1+x_2+x_3 \quad &+x_6+x_7 \geq 15 \\
x_1+x_2+x_3+x_4 \quad &+x_7 \geq 19 \\
x_1+x_2+x_3+x_4+x_5 \quad &\geq 14 \\
x_2+x_3+x_4+x_5+x_6 \quad &\geq 16 \\
x_3+x_4+x_5+x_6+x_7 &\geq 11 \\
\mathbf{x} &\geq \mathbf{0}
\end{aligned}$$

Riješimo li ovaj linearni problem, pronaći ćemo da je najmanji broj zaposlenika $\min f(\mathbf{x}) = 22.33$, što se može ostvariti za $x_1 = 6.33$, $x_2 = 5$, $x_3 = 0.33$, $x_4 = 7.33$, $x_5 = 0$, $x_6 = 3.33$ i $x_7 = 0$. S obzirom da ljude ne možemo dijeliti na trećine, poslodavac bi tipično za svaku necjelobrojnu varijablu mogao usvojiti prvu veću cijelu vrijednost, no time se može bitno udaljiti od optimuma (za ovaj primjer broj radnika bi porastao na 25). Zbog toga se za rješavanje linearnih problema sa strogo cjelobrojnim varijablama koriste specijalni rješavači, kojima se ovdje nećemo baviti.

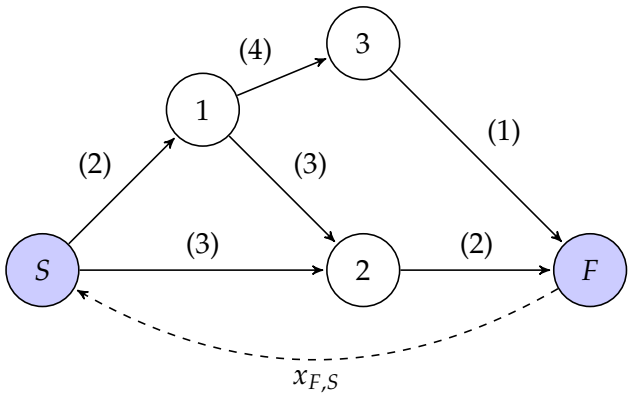
Primjer 6.3 Linearno programiranje se može koristiti i za skupinu problema koji se svode na problem maksimizacije protoka u sustavu (*maximum flow problem*). Kod takvih problema traži se način kojim se može kroz neki sustav odnosno mrežu (puteva, cijevi, komunikacijskih veza, ...) provesti što veću količinu nekog medija (robe, fluida, informacija, ...).

Uzmimo problem maksimizacije protoka na naftovodu (Winston i Goldberg (2004)). Naftovodna mreža se može pojednostavljeno prikazati kako je to dano na Slici 6.6, gdje točke označavaju naftovodne stanice (čvorišta), brojevi u zagradama označavaju kapacitet svake pojedine grane (u milijunima barela na sat), a točke S i F označavaju ulazni i izlazni čvor sustava. Tražimo optimalne protoke po granama koji će dati maksimalni protok kroz cijeli sustav, odnosno najbržu isporuku nafte od točke S do točke F.



Slika 6.6
Shema naftovodnog sustava

Prvi korak u definiranju linearnog problema je definicija varijabli. Zadatak traži da odredimo protoke po granama što znači da moramo koristiti varijable $x_{i,j}$ definirane kao protok kroz granu koja povezuje točku i i j. Da bi upotrebom ovako postavljenih varijabli mogli definirati funkciju cilja i cijeli linearni problem, moramo uvesti još i zamišljenu vezu povratnu vezu između izlazne i ulazne točke $x_{F,S}$, kako je to prikazano na Slici 6.7.



Slika 6.7
Shema naftovodnog sustava sa dodanom povratnom vezom

Sada možemo ustvrditi da funkcija cilja glasi:

$$f(\mathbf{x}) = x_{F,S},$$

gdje je $\mathbf{x} = [x_{S,1}\ x_{S,2}\ x_{1,2}\ x_{1,3}\ x_{2,F}\ x_{3,F}\ x_{F,S}]^{\mathrm{T}}$, a tražimo $\max f(\mathbf{x})$. Primijetimo da jedino varijabla $x_{F,S}$ uopće utječe na funkciju cilja $f(\mathbf{x})$. To naravno ne znači da ostale varijable nemaju utjecaja, no njihov utjecaj je prisutan posredno i to preko međusobnih veza između varijabli koje proizlaze iz "zakona očuvanja mase" za svaki čvor. Naime, u svakom čvoru mora vrijediti da ona količina medija koja je u njega ušla mora i iz njega izaći, tj. za i-ti čvor vrijedi:

$$\sum_j x_{i,j} = 0.$$

Dakle, ako se u sustavu nigdje ne zadržava ni ne gubi nafta, linearni problem

maksimizacije protoka kroz naftovodnu mrežu ima sljedeća ograničenja:

$$\begin{aligned}
x_{S,1} &\leq 2 \\
x_{S,2} &\leq 3 \\
x_{1,2} &\leq 3 \\
x_{1,3} &\leq 4 \\
x_{2,F} &\leq 2 \\
x_{3,F} &\leq 1 \\
x_{F,S} &= x_{S,1} + x_{S,2} \\
x_{S,1} &= x_{1,2} + x_{1,3} \\
x_{S,2} + x_{1,2} &= x_{2,F} \\
x_{1,3} &= x_{3,F} \\
x_{2,F} + x_{3,F} &= x_{F,S}
\end{aligned}$$

Napomena
Primijetimo da su ova ograničenja definirana sustavom nejednadžbi i sustavom jednadžbi. Kao takva, moraju se odvojeno zadavati u korištenom rješavaču. Ako rješavač inzistira na upotrebi nejednadžbi, moramo sve jednadžbe ograničenja prevesti u nejednadžbe ($a = b$ pretvoriti u $a \geq b$, $a \leq b$).

Ako ćemo za rješavanje ovog linearnog problema koristiti `linprog` paket iz modula `scipy.optimize` onda moramo problem definirati pomoću vektora i matrica. To znači da moramo pripremiti vektor koeficijenata funkcije cilja:

$$\mathbf{c} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

matricu sustava nejednakosti:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix},$$

vektor desne strane sustava nejednakosti:

$$\mathbf{b} = \begin{bmatrix} 2 \\ 3 \\ 3 \\ 4 \\ 2 \\ 1 \end{bmatrix},$$

kao i matricu sustava jednakosti:

$$\mathbf{A}_{eq} = \begin{bmatrix} -1 & -1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & -1 & -1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & -1 \end{bmatrix},$$

i pripradni vektor desne strane sustava jednakosti:

$$\mathbf{b}_{eq} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Osim toga potrebno je dodati informaciju da vrijedi i $\mathbf{x} \geq \mathbf{0}$, što se može učiniti proširivanjem sustava $\mathbf{A} \cdot \mathbf{x} \leq \mathbf{b}$ ili zadavanjem parametra `bounds` u rješavaču `linprog`.

Cjeloviti programski kod koji rješava ovaj zadatak upotrebom `linprog` rješavača iz modula `scipy.optimize` dan je u nastavku (Python kod 6.4). Ovdje koristimo funkciju `linprog` u malo naprednijoj varijanti:

```
scipy.optimize.linprog(c, A_ub, b_ub, A_eq, b_eq, bounds)
```

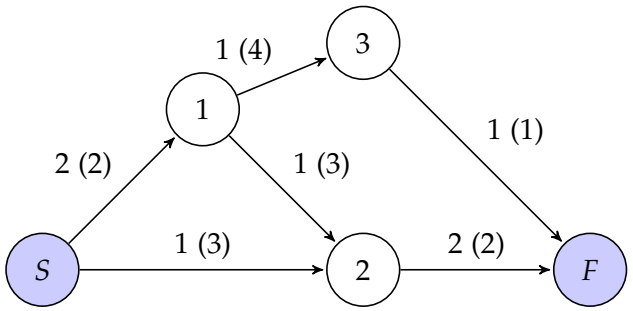
gdje, osim `c`, `A_ub` i `b_ub`, zadajemo i matricu sustava jednakosti `A_eq` (koja odgovara $\mathbf{A_{eq}}$) i vektor desne strane sustava jednakosti `b_eq` (koji odgovara $\mathbf{b_{eq}}$), gdje vrijedi $\mathbf{A_{eq}} \cdot \mathbf{x} = \mathbf{b_{eq}}$. Osim toga, ograničenje $\mathbf{x} \geq \mathbf{0}$ zadano je upotrebom varijable `bounds`.

Python kod 6.4 Primjer rješavanja LP problema maksimizacije protoka u sustavu

```
import numpy as np
from scipy.optimize import linprog

# Maximum flow (Sunco Oil) (Wayne L. Winston, Operation Research, 2004)
c = -np.array([0, 0, 0, 0, 0, 0, 1]) # minus radi prelaska na minimizaciju
A = np.array([[1, 0, 0, 0, 0, 0, 0],
              [0, 1, 0, 0, 0, 0, 0],
              [0, 0, 1, 0, 0, 0, 0],
              [0, 0, 0, 1, 0, 0, 0],
              [0, 0, 0, 0, 1, 0, 0],
              [0, 0, 0, 0, 0, 1, 0]])
b = np.array([2, 3, 3, 4, 2, 1])
Aeq = np.array([[-1, -1, 0, 0, 0, 0, 1],
                [1, 0, -1, -1, 0, 0, 0],
                [0, 1, 1, 0, -1, 0, 0],
                [0, 0, 0, 1, 0, -1, 0],
                [0, 0, 0, 0, 1, 1, -1]])
beq = np.zeros(5)
limit = [(0, np.inf)]*np.size(c)

solution = linprog(c, A_ub=A, b_ub=b, A_eq=Aeq, b_eq=beq, bounds=limit)
if solution.success:
    print(solution.x)
```

```
    print(-np.dot(c, solution.x)) # minus radi povratka na maksimizaciju
else:
    print('rješavač nije došao do rješenja!')
```

Jedno moguće rješenje ovog linearnog problema je $x_{S,1} = 2$, $x_{S,2} = 1$, $x_{1,2} = 1$, $x_{1,3} = 1$, $x_{2,F} = 2$, $x_{3,F} = 1$ te $x_{F,S} = f(\mathbf{x}) = 3$, što je prikazano i na Slici 6.8.



Slika 6.8
Optimalni transport nafte na zadanom naftovodu

Iz rezultata sa Slike 6.8 vidljivo je da se po liniji $S \to 2 \to F$ zbog kapaciteta veze $2 \to F$ može dostaviti maksimalno dva milijuna barela na sat, a gornjom linijom $S \to 1 \to 3 \to F$ dodatno još najviše jedan milijun barela na sat, zbog uskog grla $3 \to F$, što čini ukupno maksimalno tri milijuna barela koje se može transportirati kroz cijelu mrežu.

Primjer 6.4 Teretni zrakoplov ima tri odjeljka za teret: prednji, centralni i stražnji odjeljak. Odjeljci zrakoplova imaju ograničenja s obzirom na masu i volumen tereta koji mogu prihvatiti (Tablica 6.2).

Tablica 6.2
Ograničenja dopuštene mase i volumena u odjeljcima zrakoplova

Odjeljak	**Dopuštena masa tereta [t]**	**Skladišni prostor [m^3]**
Prednji	10	68
Centralni	16	87
Stražnji	8	53

Masa tereta raspoređenog po odjeljcima mora biti proporcionalna dopuštenoj masi tereta za te odjeljke kako bi se zadržao balans zrakoplova.

Za transport zrakoplovom su dostupne četri vrste tereta, koji svaki ima specificiranu masu, obujam i profit (Tablica 6.3).

Tablica 6.3
Ograničenja dopuštene mase i volumena u odjeljcima zrakoplova

Teret	**Masa [t]**	**Obujam [m^3/t]**	**Zarada [€]**
C1	18	4.80	310
C2	15	6.50	380
C3	23	5.80	350
C4	12	3.90	285

Bilo koja količina tereta je prihvatljiva. Cilj je pronaći koliko svakog pojedinog tereta treba ukrcati te kako ih distribuirati u odjeljke zrakoplova kako bi se maksimizirao ukupan profit leta.

■

Potrebno je odrediti mase svakog od četri vrste tereta koje treba ukrcati u svaki od tri odjeljaka. Neka je x_{ij} masa tereta i koji se ukrcava u odjeljak j.

$$\mathbf{x} = (x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23}, x_{31}, x_{32}, x_{33}, x_{41}, x_{42}, x_{43})$$

U zrakoplov se ne može ukrcati više tereta nego što ga ima na raspolaganju. To vrijedi za svaku od četri vrste tereta.

$$\begin{aligned} x_{11} + x_{12} + x_{13} &\leq 18 \\ x_{21} + x_{22} + x_{23} &\leq 15 \\ x_{31} + x_{32} + x_{33} &\leq 23 \\ x_{41} + x_{42} + x_{43} &\leq 12 \end{aligned}$$

Dopuštena ukupna masa tereta u svakom odjeljku ne smije biti prekoračena.

$$\begin{aligned} x_{11} + x_{21} + x_{31} + x_{41} &\leq 10 \\ x_{12} + x_{22} + x_{32} + x_{42} &\leq 16 \\ x_{13} + x_{23} + x_{33} + x_{43} &\leq 8 \end{aligned}$$

Prostorni (volumni) kapaciteti odjeljaka određuje količine tereta koji se mogu ukrcati.

$$\begin{aligned} 4.80x_{11} + 6.50x_{21} + 5.80x_{31} + 3.90x_{41} &\leq 68 \\ 4.80x_{12} + 6.50x_{22} + 5.80x_{32} + 3.90x_{42} &\leq 87 \\ 4.80x_{13} + 6.50x_{23} + 5.80x_{33} + 3.90x_{43} &\leq 53 \end{aligned}$$

Da bi se zadovoljio balans masa po odjeljcima i stabilnost zrakoplova, mase u odjeljcima moraju biti proporcionalne.

$$\frac{x_{11} + x_{21} + x_{31} + x_{41}}{10} = \frac{x_{12} + x_{22} + x_{32} + x_{42}}{16} = \frac{x_{13} + x_{23} + x_{33} + x_{43}}{8}$$

Potrebno je maksimizirati ukupni profit ostvaren prijevozom tereta.

$$\begin{aligned} f(\mathbf{x}) = \quad & (x_{11} + x_{12} + x_{13}) \cdot 310 \quad + \\ & (x_{21} + x_{22} + x_{23}) \cdot 380 \quad + \\ & (x_{31} + x_{32} + x_{33}) \cdot 350 \quad + \\ & (x_{41} + x_{42} + x_{43}) \cdot 285 \end{aligned}$$

Cjeloviti programski kod koji rješava ovaj zadatak upotrebom modula PuLP dan je u nastavku (Python kod 6.5).

Python kod 6.5 Rješavanje problema transportnog zrakoplova pomoću modula PuLP

```
from pulp import *

# Varijable
x11 = LpVariable("x11", 0)
x12 = LpVariable("x12", 0)
x13 = LpVariable("x13", 0)

x21 = LpVariable("x21", 0)
x22 = LpVariable("x22", 0)
x23 = LpVariable("x23", 0)

x31 = LpVariable("x31", 0)
x32 = LpVariable("x32", 0)
x33 = LpVariable("x33", 0)

x41 = LpVariable("x41", 0)
x42 = LpVariable("x42", 0)
x43 = LpVariable("x43", 0)

# LP problem
prob = LpProblem("Plane Cargo", LpMaximize)

# Cilj
prob += (x11 + x12 + x13)*310 + (x21 + x22 + x23)*380 + \
        (x31 + x32 + x33)*350 + (x41 + x42 + x43)*285

# Ograničenja
prob += x11 + x12 + x13 <= 18
prob += x21 + x22 + x23 <= 15
prob += x31 + x32 + x33 <= 23
prob += x41 + x42 + x43 <= 12

prob += x11 + x21 + x31 + x41 <= 10
prob += x12 + x22 + x32 + x42 <= 16
prob += x13 + x23 + x33 + x43 <= 8

prob += 4.80*x11 + 6.50*x21 + 5.80*x31 + 3.90*x41 <= 68
prob += 4.80*x12 + 6.50*x22 + 5.80*x32 + 3.90*x42 <= 87
prob += 4.80*x13 + 6.50*x23 + 5.80*x33 + 3.90*x43 <= 53

prob += (x11 + x21 + x31 + x41)/10 == (x12 + x22 + x32 + x42)/16
prob += (x12 + x22 + x32 + x42)/16 == (x13 + x23 + x33 + x43)/8

# Rješavanje
status = prob.solve()

# Ispis rješenja
```

```
print('Status:', LpStatus[status])

for v in prob.variables():
    print(v.name, "=", v.varValue)

print("Total profit = ", value(prob.objective))
```

6.3 Analiza u uvjetima neizvjesnosti

Siniša Družeta

U ovom poglavlju bavimo se optimizacijom u uvjetima neizvjesnosti. Pod time podrazumijevamo situacije u kojima ishode određenih procesa ne možemo uzeti kao sigurne, nego tek sa određenom vjerojatnošću, općenito $p \leq 1$.

6.3.1 Kriteriji odlučivanja

Svi smo imali situacije u kojima smo morali donijeti važne odluke, pri čemu nismo bili sigurni u sve faktore koji su utjecali na odluku (ili ih nismo savršeno poznavali). U tim slučajevima obično bismo se vodili intuicijom, tj. takve probleme bi rješavali "po osjećaju".

U ovom poglavlju bavimo se metodama koje nam omogućuju da odluke u uvjetima neizvjesnosti donosimo racionalno i sistemski. Postoji više pristupa odlučivanju u uvjetima neizvjenosti, što ćemo pokazati na sljedećem primjeru.

Primjer 6.5 Uzmimo problem nabave novina na kiosku, preuzet iz Winston i Goldberg (2004) i prilagođen. Zadatak nam je odlučiti koliko ćemo primjeraka određenih novina naručiti, pri čemu ne znamo koliko ćemo ih uspjeti prodati. Recimo da neprodane novine ne možemo vratiti natrag dobavljaču i da nam svaki primjerak novina u nabavi košta 2 €, a prodajna cijena mu je 2.50 €. Po iskustvu znamo da ne možemo prodati više od 10 komada, dok također znamo da ćemo sigurno prodati barem 6 komada. Osim toga, koliko smo po iskustvu uspjeli procijeniti, jednaka je vjerojatnost da ćemo prodati 6, 7, 8, 9 ili 10 komada. ■

Matematičkim jezikom izrečeno, imamo skup mogućih ishoda $S = \{6,7,8,9,10\}$ (broj traženih komada), sa vjerojatnostima $p_6 = p_7 = p_8 = p_9 = p_{10} = 1/5$ odnosno $p_j = 0.2$ za sve ishode $s_j \in S$. Naš zadatak je izabrati najbolju opciju iz skupa mogućih odluka $A = \{6,7,8,9,10\}$ (broj naručenih komada). Moguće nagrade za svaku kombinaciju (a_i, s_j) mogu se izračunati kao zarade $r_{i,j} = 2.5 \cdot \min(a_i, s_j) - 2 \cdot a_i$ i dane su u Tablici 6.4.

Tablica 6.4
Moguća zarada na kiosku od prodaje novina

Naručeno a_i	Potražnja s_j				
	6	7	8	9	10
6	3	3	3	3	3
7	1	3.5	3.5	3.5	3.5
8	-1	1.5	4	4	4
9	-3	-0.5	2	4.5	4.5
10	-5	-2.5	0	2.5	5

Nije potrebno razmatrati odluke o narudžbi 5 ili manje komada, jer bi u tom slučaju sigurno nedostajalo novina s obzirom na potražnju, tj. za bilo koji ishod s_j zarada bi bila sigurno manja od zarade postignute sa naručenih 6 komada. Analogno, nije potrebno razmatrati odluke o narudžbi 11 ili više komada, jer bi u tom slučaju sigurno ostalo neprodanih primjeraka, tj. za bilo koji ishod s_j zarada bi bila sigurno manja od

Tablica 6.5 Maximin kriterij na primjeru kioska

Naručeno a_i	Najgori ishod s_j	Najmanja moguća zarada $\min_{s_j \in S}(r_{i,j})$
6	6, 7, 8, 9, 10	**3**
7	6	1
8	6	-1
9	6	-3
10	6	-5

zarade sa naručenih 10 komada.

U nastavku razmatramo kriterije na temelju kojih bi mogli racionalno i sistemski donijeti odluku o narudžbi novina odnosno izabrati najpovoljniji A_i.

Maximin kriterij.
Konzervativno, oprezno razmišljanje nas upućuje da izaberemo onu odluku čiji najlošiji ishod je najbolji od svih najlošijih mogućih ishoda drugih odluka (*maximin*). Drugim riječima, među opcijama A tražimo opciju a_{best} za koju vrijedi da je

$$\min_{s_j \in S}(r_{best,j}) = \max_{a_i \in A}\left(\min_{s_j \in S}(r_{i,j})\right). \tag{6.25}$$

Dakle, za svaku moguću odluku a_i pronađemo njoj pripadnu najmanju nagradu $r_{i,j}$ te među njima izaberemo onu odluku koja ima najveću od svih najmanjih mogućih nagrada. Na našem primjeru kioska proveden je takav proračun u Tablici 6.5, iz kojeg se vidi da je po *maximin* kriteriju najbolje izabrati opciju $a_i = 6$, tj. naručiti 6 primjeraka novina.

Maximax kriterij.
Mogli bi se voditi i drukčijom, više "pohlepnom" logikom, po kojoj bi recimo najbolje bilo izabrati onu odluku čiji najbolji ishod je najbolji od svih mogućih ishoda (*maximax*). Znači da u tom slučaju među opcijama A tražimo opciju a_{best} za koju vrijedi da je

$$\max_{s_j \in S}(r_{best,j}) = \max_{a_i \in A}\left(\max_{s_j \in S}(r_{i,j})\right) = \max(r_{i,j}). \tag{6.26}$$

Jednostavno među svim mogućim ishodima pronađemo najbolji mogući ishod (Tablica 6.6) i izaberemo onu odluku za koju je on moguć. Iz tablice je jasno da *maximax* kriterij preporuča naručiti 10 komada novina, jer smo jedino tako u mogućnosti ostvariti maksimalnu zaradu $\max(r_{i,j}) = 5$.

Minimax žaljenja.
Ovaj kriterij je ponešto suptilniji od do sada opisanih kriterija. On koristi ideju propuštene prilike da bi njenom minimizacijom došao do najbolje odluke. Postupak primjene kriterija *minimax žaljenja* počinje tako da za svaki ishod $s_j \in S$ pronađemo akciju $a_i \in A$ koja daje najbolju nagradu te onda za sve preostale akcije i taj isti ishod izračunamo "žaljenje", tj. propušteni dobitak

$$l_{i,j} = \max_{a_i \in A}(r_{i,j}) - r_{i,j}, \text{ za svaki } s_j \in S. \tag{6.27}$$

Vrijednosti žaljenja $l_{i,j}$ za naš primjer dane su u Tablici 6.7. Sada jednostavno na vrijednostima žaljenja primijenimo *minimax* kriterij odnosno nađemo onu odluku koja

Tablica 6.6 Maximax kriterij na primjeru kioska

Naručeno a_i	Najbolji ishod s_j	Najveća moguća zarada $\max_{s_j \in S}(r_{i,j})$
6	6, 7, 8, 9, 10	3
7	7, 8, 9, 10	3.5
8	8, 9, 10	4
9	9, 10	4.5
10	10	**5**

Tablica 6.7 Izračun žaljenja na primjeru kioska

Naručeno a_i	Potražnja s_j				
	6	7	8	9	10
6	$3-3=0$	$3.5-3=0.5$	$4-3=1$	$4.5-3=1.5$	$5-3=2$
7	$3-1=2$	$3.5-3.5=0$	$4-3.5=0.5$	$4.5-3.5=1$	$5-3.5=1.5$
8	$3-(-1)=4$	$3.5-1.5=2$	$4-4=0$	$4.5-4=0.5$	$5-4=1$
9	$3-(-3)=6$	$3.5-(-0.5)=4$	$4-2=2$	$4.5-4.5=0$	$5-4.5=0.5$
10	$3-(-5)=8$	$3.5-(-2.5)=6$	$4-0=4$	$4.5-2.5=2$	$5-5=0$

će u najgorem ishodu dati najmanje žaljenja. Formalnim jezikom, među opcijama A tražimo opciju a_{best} za koju vrijedi da je

$$\min_{s_j \in S}(l_{best,j}) = \min_{a_i \in A}\left(\max_{s_j \in S}(r_{i,j})\right). \tag{6.28}$$

U našem primjeru iz Tablice 6.8 vidljivo je da za naručenih 6 i 7 primjeraka novina najveće moguće žaljenje (ili bolje rečeno propuštena zarada) iznosi 2 €, što je najmanja vrijednost među svim opcijama. Stoga možemo zaključiti da je u našem primjeru po kriteriju *minimax žaljenja* najbolje naručiti 6 ili 7 primjeraka novina.

Očekivana vrijednost.

Ovaj kriterij se vrlo često koristi za donošenje tehničkih i poslovnih odluka. Za svaku odluku izračunava se očekivana nagrada kao srednja vrijednost dobiti za sve (jednako vjerojatne) ishode. Ovdje treba napomenuti da za izračun po kriteriju *očekivane vrijednosti* nije nužno da svi ishodi imaju jednaku vjerojatnost i upravo je to razlog zašto se ovaj kriterij široko upotrebljava. Dakle, očekivanu vrijednost za odluku a_i

Tablica 6.8 Minimax žaljenja na primjeru kioska

Naručeno a_i	Najgori ishod s_j	Najveće moguće žaljenje $\max_{s_j \in S}(l_{i,j})$
6	10	**2**
7	6	**2**
8	6	4
9	6	6
10	6	8

Naručeno a_i	Očekivana vrijednost e_i
6	$(3+3+3+3+3)/5 = \mathbf{3}$
7	$(1+3.5+3.5+3.5+3.5)/5 = \mathbf{3}$
8	$(-1+1.5+4+4+4)/5 = 2.5$
9	$(-3-0.5+2+4.5+4.5)/5 = 1.5$
10	$(-5-2.5+0+2.5+5)/5 = 0$

Tablica 6.9
Očekivana vrijednost na primjeru kioska

izračunavamo po izrazu

$$e_i = \sum_{s_j \in S} p_j r_{i,j}. \tag{6.29}$$

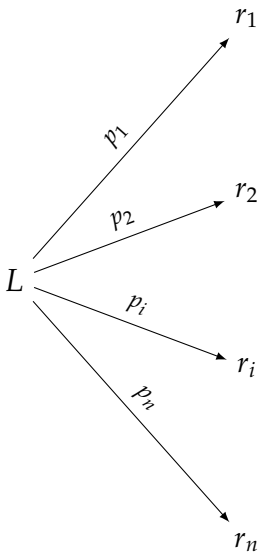
Za primjer kioska vrijednosti očekivane vrijednosti dane su u Tablici 6.9, gdje se vidi da bi po ovom kriteriju najbolji izbor bilo naručiti 6 ili 7 novina.

Kao što je već rečeno, često se tehničke, poslovne i ine odluke donose bez primjene ovdje opisanih kriterija. Sa druge strane, kao izgovor za to često se javlja argument da su problemi složeniji od npr. ovdje promatranog problema nabave novina za prodaju na kiosku. Stoga ćemo se u nastavku baviti teorijom korisnosti, koja nudi još napredniju metodologiju za donošenje odluka u uvjetima neizvjesnosti.

6.3.2 Teorija korisnosti

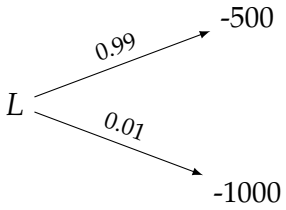
Ovdje ćemo izložiti **Von Neumann - Morgenstern** koncept teorije korisnosti (*utility theory*) koji omogućuje donošenje složenih odluka u uvjetima neizvjesnosti.

Možemo zamisliti općenitu neizvjesnost koja ima n mogućih ishoda sa pripadnim nagradama r_i, gdje svaki ishod ima vjerojatnost p_i. Takva situacija zove se **lutrija** i može se zapisati kao $L = (p_1, r_1;\ p_2, r_2;\ ...;\ p_n, r_n)$, ili dijagramom:

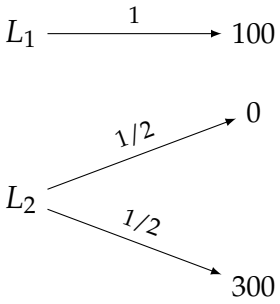


Jedan takav primjer bila bi situacija u kojoj smo stavili novo staklo na prozor koje je koštalo 500 € i procjenjujemo vjerojatnost da ćemo tijekom naredne godine dana morati ponovno stavljati novo staklo na $p = 0.01$ (1% šanse), s time da također procjenjujemo da je nemoguće da ćemo morati dva ili više puta stavljati novo staklo tijekom naredne

godine dana. U tom slučaju troškovi za staklo za taj prozor za narednu godinu dana iznose $L = (0.99, -500;\ 0.01, -1000)$ odnosno:



Uzmemo li sada da smo suočeni sa izborom između dviju lutrija; na primjer netko nam nudi sigurnih 100 €, ili pak da bacamo novčić za mogući dobitak od 300 €. To bi mogli zapisati ovako:



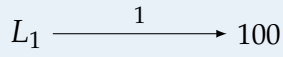
Postavlja se pitanje koja je lutrija bolja odnosno povoljnija za nas. Iako nam izračun očekivane vrijednosti govori da je lutrija L_2 bolja od lutrije L_1 ($e_1 = 1 \cdot 100 = 100$, $e_2 = 0.5 \cdot 300 + 0.5 \cdot 0 = 150$), mnogi bi ipak rađe izabrali "vrapca u ruci" odnosno lutriju L_1.

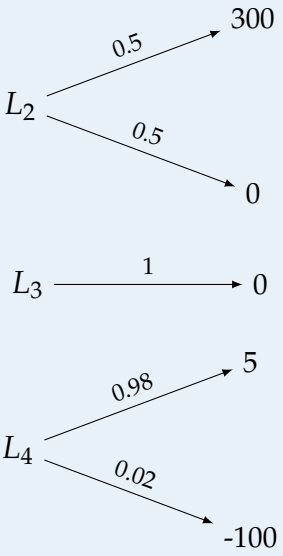
Da bi mogli odrediti koja lutrija je bolja za nas, moramo prvo definirati pojam **ekvivalentne lutrije**. Naime, moguće je da je nekome baš savršeno svejedno da li će izabrati lutriju L_1 ili lutriju L_2, jer su po njegovoj procjeni to potpuno jednako dobre opcije za njega. U tom slučaju lutrije L_1 i L_2 proglašavamo ekvivalentnim lutrijama, što zapisujemo kao $L_1 \equiv L_2$.

Kao što je sada već očito, procjena povoljnosti svake lutrije, pa tako i procjena o ekvivalentnosti dviju lutrija, u principu je individualna i ovisi o onome tko donosi odluku. To može biti jedna osoba, grupa ljudi (npr. uprava tvrtke) ili nekakav algoritam odnosno računalni program.

Postupak koji ćemo ovdje opisati omogućuje da između više različitih lutrija izaberemo onu najpovoljniju za nas. Da bi to bilo moguće moramo ih učiniti usporedivim, odnosno moramo ih na neki način rangirati.

Primjer 6.6 Uzmimo kao primjer zadatak sa četiri lutrije iz Winston i Goldberg (2004):

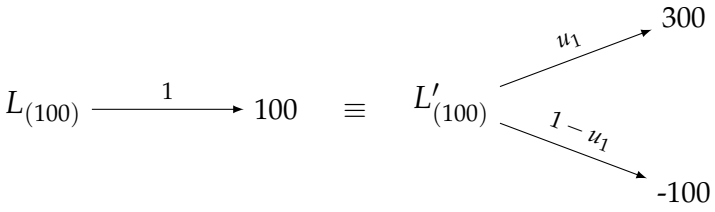




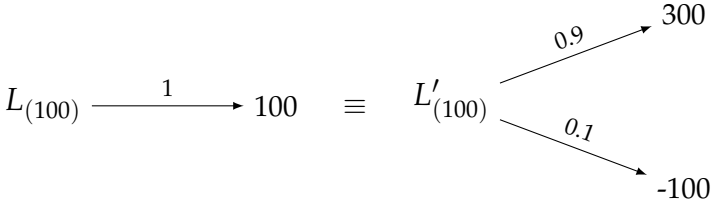
Na ovom primjeru odmah možemo primijetiti da je lutrija L_1 apsolutno povoljnija od lutrija L_3 i L_4, no nije na primjer jasno tko je povoljniji između lutrija L_3 ili L_4 niti kako se L_2 odnosi prema lutrijama L_1 i L_4. Von Neumann - Morgenstern pristup koji ćemo ovdje opisati daje odgovore na sva takva pitanja.

Postupak izbora najbolje lutrije počinje tako da se iz svih lutrija ukupno izdvoje globalno najpovoljniji ishod $r_{max} = \max(r_i)$ i najnepovoljniji ishod $r_{min} = \min(r_i)$. U našem slučaju to su $r_{max} = 300$ i $r_{min} = -100$. Za sve ostale ishode r_i, osoba koja odlučuje (ili grupa ljudi, sustav, algoritam) mora odrediti vjerojatnost u_i za koju bi ove dvije lutrije bile ekvivalentne, tj. jednako povoljne.

Tako za $r_1 = 100$ (jedini ishod iz L_1) tražimo vjerojatnost u_1 takvu da vrijedi:

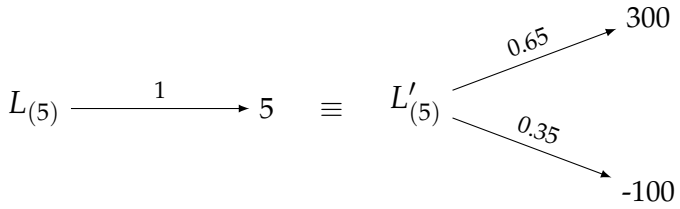


Recimo da dobijemo odgovor da je $u_1 = 0.9$, što znači da vrijedi:

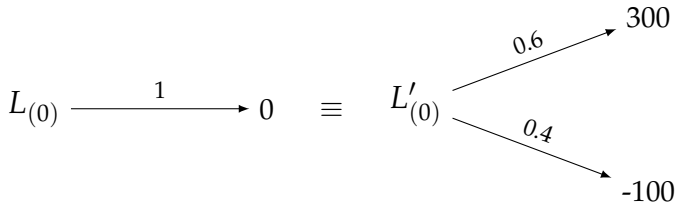


odnosno da je osobi koja odlučuje potpuno svejedno da li će dobiti sigurnih 100 €, ili se kockati između dobitka od 300 € i gubitka od 100 €, ako bi vjerojatnost dobitka bila 0.9.

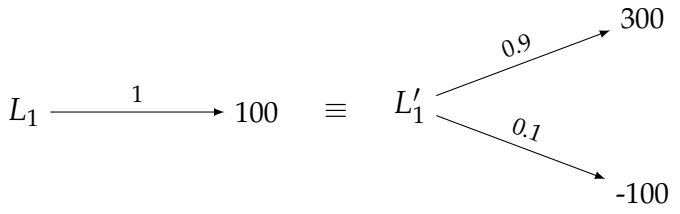
Nadalje, recimo da za $r_2 = 5$ (povoljniji ishod iz L_4), pripadajuća vjerojatnost u ekvivalentnoj lutriji iznosi $u_2 = 0.65$, tj. osoba je indiferentna između ovih dviju lutrija:

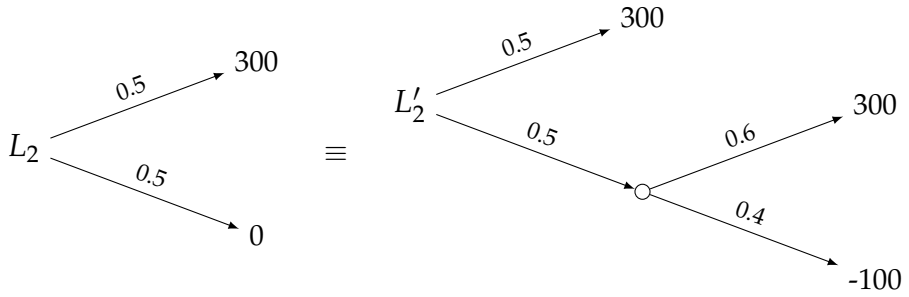


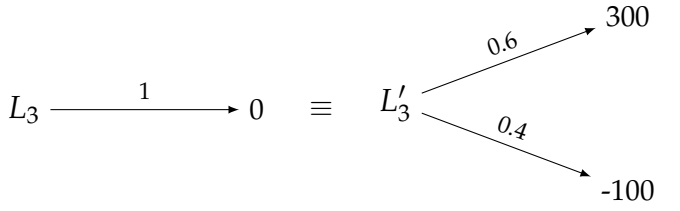
Isto tako, za $r_3 = 0$ (nepovoljniji ishod iz L_2 i jedini ishod iz L_3) dobili smo odgovor da je $u_3 = 0.6$ odnosno:

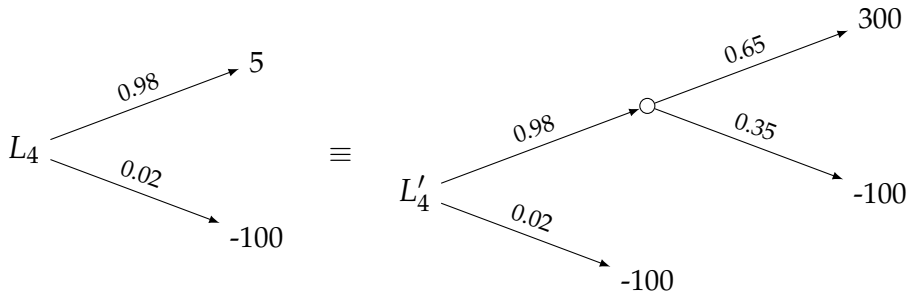


Sada, koristeći dobivene informacije o u_1, u_2 i u_3, možemo postaviti ekvivalentne lutrije L'_1, L'_2, L'_3 i L'_4 koje će kao ishode imati samo $r_{max} = 300$ i $r_{min} = -100$. Redom, one izgledaju ovako:





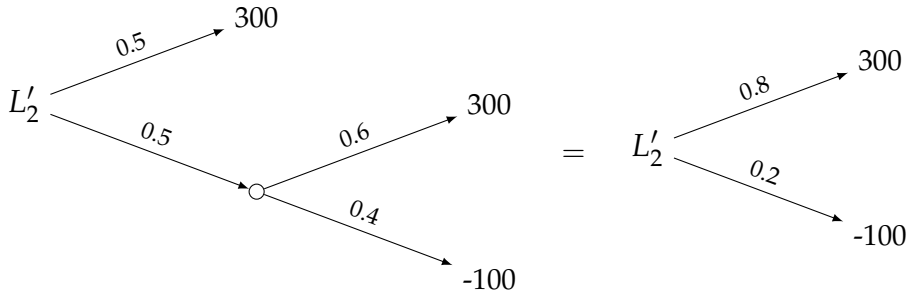




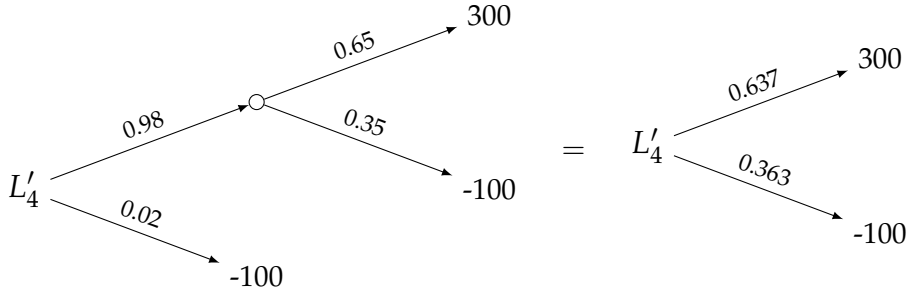
Lutrije L_2' i L_4' su **složene lutrije**, za razliku od lutrija koje imaju samo jedno grananje, koje se nazivaju **jednostavnim lutrijama**. Kao složene lutrije, L_2' i L_4' se mogu svesti na jednostavne, izračunavanjem ukupne vjerojatnosti za svaki od mogućih ishoda (a to su sada samo r_{max} i r_{min}):

$$q_i = \sum_i p_i u_i. \tag{6.30}$$

To znači da za lutriju L_2' ukupnu vjerojatnost dobitka 300 € izračunavamo kao $q_{(300)} = 0.5 + 0.5 \cdot 0.6 = 0.8$, a ukupnu vjerojatnost dobitka -100 € kao $q_{(-100)} = 0.5 \cdot 0.4 = 0.2$, tj.

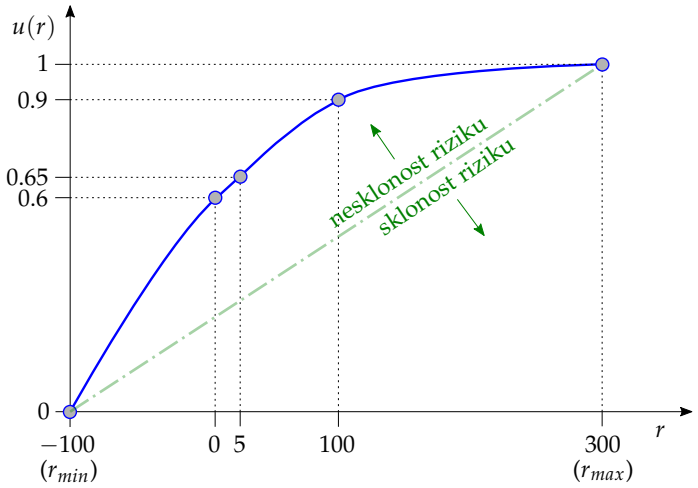


Na isti način rješavamo i složenu lutriju L_4':



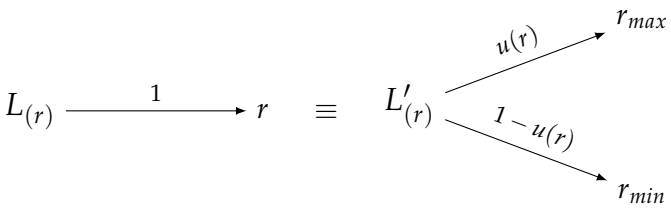
Nakon ovako provedenog postupka dobili smo četiri jednostavne lutrije L_1', L_2', L_3' i L_4' koje za ishode imaju r_{max} i r_{min} i kao takve lako ih možemo rangirati. Elementarna logika upućuje nas na to da je lutrija koja ima veću ukupnu vjerojatnost povoljnijeg ishoda $q(r_{max})$ u cjelini povoljnija. Na korištenom primjeru to znači da je najpovoljnija lutrija L_1 sa $q(r_{max}) = 0.9$, zatim lutrija L_2 sa $q(r_{max}) = 0.8$, zatim L_4 sa $q(r_{max}) = 0.637$ i konačno najnepovoljnija lutrija je L_3 sa $q(r_{max}) = 0.6$.

Dakle, Von Neumann - Morgenstern postupkom smo četiri lutrije koje međusobno nisu bile (potpuno) usporedive sveli na četiri ekvivalentne lutrije koje su međusobno potpuno usporedive, što nam omogućuje da među njima izaberemo najpovoljniju.



Slika 6.9
Dobivena funkcija korisnosti za dani primjer sa četiri lutrije

Rekapitulirajmo provedeni postupak općenitim riječnikom. Za svaku od nagrada (ishoda) iz svih lutrija odredili smo vjerojatnost $u(r)$ za koju je osobi koja odlučuje potpuno svejedno da li će sigurno primiti tu istu nagradu r ili će se "kockati" između najbolje nagrade r_{max} i najslabije nagrade r_{min}, uz vjerojatnost dobitka najbolje nagrade $u(r)$. Dakle, za svaki ishod r postavljamo ekvivalentnu lutriju $L'_{(r)}$:



i osobu koja odlučuje (ili grupu ljudi, sustav, algoritam) pitamo da nam procjeni $u(r)$.

Vjerojatnost $u(r)$ zove se **korisnost** (*utility*) i ako je definiramo kako je opisano očito je da mora biti $u(r_{max}) = 1$, kao i $u(r_{min}) = 0$, jer u ta dva slučaja ne samo da vrijedi ekvivalentnost $L_{(r_{max})} \equiv L'_{(r_{max})}$ i $L_{(r_{min})} \equiv L'_{(r_{min})}$, nego su te lutrije upravo potpuno jednake, tj. $L_{(r_{max})} = L'_{(r_{max})}$ i $L_{(r_{min})} = L'_{(r_{min})}$, a sve jednake lutrije automatski moraju biti i ekvivalentne.

Procjene $u(r)$ za naš primjer dane su u Tablici 6.10. Na temelju ovih podataka mogli bi konstruirati krivulju; takva krivulja zove se **funkcija korisnosti** (Slika 6.9) i ona nam može dati neke korisne informacije o osobi koja donosi odluke.

nagrada r	korisnost $u(r)$
-100	0
0	0.6
5	0.65
100	0.9
300	1

Tablica 6.10
Korisnosti nagrada na analiziranom primjeru sa četiri lutrije

Uvidom u izgled krivulje korisnosti možemo zaključiti da je osoba čija se krivulja korisnosti nalazi iznad dijagonale koja povezuje točke $(r_{min}, 0)$ i $(r_{max}, 1)$ nesklona riziku, jer zahtjeva veću sigurnost (tj. veće vjerojatnosti) da bi ušla u nesigurnost

umjesto prihvatila mogućnost sigurnog dobitka. Analogno, osoba čija se krivulja korisnosti nalazi u području ispod dijagonale može se proglasiti sklonom riziku, jer već za manje vjerojatnosti dobitka radije prihvaća nesigurnost nego siguran dobitak. To možemo izreći i geometrijski:

- osoba je nesklona riziku za strogo konkavnu $u(r)$,
- osoba je sklona riziku za strogo konveksnu $u(r)$.

Primjer 6.7 U nastavku dajemo programski kod koji rješava problem određivanja najpovoljnije lutrije za proizvoljan broj lutrija sa proizvoljnim brojem ishoda (Python kod 6.6). Pokušajte ga iskoristiti da riješite sljedeći problem: u svoj proizvod možete ugraditi uređaj prvog dobavljača, koji ima 5% vjerojatnosti neispravnosti, a na kojem ćete zaraditi 90 €, ili uređaj drugog dobavljača, koji ima 1% vjerojatnosti neispravnosti, a na kojem ćete zaraditi 100 €; u slučaju nužne zamjene uređaja prvog dobavljača zaradit ćete u konačnici 50 €, dok ćete kod drugog dobavljača u takvom slučaju ostvariti gubitak od 1000 €; koji dobavljač je povoljniji za vas? ■

Python kod 6.6 Određivanje najpovoljnije lutrije

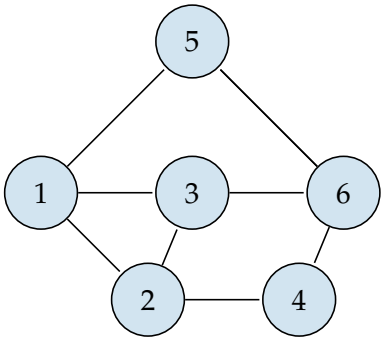
```
"""
format ulazne datoteke:
r1 p1 r2 p2 r3 p3 ... rn
(jedna lutrija po redu)
"""

import numpy as np
import matplotlib.pyplot as plt

filename = 'problem.txt'

# učitavanje podataka
retci = open(filename).read().splitlines()
L = []
for redak in retci:
    lut = np.empty(0)
    for x in redak.split(' '):
        lut = np.append(lut, float(x))
    lut = np.append(lut, 1 - np.sum(lut[1::2])) # vjerojatnost zadnjeg ishoda
    lutm = np.empty([np.size(lut)/2, 2])
    lutm[:, 0] = lut[::2]
    lutm[:, 1] = lut[1::2]
    L.append(lutm)

# ispis problema
print('problem:')
for i in range(len(L)):
    print('L' + str(i+1) + ' ---')
    lutm = L[i]
    for j in range(np.size(lutm[:,0])):
        print('- ' + str(lutm[j,0]) + ' (' + str(lutm[j,1]) +')')

# pronaći sve vrijednosti ishoda
r = np.empty(0)
for lutm in L:
    r = np.unique(np.append(r, lutm[:,0]))
```

```
rmax = np.max(r)
rmin = np.min(r)

# odrediti vjerojatnosti u za ekvivalentne sigurne ishode
print('rjesavam:')
u = np.empty(np.size(r))
for i in range(np.size(r)):
    if r[i] == rmin:
        u[i] = 0.0
    elif r[i] == rmax:
        u[i] = 1.0
    else:
        print('sigurnih ' + str(r[i]) + ' ili lutrija ' \
                + str(rmax) + ' (u) / ' + str(rmin) + ' (1-u) ?')
        u[i] = input('- svejedno mi je za u = ')

# odrediti najbolju lutriju
prmax = np.zeros(len(L))
for i in range(len(L)):
    lutm = L[i]
    for j in range(np.size(lutm[:,0])):
        for k in range(np.size(r)):
            if lutm[j, 0] == r[k]:
                prmax[i] = prmax[i] + lutm[j, 1] * u[k]
best = np.where(prmax == np.max(prmax))[0][0]
print('najbolja lutrija je L' + str(best+1) + \
            ' (u = ' + str(prmax[best]) + ')')

# funkcija korisnosti
plt.plot(r, u)
plt.plot([rmin, rmax], [0, 1], '--')
plt.xlim([rmin, rmax])
plt.xlabel('ishod r')
plt.ylabel('vjerojatnost u')
plt.title('funkcija korisnosti')
plt.show()
```

6.4 Određivanje najkraćeg puta

Siniša Družeta

Jedan od karakterističnih optimizacijskih problema je određivanje najkraćeg puta (*shortest path problem*), gdje u nekakvoj mreži točaka povezanih vezama tražimo najkraći put između dviju točaka. Matematički naziv za takvu mrežu je graf, gdje razlikujemo usmjerene grafove (gdje su točke povezane samo u jednom smjeru) i neusmjerene grafove (gdje su točke povezane u oba smjera). Primjer jednog neusmjerenog grafa dan je na Slici 6.10.



Slika 6.10
Primjer neusmjerenog grafa

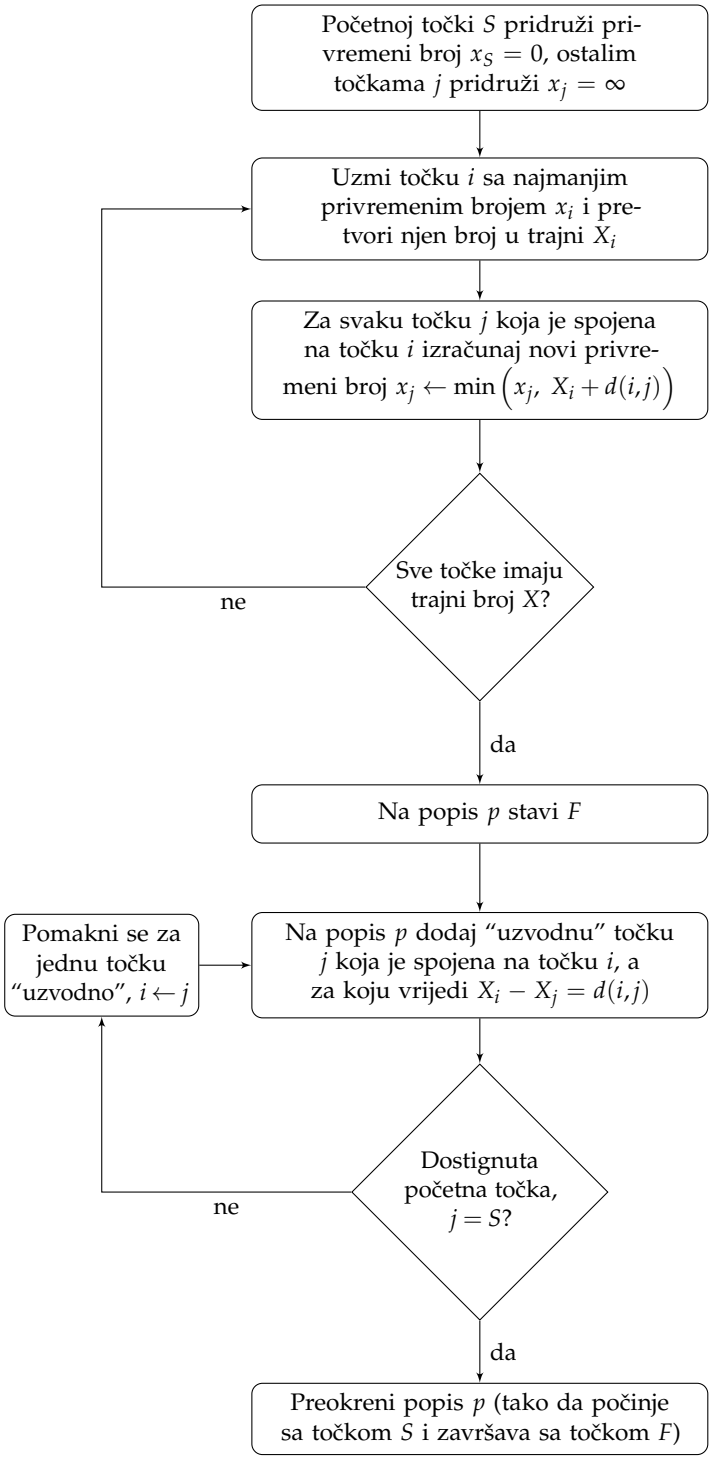
Za određivanje najkraćeg puta u mreži može se koristiti **Dijkstrin algoritam** (Edsger W. Dijkstra) iz 1956. godine. Ovaj algoritam predviđa da svaka točka ima svoj indeks i, a svaka veza ima duljinu $d(i,j) > 0$. Svakoj točki na mreži se pridružuje broj x_i koji se mijenja tijekom provođenja algoritma i u nekom trenutku se pretvara u trajni broj X_i. Najkraći put se u konačnici određuje na temelju trajnih brojeva X_i. Na mreži postoji ulazna (početna) točka S i izlazna (krajnja) točka F. Postupak koji pronalazi najkraći put p prikazan je dijagramom toka na Slici 6.11.

Ako uočimo da udaljenosti između točaka na mreži ne moraju predstavljati isključivo geometrijske udaljenosti, nego bilo kakav napor ili trošak, onda postaje jasno da određivanje najkraćeg puta može biti korisno i u rješavanju problema koji nisu geometrijske prirode. Na jednom takvom primjeru (iz Winston i Goldberg (2004)) pokazat ćemo kako Dijkstrin algoritam funkcionira.

Recimo da smo u nekom trenutku kupili novi automobil po cijeni 24000 €. Godišnji troškovi održavanja automobila za svaku godinu starosti automobila, kao i vrijednost auta na tržištu dani su u Tablici 6.11. Radi jednostavnosti, recimo da novi automobil u svakoj godini možemo kupiti po inicijalnoj cijeni od 24000 €. Potrebno je minimizirati ukupne troškove korištenja takvog automobila u periodu od pet godina.

Tablica 6.11 Troškovi održavanja i vrijednost automobila

Godine starosti	**Godišnji troškovi održavanja** t	**Tržišna vrijednost** v
0	4000	-
1	8000	14000
2	10000	12000
3	18000	4000
4	24000	2000
5	-	0



Slika 6.11
Dijkstrin algoritam

Problem možemo formulirati kao problem najkraćeg puta, gdje će točke u mreži predstavljati početak i-te godine ($i \in \{1,2,3,4,5,6\}$). Duljina veze $d(i,j)$ reprezentira ukupne troškove upotrebe automobila u tom periodu, tj. troškove kupnje automobila na početku i-te godine, troškove održavanja automobila do početka j-te godine i dobitak od prodaje automobila na početku j-te godine. Drugim riječima, ako automobil u trenutku prodaje ima $k = j - i$ godina, slijedi:

$$d_k = 24000 + \sum_{n=0}^{k-1} t_n - v_k, \tag{6.31}$$

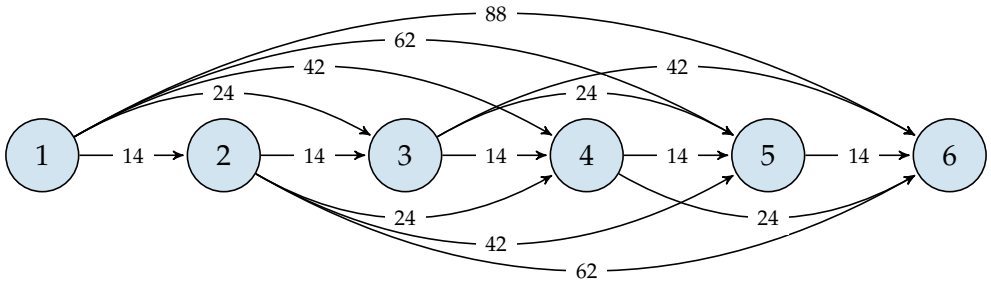
gdje $d_k = d(i,j)$ predstavlja trošak korištenja automobila u periodu od k godina, t_n je trošak održavanja automobila u n-toj godini starosti automobila, a v_k vrijednost automobila starog k godina.

Uzmemo li podatke iz Tablice 6.11, možemo izračunati troškove svih mogućih strategija obnove automobila (iznosi su dani u tisućama €):

$$\begin{aligned}
d(1,2) = d_1 &= 24 + (4) - 14 = 14 \\
d(1,3) = d_2 &= 24 + (4+8) - 12 = 24 \\
d(1,4) = d_3 &= 24 + (4+8+10) - 4 = 42 \\
d(1,5) = d_4 &= 24 + (4+8+10+18) - 2 = 62 \\
d(1,6) = d_5 &= 24 + (4+8+10+18+24) - 0 = 88 \\
d(2,3) = d_1 &= 14 \\
d(2,4) = d_2 &= 24 \\
d(2,5) = d_3 &= 42 \\
d(2,6) = d_4 &= 62 \\
d(3,4) = d_1 &= 14 \\
d(3,5) = d_2 &= 24 \\
d(3,6) = d_3 &= 42 \\
d(4,5) = d_1 &= 14 \\
d(4,6) = d_2 &= 24 \\
d(5,6) = d_1 &= 14
\end{aligned}$$

Sada možemo kreirati mrežu na Slici 6.12 kao usmjereni graf na kojem točke (čvorovi) određuju početak i-te godine, a duljine veza $d(i,j)$ označuju troškove upotrebe automobila od početka i-te godine do kraja j-te godine. Postavlja se pitanje koja strategija obnove automobila je najbolja, tj. najjeftinija, odnosno koji "put" kroz mrežu će biti najkraći. Da li mijenjati automobil svaku godinu, ili svake dvije, tri, četiri godine, ili pak voziti jedan te isti automobil svih pet godina?

Prije početka provođenja Dijkstrinog algoritma potrebno je utvrditi da točka 1 predstavlja početnu točku $S = 1$, a točka 6 krajnju točku $F = 6$. Nakon toga, prvi korak algoritma kazuje da moramo početnoj točki pridružiti privremeni broj $x_S = 0$, a svim ostalim točkama pridružimo $x_j = \infty$. Sada, u petlji od drugog do četvrtog koraka metode pretvaramo najmanji privremeni broj u trajni i ažuriramo privremene brojeve za sve preostale točke, dok sve točke ne dobiju trajni broj. Cijeli ovaj postupak sa svim stanjima svih privremenih i trajnih brojeva prikazan je u Tablici 6.12, gdje su trajni brojevi X otisnuti masnim.



Slika 6.12 Mreža za minimizaciju troškova korištenja automobila

Tablica 6.12 Određivanje privremenih i trajnih brojeva na mreži za minimizaciju troškova korištenja automobila

Ciklus postupka	1	2	3	4	5	6	Opis postupka
1	0	∞	∞	∞	∞	∞	inicijalizacija
2	**0**	∞	∞	∞	∞	∞	$\min x_i = x_1 \rightarrow X_1$
3	**0**	14	24	42	62	88	ažuriranje x_2, x_3, x_4, x_5, x_6
4	**0**	**14**	24	42	62	88	$\min x_i = x_2 \rightarrow X_2$
5	**0**	**14**	24	38	56	76	ažuriranje x_3, x_4, x_5, x_6
6	**0**	**14**	**24**	38	56	76	$\min x_i = x_3 \rightarrow X_3$
7	**0**	**14**	**24**	38	48	66	ažuriranje x_4, x_5, x_6
8	**0**	**14**	**24**	**38**	48	66	$\min x_i = x_4 \rightarrow X_4$
9	**0**	**14**	**24**	**38**	48	62	ažuriranje x_5, x_6
10	**0**	**14**	**24**	**38**	**48**	62	$\min x_i = x_5 \rightarrow X_5$
11	**0**	**14**	**24**	**38**	**48**	62	ažuriranje x_6
12	**0**	**14**	**24**	**38**	**48**	**62**	$\min x_i = x_6 \rightarrow X_6$

Nakon što smo proveli postupak određivanja trajnih brojeva X za sve točke, možemo krenuti sa formiranjem najkraćeg puta p. Dijsktrin algoritam kaže da krećemo od krajnje točke $F = 6$ i na popis dodajemo točku za koju vrijedi $X_j - X_i = d(i,j)$ i tako dalje, dok ne stignemo do početne točke $S = 1$.

Dakle, krećemo od točke 6 i tražimo sljedeću "uzvodnu" točku; u ovom slučaju odmah pronalazimo dvije moguće točke, to su točka 5 ($X_6 - X_5 = 62 - 48 = 14 = d(5,6)$) i točka 4 ($X_6 - X_4 = 62 - 38 = 24 = d(4,6)$). Formiramo odmah dva moguća popisa $p_1 = (6,5)$ i $p_2 = (6,4)$. Nastavljamo dalje razvoj puta p_1 i pronalazimo sljedeću točku, a to je točka 3 ($X_5 - X_3 = 48 - 24 = 24 = d(3,5)$) koju dodajemo na popis $p_1 = (6,5,3)$. Konačno, na p_1 dodajemo još i točku 1 ($X_3 - X_1 = 24 - 0 = 24 = d(1,3)$), čime je popis p_1 završen i glasi $p_1 = (6,5,3,1)$. Sada se možemo pozabaviti popisom p_2, gdje nastavljamo "uzvodno" od točke 4 i pronalazimo točku 3 ($X_4 - X_3 = 38 - 24 = 14 = d(3,4)$) i točku 2 ($X_4 - X_2 = 38 - 14 = 24 = d(2,4)$). Točku 3 dodajemo na popis $p_2 = (6,4,3)$, a točku 2 dodajemo na novi popis $p_3 = (6,4,2)$. Da dovršimo popis p_2 moramo još pronaći točku 1 ($X_3 - X_1 = 24 - 0 = 24 = d(1,3)$), a da dovršimo popis p_3 na njega također moramo dodati točku 1 ($X_2 - X_1 = 14 - 0 = 14 = d(1,2)$). Time smo, uz p_1 dobili još popise $p_2 = (6,4,3,1)$ i $p_3 = (6,4,2,1)$.

Na samom kraju još moramo preokrenuti popise. Na taj način u našem slučaju

možemo formirati tri jednako kratka puta:

- $p_1: \ 1 \to 3 \to 5 \to 6$,
- $p_2: \ 1 \to 3 \to 4 \to 6$,
- $p_3: \ 1 \to 2 \to 4 \to 6$.

Putevi p_1, p_2 i p_3 jednako su kratki i predstavljaju najkraće moguće puteve kroz mrežu prikazanu na Slici 6.12. Analizom dobivenih puteva vidimo da je najekonomičnije mijenjati automobil svake dvije godine. S obzirom da zadatak traži da se planiraju troškovi za period od pet godina, nije moguće cijeli period odraditi isključivo u dvogodišnjim ciklusima. Zbog toga je neizbježno u nekom trenutku promijeniti automobil nakon samo jedne godine (rješenje p_1 predviđa da se to desi u zadnjoj godini, p_2 u četvrtoj godini, a p_3 odmah u drugoj godini).

6.5 Pretraživanje uzorkom

Stefan Ivić
Siniša Družeta

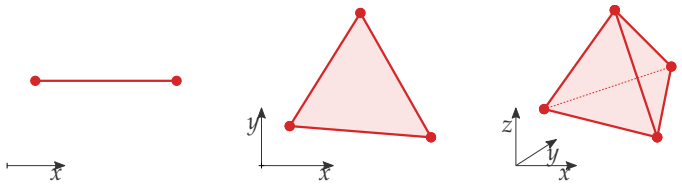
Optimizacija je tradicionalno bazirana na upotrebi derivacija, bilo izravnim izračunavanjem minimuma funkcije pomoću derivacije, ili korištenjem (točno ili približno izračunatih) derivacija za određivanje smjera spuštanja prema minimumu iz neke početne točke. S obzirom da u realnim optimizacijskim problemima derivaciju funkcije cilja često ne poznajemo, a njena aproksimacija je računalno zahtjevna, razvijene su metode za optimizaciju funkcije više varijabli koje ne koriste derivaciju. Takve metode su razvijene u drugoj polovici 20. st. i zovu se **metode direktnog pretraživanja** (eng. *Direct Search*).

Jedna skupina metoda direktnog pretraživanja su **metode pretraživanja uzorkom** (eng. *Pattern Search*), među kojima je jedna od najjednostavnijih **Nelder-Mead metoda**, poznata još kao *Downhill Simplex* metoda ili *Amoeba* metoda.

6.5.1 Nelder-Mead metoda

Nelder-Mead metoda je tehnika za minimizaciju funkcije cilja u višedimenzionalnom prostoru, koju su osmislili John Nelder i Roger Mead 1965. godine (Nelder i Mead (1965)).

Nelder-Mead metoda koristi koncept simpleksa, što je specijalni politop od $n+1$ točaka u n dimenzija. Politop je geometrijski oblik sa ravnim stranicama koji egzistira u bilo kojem broju dimenzija. Poligon je politop u dvije dimenzije. Primjeri simpleksa su dužina na pravcu (1D), trokut u ravnini (2D), tetraedron u prostoru (3D), itd. Moglo bi se reći da je simpleks najjednostavniji politop koji ima istu dimenzionalnost kao i prostor u kojem se nalazi (Slika 6.13).



Slika 6.13
Simpleks u 1, 2 i 3 dimenzije

Metoda se zasniva na praćenju uzorka (simpleksa) koji uspoređuje vrijednosti funkcije u svojim vrhovima, tj. točkama simpleksa. Najlošija točka se odbacuje i uvodi se nova točka koja sa ostalima čini novi simpleks. Pronalaženje nove točke simpleksa u svakoj iteraciji može uzrokovati širenje ili smanjenje raspona simpleksa. Smanjivanje simpleksa kroz iteracije metode dovodi do konvergencije (svih točaka simpleksa) ka optimumu.

Prije nego što metoda krene sa svojim iteracijskim postupkom, potrebno je definirati (tj. inicijalizirati) početni simpleks (uzorak). Ako funkcija cilja ima n varijabli, simpleks sadrži $n+1$ točku: $(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n)$. Uobičajeno se zadaje pravokutni simpleks, što znači da se ostali vrhovi simpleksa dobiju ekspanzijom iz početne točke $\mathbf{x}_0$:

$$\mathbf{x}_i = \mathbf{x}_0 + \Delta_i \cdot \mathbf{e}_i, \qquad i = 1, \ldots, n \,, \tag{6.32}$$

gdje je Δ_i veličina početnog koraka te $\mathbf{e}_i$ jedinični vektor u smjeru osi i.

Kao primjer uzmimo trodimenzionalni prostor pretraživanja, u kojemu ćemo imati simpleks od četiri točke. Moramo imati zadanu početnu točku

$$\mathbf{x}_0 = \begin{bmatrix} x_{0,x} \\ x_{0,y} \\ x_{0,z} \end{bmatrix},$$

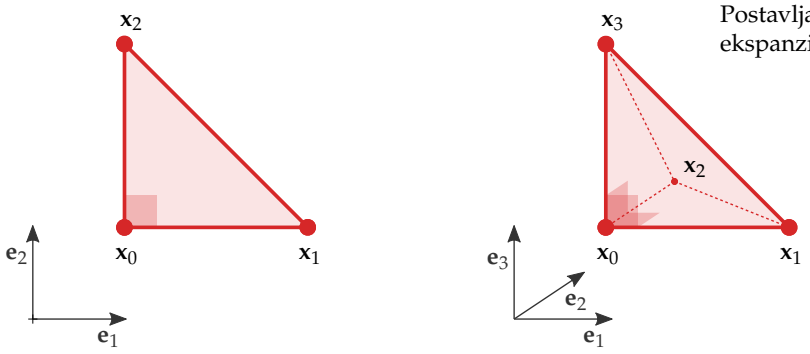
pa ćemo od nje, proširenjem po izrazu (6.32), dobiti sljedeću točku

$$\mathbf{x}_1 = \begin{bmatrix} x_{1,x} \\ x_{1,y} \\ x_{1,z} \end{bmatrix} = \begin{bmatrix} x_{0,x} \\ x_{0,y} \\ x_{0,z} \end{bmatrix} + \Delta_x \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} x_{0,x} + \Delta_x \\ x_{0,y} \\ x_{0,z} \end{bmatrix}$$

te analogno preostale dvije točke:

$$\mathbf{x}_2 = \begin{bmatrix} x_{0,x} \\ x_{0,y} + \Delta_y \\ x_{0,z} \end{bmatrix}, \qquad \mathbf{x}_3 = \begin{bmatrix} x_{0,x} \\ x_{0,y} \\ x_{0,z} + \Delta_z \end{bmatrix},$$

što je i ilustrirano na Slici 6.14.



Slika 6.14
Postavljanje početnog simpleksa ekspanzijom iz početne točke x_0

Nakon što je postavljen početni simpleks, metoda u svakoj iteraciji transformira simpleks, pomičući vrh po vrh, u svrhu istraživanja okolnog područja u prostoru pretraživanja. Tijekom svakog koraka testira se nekoliko modifikacija trenutnog simpleksa i zadržava se ona verzija simpleksa koja je najbliža minimumu funkcije. U nastavku su opisani svi koraci postupka koji se ponavlja kroz iteracije.

Evaluacija i sortiranje

U prvom koraku metode, za sve točke simpleksa izračunava se vrijednost funkcije cilja:

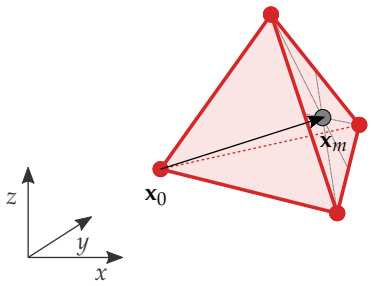
$$f_i = f(\mathbf{x}_i) \qquad i = 0, 1, \ldots, n.$$

Uz pretpostavku da se riješava minimizacijski problem, točke se na početku svake iteracije metode sortiraju (renumeriraju) tako da vrijedi:

$$f_0 \geq f_1 \geq \ldots \geq f_n.$$

Nakon preslagivanja, točka $\mathbf{x}_0$ predstavlja najlošiju točku, a $\mathbf{x}_n$ najbolju točku simpleksa. Izuzimanjem najlošije točke $\mathbf{x}_0$, dobijemo simpleks dimenzije n za koji možemo izračunati težište (Slika 6.15):

$$\mathbf{x}_m = \frac{1}{n}\sum_{i=1}^{n}\mathbf{x}_i.$$



Slika 6.15
Određivanje težišta reduciranog simpleksa (koji ne sadrži najlošiju točku $\mathbf{x}_0$)

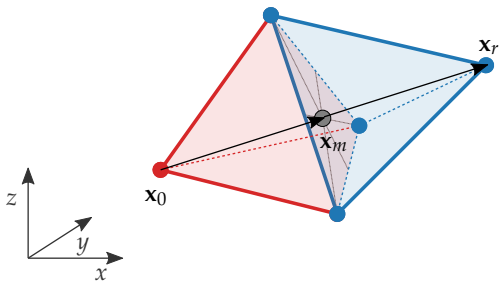
Ovako dobivena točka $\mathbf{x}_m$ je sada mjerodavna za transformacije simpleksa koje slijede.

Refleksija

Prva transformacija simpleksa koja se provodi je izračunavanje točke refleksije, koji se dobije zrcaljenjem najlošije točke $\mathbf{x}_0$ s obzirom na težište $\mathbf{x}_m$ (Slika 6.16):

$$\mathbf{x}_r = \mathbf{x}_m + \alpha(\mathbf{x}_m - \mathbf{x}_0),$$

gdje je α parametar refleksije, koji očito mora biti pozitivan da bi se refleksija ostvarila.



Slika 6.16
Određivanje nove točke zrcaljenjem najlošije točke

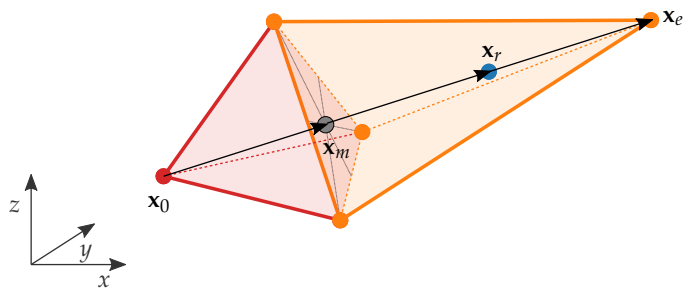
Sada se izračunava funkcija cilja u novodobivenoj točki $f_r = f(\mathbf{x}_r)$ te se na bazi te vrijednosti donosi odluka o eventualnim alternativnim transformacijama. Ako je novodobivena točka bolja od druge najlošije točke, ali lošija od najbolje točke simpleksa, tj. $f_1 > f_r \geq f_n$, ne provode se dodatne transformacije, nego se usvaja novi simpleks $(\mathbf{x}_r, \mathbf{x}_1, \ldots, \mathbf{x}_n)$ i kreće se u novu iteraciju metode.

Ekspanzija

Ako se točka refleksije pokaže kao najbolja točka od svih točaka simpleksa, tj. $f_n > f_r$, efekt refleksije se "pojačava" izračunavanjem točke ekspanzije (Slika 6.17):

$$\mathbf{x}_e = \mathbf{x}_m + \beta(\mathbf{x}_m - \mathbf{x}_0),$$

gdje je β parametar ekspanzije, s time da očito mora biti $\beta > \alpha$ da bi se ekspanzija ostvarila.



Slika 6.17
Ekspanzija simpleksa određivanjem nove, udaljenije točke u smjeru reflektirane točke

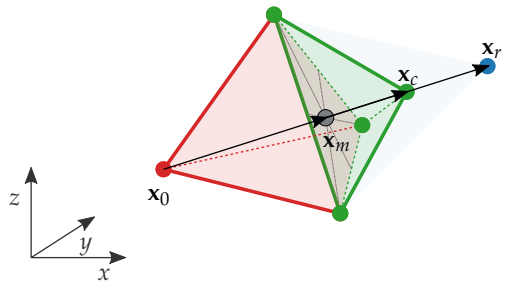
Novodobivenu točku $\mathbf{x}_e$ sada se evaluira, tj. izračunava se $f_e = f(\mathbf{x}_e)$. Ako se ona pokaže kao bolja od reflektirane točke, tj. $f_r > f_e$, usvaja se novi simpleks $(\mathbf{x}_e, \mathbf{x}_1, \ldots, \mathbf{x}_n)$ i pokreće se nova iteracija metode. U suprotnom (ako je $f_e \geq f_r$), točka ekspanzije se ignorira, usvaja se simpleks $(\mathbf{x}_r, \mathbf{x}_1, \ldots, \mathbf{x}_n)$ i pokreće nova iteracija.

Vanjska kontrakcija

Ako je točka refleksije druga najlošija točka, tj. $f_0 > f_r \geq f_1$, provodi se kontrakcija u cilju smanjenja efekta refleksije, pri čemu se izračunava nova točka (Slika 6.18):

$$\mathbf{x}_c = \mathbf{x}_m + \gamma(\mathbf{x}_m - \mathbf{x}_0),$$

gdje je γ parametar vanjske kontrakcije, s time da mora vrijediti $\gamma < \alpha$ da bi kontrakcija funkcionirala.



Slika 6.18
Vanjska kontrakcija simpleksa određivanjem nove, bliže točke u smjeru reflektirane točke

Sada se može izračunati $f_c = f(\mathbf{x}_c)$ i tako provjeriti "kvalitetu" točke $\mathbf{x}_c$: ako je $f_r > f_c$ onda se simpleks definira točkama $(\mathbf{x}_c, \mathbf{x}_1, \ldots, \mathbf{x}_n)$ i kreće se u novu iteraciju. U suprotnom (ako je $f_c \geq f_r$) prelazi se na smanjivanje simpleksa, koje je objašnjeno u nastavku.

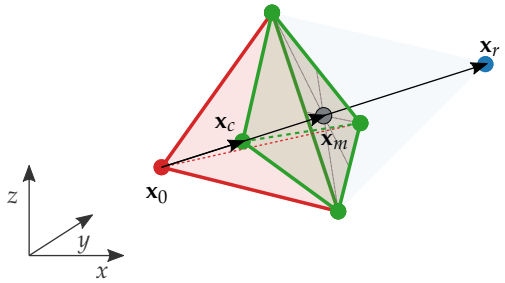
Unutarnja kontrakcija

Ako je točka refleksije najlošija od svih točaka simpleksa, tj. $f_r \geq f_0$, efekt refleksije se poništava izračunavanjem točke kontrakcije (Slika 6.19):

$$\mathbf{x}_c = \mathbf{x}_m - \delta(\mathbf{x}_m - \mathbf{x}_0),$$

gdje je δ parametar unutarnje kontrakcije ($0 < \delta < 1$).

Kao i u prethodnim koracima postupka, novodobivena točka kontrakcije se evaluira, $f_c = f(\mathbf{x}_c)$ i ako se desilo poboljšanje, tj. $f_0 > f_c$, simpleks se definira točkama $(\mathbf{x}_c, \mathbf{x}_1, \ldots, \mathbf{x}_n)$ i ide se u novu iteraciju. U suprotnom (ako je $f_c \geq f_0$) prelazi se na smanjivanje simpleksa.



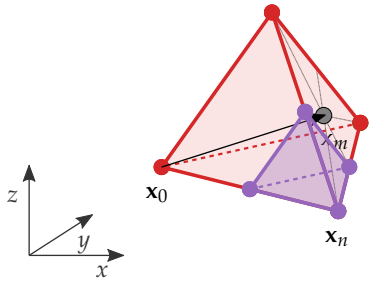
Slika 6.19
Unutarnja kontrakcija simpleksa određivanjem nove točke smještene između najlošije točke i težišta

Smanjivanje simpleksa

Smanjivanje simpleksa se poduzima u rijetkom slučaju u kojem prilikom kontrakcije, bilo unutarnje ili vanjske, nije pronađena bolja točka. Smanjivanje simpleksa podrazumijeva transformaciju svih točaka osim najbolje i to približavanjem točaka najboljoj točki (Slika 6.20):

$$\mathbf{x}_i = \mathbf{x}_n + \rho(\mathbf{x}_i - \mathbf{x}_n) \qquad i = 0, \ldots, n-1 \ ,$$

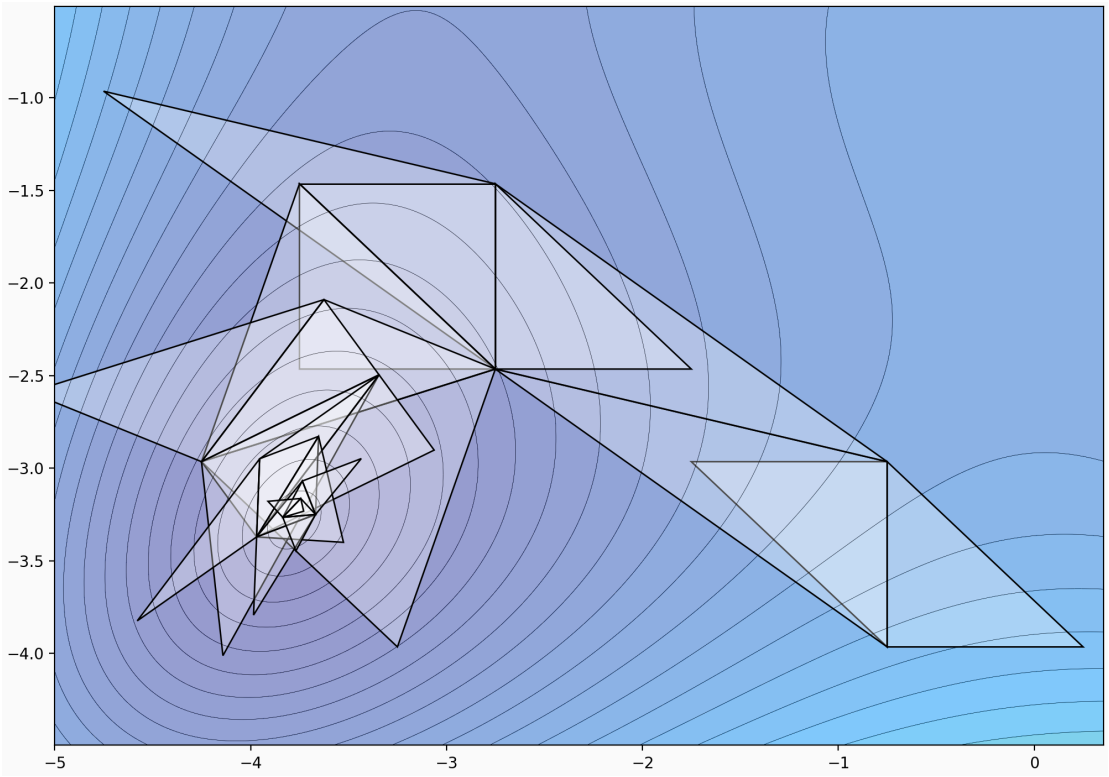
gdje je ρ parametar smanjivanja ($0 < \rho < 1$).



Slika 6.20
Smanjivanje simpleksa pomicanjem točaka prema najboljoj točki

Nakon smanjivanja, novodobivene točke se evaluiraju, tj. izračunava se $f_i = f(\mathbf{x}_i)$, $i = 0, \ldots, n-1$ i pokreće se nova iteracija sa simpleksom $(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n)$.

Opisani postupak se ponavlja dok je god veličina simpleksa veća od dopuštene greške rješenja ili dok se ne ostvari neki drugi kriterij zaustavljanja. Vizualizacija transformiranja i kretanja simpleksa prilikom traženja minimuma u dvodimenzionalnom prostoru (tj. na funkciji dvije varijable) dana je na Slici 6.21.



Slika 6.21 Konvergencija metode Nelder-Mead na primjeru funkcije dvije varijable (početni simpleks je pravokutni trokut smješten desno-dolje)

Programski kod koji implementira opisani algoritam Nelder-Mead metode dan je u nastavku (Python kod 6.7).

Python kod 6.7 Ilustrativna implementacija Nelder-Mead metode

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

# funkcija cilja
def f(X):
    x, y = X
    return (x-1)**2 + (y-2)**2

# broj varijabli funkcije cilja
n = 2

# početna točka
x0, y0 = 0, 0

# dopuštena greška konvergencije
eps = 0.1

# prostorni korak inicijalizacije simpleksa
d = 0.1

# inicijalizacija simpleksa
X = np.empty([n+1, n])
F = np.empty(n+1)
X[0,:] = np.array([x0, y0])
for i in range(1, n+1):
```

```
    E = np.zeros(n)
    E[i-1] = 1
    X[i,:] = X[0,:] + d * E

# crtanje
c1x = np.sin(np.linspace(-np.pi, np.pi))
c1y = np.cos(np.linspace(-np.pi, np.pi))
plt.plot(1 + 0.5*c1x, 2 + 0.5*c1y, 'g--')
plt.plot(1 + 1*c1x, 2 + 1*c1y, 'g--')
plt.plot(1 + 2*c1x, 2 + 2*c1y, 'g--')
plt.plot(plt.xlim(), [2, 2], 'r-.')
plt.plot([1, 1], plt.ylim(), 'r-.')
plt.plot(X[:,0], X[:,1], 'b')
plt.plot([X[-1,0], X[0,0]], [X[-1,1], X[0,1]], 'b')
plt.text(np.average(X[:,0]), np.average(X[:,1]), '0')

# metoda
alfa, beta, gama, delta, ro = 1, 2, 0.5, 0.5, 0.5

it = 0
maxit = 100

while it < maxit:
    it = it + 1
    smanjivanje = False

    # evaluacija i sortiranje
    for i in range(n+1):
        F[i] = f(X[i,:])
    poredak = np.argsort(F)[::-1]
    X = X[poredak, :]
    F = F[poredak]
    Xm = np.empty(n)
    for i in range(n):
        Xm[i] = np.average(X[1:,i])

    # refleksija
    Xr = Xm + alfa * (Xm - X[0,:])
    Fr = f(Xr)
    if F[1] >= Fr > F[n]:
        X[0,:] = Xr
        F[0] = Fr

    # ekspanzija
    elif F[n] > Fr:
        Xe = Xm + beta * (Xm - X[0,:])
        Fe = f(Xe)
        if Fr > Fe:
            X[0,:] = Xe
            F[0] = Fe
        else:
            X[0,:] = Xr
            F[0] = Fr

    # vanjska kontrakcija
    elif F[0] >= Fr > F[1]:
        Xc = Xm + gama * (Xm - X[0,:])
        Fc = f(Xc)
        if Fr > Fc:
            X[0,:] = Xc
            F[0] = Fc
```

```
        else:
            smanjivanje = True

    # unutarnja kontrakcija
    elif Fr >= F[0]:
        Xc = Xm - delta * (Xm - X[0,:])
        Fc = f(Xc)
        if F[0] > Fc:
            X[0,:] = Xc
            F[0] = Fc
        else:
            smanjivanje = True

    # smanjivanje simpleksa
    if smanjivanje:
        for i in range(n):
            X[i,:] = X[n] + ro * (X[i,:] - X[n])
            F[i] = f(X[i,:])

    # crtanje
    plt.plot(X[:,0], X[:,1], 'b')
    plt.plot([X[-1,0], X[0,0]], [X[-1,1], X[0,1]], 'b')
    plt.text(np.average(X[:,0]), np.average(X[:,1]), str(it))

    # metoda konvergirala?
    D = np.empty(n)
    for i in range(n):
        D[i] = np.sqrt(np.sum((X[i] - X[n])**2))
    if np.max(D) < eps:
        break

# rezultat
Xmin = X[np.argmin(F)]
print('minimum: ' + str(Xmin))
print('rješenje postignuto u ' + str(it) + ' iteracija')

plt.show()
```

6.5.2 Svojstva Nelder-Mead metode

Kao što je objašnjeno u opisu algoritma, Nelder-Mead metoda koristi pet parametara. Ovi parametri se mogu prilagođavati pojedinim optimizacijskim problemima koje se rješava, a tipične njihove vrijednosti jesu:

- $\alpha = 1$ - faktor refleksije
- $\beta = 2$ - faktor ekspanzije
- $\gamma = 0.5$ - faktor vanjske kontrakcije
- $\delta = 0.5$ - faktor unutarnje kontrakcije
- $\rho = 0.5$ - faktor smanjivanja

Prva prednost Nelder-Mead metode koja bi se mogla istaknuti je njena jednostavnost i intuitivnost. Nadalje, posebno je poželjna njena osobina da iziskuje mali broj evaluacija funkcije cilja, jer se u svakoj iteraciji funkcija cilja tipično izračunava samo jednom ili dvaput (korak smanjivanja simpleksa se vrlo rijetko provodi). To je čini vrlo pogodnom za optimizaciju na računski skupim funkcijama cilja, koje u tehničkoj primjeni nisu rijetkost.

Treba napomenuti da Nelder-Mead metoda nema garantiranu konvergenciju. To znači da nema garancije da će pronaći optimum, pa se ponekada može desiti da metoda "zaglavi" daleko od optimuma, gdje potroši veliki broj iteracija bez bitnog napretka. Nadalje, učinkovitost metode pada sa povećanjem broja varijabli funkcije cilja (tj. sa porastom broja dimenzija prostora pretraživanja n), iako treba priznati da to načelno vrijedi za sve optimizacijske metode. Unatoč svemu navedenom, treba zaključiti da je Nelder-Mead metoda najčešće vrlo uspješna u primjeni na nisko-dimenzionalne probleme koji nisu izrazito multimodalni, s time da vrlo često već u prvih nekoliko iteracija postiže značajna približenja optimumu.

6.6 Optimizacije pomoću scipy modula

SciPy metode za *bounded* optimizaciju. Metode koje u `scipy.optimize.minimize` podržavaju argument `bounds`:

- Nelder-Mead,
- L-BFGS-B,
- TNC,
- SLSQP,
- Powell,
- trust-constr

Zadavanja granica

Python kod 6.8 Osnovni argumenti za `scipy.optimize.minimize` funkciju

```
import scipy.optimize as spo
import numpy as np

def f(x):
    return x[0]*x[1] + x[0]**2 + x[1]**2 + x[2]**2

lb = np.array([0, 0, -2])
ub = np.array([1, 10, 2])
x0 = np.random.uniform(lb, ub)

# Pokretanje optimizacije sa zadavanjem granica varijabli:
r = spo.minimize(f, x0, bounds=zip(lb, ub))
# ili
bounds = np.array([lb, ub]).T
r = spo.minimize(f, x0, bounds=bounds)

# Pokretanje optimizacije sa zadavanjem metode:
r = spo.minimize(f, x0, method='Powell', bounds=zip(lb, ub))

# Rezultati optimizacije
r = spo.minimize(f, x0)
print(r.x)  # Optimum (vektor optimizacijskih varijabli)
print(r.fun)  # Vrijednost minimizirane funckije
print(r.nfev)  # Broj evaluacija (broj poziva funkcije cilja)
```

6.6.1 Nelder-Mead metoda

Python kod 6.9 Osnovne opcije Nelder-Mead metode u `scipy.optimize.minimize`

```
r = opt.minimize(f,  # Funkcija cilja
                 x0,  # Početna točka
                 method='Nelder-Mead',  # Metoda
                 bounds=zip(lb, ub),  # Granice
                 options={'maxiter': 1e4,  # Maksimalni broj iteracija
                          'maxfev': max_evals,  # Maksimalni broj evaluacija
                          'disp': True,  # Ispis statusa optimizacije
                          'xatol': x_tol,  # Tolerancija na opt. varijable
                          'fatol': f_tol,  # Tolerancija na funkciju
                          'adaptive': True,  # Redovito dovodi do ubrzanja
                          },)
```

6.6.2 Powellova metoda

Python kod 6.10 Osnovne opcije Powellove metode u `scipy.optimize.minimize`

```
r = opt.minimize(f,  # Funkcija cilja
                 x0,  # Početna točka
                 method='Powell',  # Metoda
                 bounds=zip(lb, ub),  # Granice
                 options={'maxiter': None,  # Maksimalni broj iteracija
                          'maxfev': max_evals,  # Maksimalni broj evaluacija
                          'disp': True,  # Ispis statusa optimizacije
                          'xtol': x_tol,  # Tolerancija na opt. varijable
                          'ftol': f_tol,  # Tolerancija na funkciju
                          },)
```

6.6.3 L-BFGS-B metoda

Limited-memory **B**royden–**F**letcher–**G**oldfarb–**S**hanno with **B**ound constraints

Python kod 6.11 Osnovne opcije L-BFGS-B metode u `scipy.optimize.minimize`

```
r = opt.minimize(f,  # Funkcija cilja
                 x0,  # Početna točka
                 method='L-BFGS-B',  # Metoda
                 bounds=zip(lb, ub),  # Granice
                 jac='3-point',  # Metoda aproksimacije Jakobijane
                 options={'maxiter': 1e4,  # Maksimalni broj iteracija
                          'maxfun': max_evals,  # Maksimalni broj evaluacija
                          'disp': False,  # Ispis statusa optimizacije
                          'eps': x_tol,  # Tolerancija na opt. varijable
                          'ftol': f_tol,  # Tolerancija na funkciju
                          'finite_diff_rel_step': x_tol,
                          # Relativni korak za aproksimaciju Jakobijane
                          },)
```

6.6.4 SLSQP metoda

Sequential **L**east **SQ**uares **P**rogramming

Python kod 6.12 Osnovne opcije SLSQP metode u `scipy.optimize.minimize`

```
r = opt.minimize(f,  # Funkcija cilja
                 x0,  # Početna točka
                 method='SLSQP',  # Metoda
                 bounds=zip(lb, ub),  # Granice
                 jac='3-point',  # Metoda aproksimacije Jakobijane
                 options={'maxiter': max_evals,  # Maksimalni broj iteracija
                          'disp': True,  # Ispis statusa optimizacije
                          'eps': x_tol,  # Tolerancija na opt. varijable
                          'ftol': f_tol,  # Tolerancija na funkciju
                          'finite_diff_rel_step': x_tol,
                          # Relativni korak za aproksimaciju Jakobijane
                          },)
```

6.6.5 TNC metoda

Truncated Newton method

Python kod 6.13 Osnovne opcije TNC metode u `scipy.optimize.minimize`

```
r = opt.minimize(f,  # Funkcija cilja
                 x0,  # Početna točka
                 method='TNC',  # Metoda
                 bounds=zip(lb, ub),  # Granice
                 jac='3-point',  # Metoda aproksimacije Jakobijane
                 options={'maxiter': max_evals,  # Maksimalni broj iteracija
                          'maxfun': max_evals,  # Maksimalni broj evaluacija
                          'disp': False,  # Ispis statusa optimizacije
                          'eps': x_tol,  # Tolerancija na opt. varijable
                          'xtol': x_tol,
                          # Apsolutni korak za aproksimaciju Jakobijane
                          'ftol': f_tol,  # Tolerancija na funkciju
                          'finite_diff_rel_step': x_tol,
                          # Relativni korak za aproksimaciju Jakobijane
                          },)
```

6.6.6 trust-constr

Trust Region Methods

Python kod 6.14 Osnovne opcije Trust Region metode u `scipy.optimize.minimize`

```
r = opt.minimize(f,  # Funkcija cilja
                 x0,  # Početna točka
                 method='trust-constr',  # Metoda
                 bounds=zip(lb, ub),  # Granice
                 jac='3-point',  # Metoda aproksimacije Jakobijane
                 options={'maxiter': max_evals,  # Maksimalni broj iteracija
                          'disp': False,  # Ispis statusa optimizacije
                          'verbose': 1,  # Detaljnost ispisa
                          'xtol': x_tol,
                          # Apsolutni korak za aproksimaciju Jakobijane
                          'finite_diff_rel_step': x_tol,
                          # Relativni korak za aproksimaciju Jakobijane
                          },)
```

6.7 Optimizacija rojem čestica

Stefan Ivić

Optimizacija rojem čestica (*Particle Swarm Optimization*, PSO) je stohastička optimizacijska metoda temeljena na idejama opaženim na kretanju jata i rojeva.

6.7.1 Razvoj metode

Reynolds (1987) je predložio model ponašanja jata, prema kojem se svaki član jata ponaša prema slijedećim pravilima:

- Držanje razmaka - svaki član jata se pokušava udaljiti od susjednih ako su mu oni preblizu,
- Prilagođavanje smjera - svaki član jata giba se u usrednjenom smjeru kretanja svojih susjeda,
- Kohezija - svaki član jata ostaje blizu svojih susjeda tj. ostaje u jatu.

Kennedy i Eberhart (1995) su dodali 'gnijezdo' u pojednostavljenu simulaciju temeljenu na Reynoldsovom modelu. Simulacija je obuhvaćala sljedeće utjecaje gnijezda na jedinke jata:

- Gnijezdo privlači svaku jedinku,
- Svaka jedinka pamti gdje je bila najbliža gnijezdu.
- Svaka jedinka razmijenjuje informacije sa susjedima o lokaciji najbližoj gnijezdu.

Nakon dovoljno vremena provođenja simulacije, sve su jedinke došle do gnijezda. Postavlja se pitanje što se dešava sa sustavom ako se gnijezdo giba prema nepoznatoj funkciji? Da li će jedinke stići do gnijezda?

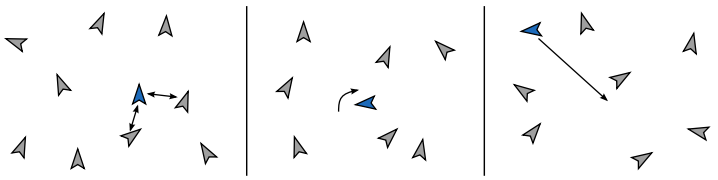
$$\mathbf{v}_i^{t+1} = \mathbf{v}_i^t + \phi_1 \mathbf{U}_1^t(\mathbf{pb}_i^t - \mathbf{x}_i^t) + \phi_2 \mathbf{U}_2^t(\mathbf{gb}^t - \mathbf{x}_i^t) \tag{6.33}$$

$$\mathbf{x}_i^{t+1} = \mathbf{x}_i^t + \mathbf{v}_i^{t+1} \tag{6.34}$$

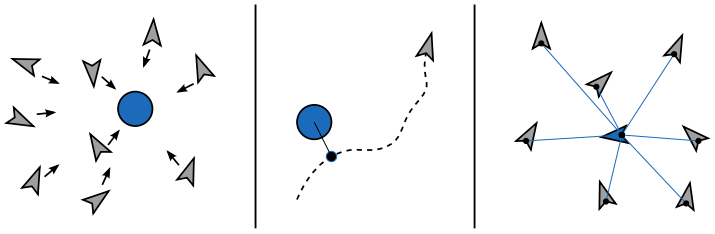
Shi i Eberhart (1998) uvode inerciju čestice koja je kontrolirana preko faktora inercije w^t.

$$\mathbf{v}_i^{t+1} = w^t \mathbf{v}_i^t + \phi_1 \mathbf{U}_1^t(\mathbf{pb}_i^t - \mathbf{x}_i^t) + \phi_2 \mathbf{U}_2^t(\mathbf{gb}^t - \mathbf{x}_i^t) \tag{6.35}$$

U osnovnom obliku, faktor inercije je konstantan u vremenu $w_t = w$. U slučaju $w \geq 1$, brzine čestica rastu u vremenu (čestice ubrzavaju) a formacija roja divergira. Usporavanje čestica i smanjivanje brzine do nule postiže se sa $w < 1$. Veći faktor



Slika 6.22
Ponašanje jedinki u Reynoldsovom modelu. *Odvajanje, usmjeravanje i kohezija.*



Slika 6.23
Utjecaj gnijezda na jedinke jata.

inercije omogućava lakše globalno pretraživanje, sa povećanjem rasapa čestica, dok manji w omogućuje bolje lokalno pretraživanje i bržu konvergenciju. Odabir faktora inercije treba temeljiti na karkateristikama funkcije cilja i svodi se na balansiranje širine pretraživanja i brzine konvergencije.

Prema (van den Bergh, 2001)

$$w > \frac{c_1 + c_2}{2} - 1 \tag{6.36}$$

garantira konvergenciju putanja čestica, a preporučeni faktor inercije je $w = 0.73$ (Bratton i Kennedy, 2007).

6.7.2 Korištenje PSO metode pomoću modula inspyred

Kako instaliraty inspyred: http://inspyred.github.com/index.html

```
user@machine:~>sudo pip install inspyred
```

Python kod 6.15 Particle Swarm Optimization na primjeru Rastrigin funkcije

```
from random import Random
from time import time
from math import cos, pi
import inspyred

# Random generiranje
def generate_rastrigin(random, args):
    size = args.get('num_inputs', 10)
    return [random.uniform(-5.12, 5.12) for i in range(size)]

# Evaluacija (funkcija cilja)
def evaluate_rastrigin(candidates, args):
    fitness = []
    for cs in candidates:
        fit = 10 * len(cs) + sum([((x - 1)**2 - 10 * cos(2 * pi * (x - 1))) for x in
             cs])
        fitness.append(fit)
    return fitness

rand = Random()
rand.seed(int(time()))
pso = inspyred.swarm.PSO(rand)
pso.terminator = inspyred.ec.terminators.evaluation_termination

final_pop = pso.evolve(generator=generate_rastrigin,
                       evaluator=evaluate_rastrigin,
                       pop_size=100,
                       maximize=False,
                       bounder=inspyred.ec.Bounder(-5.12, 5.12),
                       max_evaluations=20000,
```

```
                      mutation_rate=0.25,
                      num_inputs=5)

# Sort and print the best individual, who will be at index 0.
final_pop.sort(reverse=True)
print(final_pop[0])
```

Reference i popisi

Bibliografija

D. Bratton i J. Kennedy. Defining a standard for particle swarm optimization. In *Swarm Intelligence Symposium, 2007. SIS 2007. IEEE*, pages 120–127. IEEE, 2007.

J. W. Cooley i J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

L. V. Kantorovič. Mathematical methods of organizing and planning prodution. *Management Science*, 6:363–422, 1960. (in Russian 1939).

J. Kennedy i R. Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948. IEEE, 1995.

J. A. Nelder i R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

C. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *ACM SIGGRAPH Computer Graphics*, volume 21, pages 25–34. ACM, 1987.

Y. Shi i R. Eberhart. A modified particle swarm optimizer. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pages 69–73. IEEE, 1998.

F. van den Bergh. *An Analysis of Particle Swarm Optimizers., Submitted Ph. D.* PhD thesis, thesis, University of Pretoria, Pretoria, 2001.

W. Winston i J. Goldberg. *Operations Research: Applications and Algorithms*. Thomson Brooks/Cole, 2004. ISBN 9780534423582. URL https://books.google.hr/books?id=tg5DAQAAIAAJ.

Popis programskog koda

Kazalo

Autorska prava i licence

Slike na naslovnicama poglavlja

- Slika na naslovnoj stranici objavljena je pod licencom "Free for commercial use with attribution" na http://www.freepik.com (autor: evening_tao)
- Slika Sadržaja objavljena je bez ograničenja na http://www.freepik.com
- Slika poglavlja Osnove Pythona objavljena je pod licencom Creative Commons Zero (CC0) na https://www.pexels.com/photo/close-up-of-computer-keyboard-257919/
- Slika poglavlja NumPy objavljena je pod Attribution-ShareAlike 2.0 Generic (CC BY-SA 2.0) licencom na https://www.flickr.com/photos/raselased/6223987881.
- Slika poglavlja Vizualizacija podataka objavljena je pod CC0 Creative Commons licencom na https://pixabay.com/en/sea-water-vacation-holidays-blue-2755908/.
- Slika poglavlja Numeričke metode objavljena je pod CC0 Creative Commons licencom na https://pixabay.com/en/kings-cross-pillar-geometry-grid-1031629/.
- Slika poglavlja Modeliranje pomoću diferencijalnih jednadžbi objavljena je pod CC0 Creative Commons licencom na https://pixnio.com/nature-landscapes/coast/sea-seashore-water-waves-beach-coast
- Slika poglavlja Optimizacijske metode objavljena je pod CC0 Creative Commons licencom na https://pixabay.com/en/serfaus-austria-mountains-snow-1255818/
- Slika poglavlja Reference i popisi objavljena je pod CC0 Creative Commons licencom na https://pixabay.com/en/book-open-pages-library-books-408302/