UNIVERZA V LJUBLJANI
Fakulteta za strojništvo

Janko Slavič

Programiranje in numerične metode v ekosistemu Pythona

Ta dokument je okrnjena verzija izvršljivega učbenika, ki je prosto dostopen na spletnem naslovu https://github.com/jankoslavic/pypinm

Ljubljana, 2017

Naslov dela:	Programiranje in numerične metode v ekosistemu Pythona
Avtor:	izr. prof. dr. Janko Slavič u.d.i.s
Recenzenta:	izr. prof. dr. Jože Petrišič, univ. dipl. ing. mat.
	prof. dr. Janez Demšar, univ. dipl. inž. rač. in inf.
Jezikovni pregled:	Andreja Cigale, prof. slov.

CIP - Kataložni zapis o publikaciji
Narodna in univerzitetna knjižnica, Ljubljana

za popraviti 531.3(075.8)
za popraviti 532.5(075.8)

Slavič, Janko, 1978-
Programiranje in numerične metode v ekosistemu Pythona/ Janko Slavič. - - Ljubljana: Fakulteta za strojništvo, 2017

ISBN 978-961-6980-45-6

???

Kazalo

Predgovor

Ta učbenik je v izvorni obliki t. i. izvršljiv učbenik. Gre za nov tip učbenikov, kjer sta tekst in koda združena v en dokument; bralec pa bere in poganja kodo, pri čemer lahko kodo in tekst poljubno spreminja. Bralec izvršljivega učbenika ima na razpolago vse izvorne datoteke. Ta izvršljivi učbenik je prosto dostopen na spletnem naslovu https://github.com/jankoslavic/pypinm. Dokument, ki ga berete, je konverzija izvornih datotek v pdf formatu; pri tej konverziji se izgubi izvršljivost, interaktivnost in delno tudi oblika. Bralcu torej priporočam, da uporablja izvršljiv spletni učbenik, ki se tudi sproti posodablja.

Učbenik je namenjen študentom pri predmetih Numerične metode ter Programiranje in numerične metode, ki se izvajata na Fakulteti za strojništvo, Univerze v Ljubljani.

Pri izdelavi učbenika se želim zahvaliti obema recenzentoma, ki sta mi pomagala, da je spletni učbenik boljši. Izr. prof. dr. Jože Petrišič mi je tako predvsem pomagal pri vsebinah iz numeričnih metod, prof. dr. Janez Demšar pa pri vsebinah iz programiranja. Zahvala gre tudi asist. Domnu Gorjupu za tehnično pomoč in Andreji Cigale za skrbno lektoriranje.

Zahvaljujem se tudi družini; takšen učbenik zahteva veliko prilagajanja.

Na koncu pa se bi želel zahvaliti bralcu, ker zanj sem to knjigo pisal!

Janko Slavič
November, 2017.

Poglavje 1

Uvod v Python

1.1 Uvod

1.1.1 Kaj je ekosistem Pythona?

Marsikaterega bralca beseda *ekosistem* zmoti/zmede. Slovar slovenskega knjižnega jezika[1] danes pozna samo rabo v povezavi z ekologijo. V računalništvu pa se v zadnjih desetletjih beseda ekosistem uporablja tudi v kontekstu t. i. digitalnega ekosistema; v primeru Pythona to pomeni vso množico različih paketov (samo pypi.python.org[2] trenutno hrani 117 tisoč različnih paketov), ki temeljijo na programskem jeziku Python in so med seboj povezani.

Ekosistem Pythona je veliko več kot pa samo programski jezik Python! Ta knjiga se v okviru prvih štirih predavanj osredotoča na tisti del tega ekosistema, ki je zanimiv in koristen predvsem za inženirske poklice.

Python je **visokonivojski** programski jezik, z visokim nivojim abstrakcije. **Nizkonivojski** programski jeziki, v nasprotju, imajo nizek nivo abstrakcije (npr. strojna koda).

Poglemo si izračun Fibonaccijevega števila[3], ki sledi definiciji:

$$F_1 = F_2 = 1 \qquad \text{in} \qquad F_n = F_{n-1} + F_{n-2}.$$

Izračun F_n v strojni kodi[4]:

```
8B542408 83FA0077 06B80000 0000C383
FA027706 B8010000 00C353BB 01000000
B9010000 008D0419 83FA0376 078BD98B
C84AEBF1 5BC3
```

MASM koda

[1] http://www.fran.si/130/sskj-slovar-slovenskega-knjiznega-jezika/3539971/ekosistem?page=4&Query=sistem&All=sistem&FilteredDictionaryIds=130&View=1

[2] https://pypi.python.org/pypi

[3] http://sl.wikipedia.org/wiki/Fibonaccijevo_%C5%A1tevilo

[4] http://en.wikipedia.org/wiki/Low-level_programming_language

```
fib:
    mov edx, [esp+8]
    cmp edx, 0
    ja @f
    mov eax, 0
    ret

    @@:
    cmp edx, 2
    ja @f
    mov eax, 1
    ret

    @@:
    push ebx
```

C

```c
int Fibonacci(int n)
{
   if ( n == 0 )
      return 0;
   else if ( n == 1 )
      return 1;
   else
      return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

Python

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Python omogoča visok nivo abstrakcije in zato abstraktno kodo. Sedaj si bomo pogledali primer kode Python:

```python
števila = [1, 2, 3, 4, 5] # seznam števil
[število for število in števila if število%2]
```

Čeprav v tem trenutku še nimamo dovolj znanja, poskusimo razumeti/prebrati:

1. prva vrstica definira seznam števil; vse kar sledi znaku # pa predstavlja komentar
2. rezultat druge vrstice je seznam, kar definirata oglata oklepaja; znotraj oklepaja beremo:
   - vrni `število` za vsako (**for**) `število` v (**in**) seznamu `števila`, če (**if**) je pri deljenju `število` z 2 ostanek 1.

Vidimo, da je nivo abstrakcije res visok, saj smo potrebovali veliko besed, da smo pojasnili vsebino!

Poglejmo rezultat:

```
In [1]: števila = [1, 2, 3, 4, 5] # seznam števil
        [število for število in števila if število%2]
```

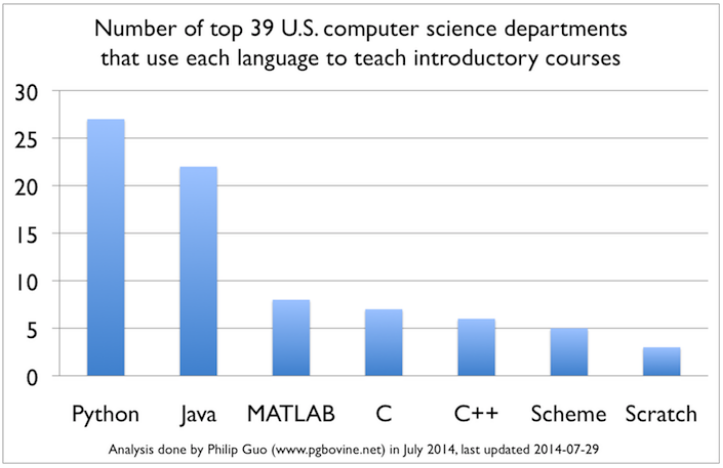
```
Out[1]: [1, 3, 5]
```

Kratek pregled začetkov nekaterih programskih jezikov: * 1957 Fortran, * 1970 Pascal, * 1972 C * 1980 C++, * 1984 Matlab, * 1986 Objective-C * **1991 Python**, * 1995 Java, * 1995 Delphi * 2009 Go, 2012 Julija

1.2 Kdo se uči Python, zakaj ga uporabljati?

ZDA

Python je jezik, ki se ga najpogosteje učijo na najboljših Ameriških univerzah.[5]



Na univerzi Berkeley v začetku študijskega leta 2017/18[6] (več kot 1200 študentov pri predmetu *Data science*, ki temelji na tehnologiji Jupyter notebook).

```
In [2]: %%html
        <blockquote class="twitter-tweet" data-lang="en"><p lang="en" dir="ltr">First day of class in F
        <script async src="//platform.twitter.com/widgets.js" charset="utf-8"></script>
```

```
<IPython.core.display.HTML object>
```

Python v zadnjih 10 letih pospešeno pridobiva na popularnosti in je danes eden od najboljših (IEEE Spectrum[7].) in tudi najpopularnejših programskih jezikov (Why is Python Growing so Quickly?[8]).

Slovenija

- FRI: Programiranje 1[9]
- FMF: Naloge na www.projekt-tomo.si[10], poiščite: Prvi koraki v Python, Drugi koraki v Python, Python za začetnike

[5] http://goo.gl/mJC1P8
[6] https://youtu.be/xuNj5paMuow?t=13m18s
[7] https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages
[8] https://stackoverflow.blog/2017/09/14/python-growing-quickly/
[9] https://ucilnica.fri.uni-lj.si/course/view.php?id=166
[10] https://www.projekt-tomo.si

FS

- Numerične metode[11]
- Programiranje in numerične metode[12]
- Poletna šola HPC[13]

1.2.1 Uporaba Pythona?

Nekatera mednarodna podjetja: *Nasa, Google, Cern.*

Nekatera podjetja v Sloveniji: *Kolektor, Iskra Mehanizmi, Hidria, Mahle Letrika, Domel, ebm-papst Slovenia (prej Ydria Motors), Cimos, Eti.*

Nekateri programi, ki imajo vgrajeno skriptno podporo za Python: *Abaqus, Ansys Workbench, MSC (Adams, SimXpert), ParaView, LMS Siemens, AVL.*

Nekateri komercialni programi v Pythonu: *BitTorrent, Dropbox.*

Obširni seznam je tukaj[14].

1.2.2 Python vs Matlab

Ker ni bilo boljše možnosti, je bil v preteklosti na področju strojništva bolj popularen Matlab; Matlab je še vedno močno orodje, vendar je danes ekosistem Pythona za večino opravil bistveno močnejše orodje. Na spletu boste našli ogromno primerjav, npr.: Python vs Matlab[15].

Kratka primerjave glede na vir[16]:

Sintaksa je načeloma zelo podobna. Pomembna razlika je, da se indeksi pri **Matlabu** začnejo z 1, v **Pythonu** pa z 0.

1.3 Namestitev Pythona

Pojdite na continuum.io[17] ter namestite *Anaconda* distribucijo Pythona; *pazite, da izberete verzijo 3.6* (64 bit). Potem sledite video navodilom[18]!

1.3.1 Uporaba githuba repozitorija in posodobitve

Ta knjiga se vedno dopolnjuje, zadnja verzija je dosegljiva na:

- v izvorni obliki na naslovu github.com/jankoslavic/pypinm[19],
- v spletni obliki na naslovu jankoslavic.github.io/pypinm.io[20].

[11] http://lab.fs.uni-lj.si/ladisk/?what=incfl&flnm=NM.php
[12] http://lab.fs.uni-lj.si/ladisk/?what=incfl&flnm=PiNM.php
[13] http://hpc.fs.uni-lj.si/python-intro
[14] http://en.wikipedia.org/wiki/List_of_Python_software
[15] http://www.pyzo.org/python_vs_matlab.html
[16] http://www.southampton.ac.uk/~fangohr/training/python/pdfs/Python-for-Computational-Science-and-Engineering-slides.pdf
[17] http://continuum.io/downloads
[18] https://www.youtube.com/watch?v=k_fJJ7Ak33c
[19] https://github.com/jankoslavic/pypinm/
[20] https://jankoslavic.github.io/pypinm.io/

Selected criteria:

	Fortran	C	C++	Matlab	Python
performance	+	+	+	o	o
object orientation	−	−	+	−	+
exceptions	−	−	+	−	+
open source	+	+	+	−	+
easy to learn	o+	o	o−	+	+

legend:

+ = good/yes
o = so-so
− = poor/no

Tukaj smo pripravili video navodila[21] za kloniranje spletnega repozitorija in izvajanje posodobitev.

1.3.2 pyCharm

Namestite še integrirano razvojno okolje pyCharm[22] (glejte Community Edition).

pyCharm sicer zahteva relativno dober računalnik (še posebej pri pri prvem zagonu, ko izvaja indeksiranje datotek), a za večji del te knjige ni potreben. Za nastavitve, potrebne za kloniranje in posodabljanje repozitorja, glejte drugi del teh video navodil[23].

1.3.3 Nameščanje dodatkov in posodobitev

Na pypi.python.org[24] se nahaja več kot 117 tisoč različnih paketov! Namestimo jih iz ukazne vrstice* (angl. *command prompt*) tako:

- z ukazom **pip**,
- bolj popularne pakete lahko namestim tudi z ukazom **conda**,
- namestitveni program s strani www.lfd.uci.edu/~gohlke/pythonlibs/[25].

* Ukazno vrstico prikličete v operacijskem sistemu Windows 10 tako, da kliknite `Štart"` in napišite `čommand prompt"` ter pritisnete `"enter"`.

Podrobnosti nameščanja dodatkov si bomo pogledali naslednjič. Da boste imeli zadnjo verzijo paketov, izvedite naslednje:

[21] https://www.youtube.com/watch?v=2uUcGjWY224
[22] https://www.jetbrains.com/pycharm/download/
[23] https://www.youtube.com/watch?v=2uUcGjWY224
[24] https://pypi.python.org/pypi
[25] http://www.lfd.uci.edu/~gohlke/pythonlibs/

1. Odprite ukazno vrstico in sprožite ukaz za posodobitev *Anaconde*:

   conda update conda

- nato posodobite še vse nameščene pakete:

  conda update --all

1.4 Jupyter notebook

Jupyter notebook[26] je interaktivno okolje, ki deluje kot spletna aplikacija in se lahko uporablja z različnimi programskimi jeziki (npr. Python, Julia, R, C, Fortran ... iz prvih treh je nastalo tudi ime JuPyt[e]R). Jupyter notebook je najlažje preizkusiti tukaj: try.jupyter.org[27].

Jupyter notebook zaženemo (v poljubni mapi) iz ukazne vrstice z ukazom: `jupyter notebook`.

Opombi:

- tukaj predpostavimo, da je pot do datoteke jupyter[.exe] dodana v spremenljivke okolja; distribucija *Anaconda* s privzetimi nastavitvami to uredi pravilno, sicer iščite pomoč na sletu, npr. tukaj[28],
- če želite ukazno vrstico sprožiti v poljubni mapi, potem v *explorerju* pridržite `shift` in kliknite mapo z desnim gumbom; nato izberite *Odpri ukazno vrsticu tukaj / Open command window here*.

Do Jupyter strežnika dostopamo prek spletnega brskalnika. Najprej se vam prikaže seznam map in datotek do katerih lahko dostopate v okviru mape, v kateri ste sprožili ukaz `jupyter notebook`. Nato poiščite v desnem kotu zgoraj ikono *new* in nato kliknite *Python 3*. S tem boste zagnali vaš prvi Jupyter notebook (v okviru jedra Python). Najdite ukazno vrstico in na desni strani kliknite na *Help* in nato *User interface tour*. Poglejte si še *Help/Keyboard shortcuts*.

Jupyter notebook je sestavljen iz t. i. *celic* (angl. cell).

Celica ima dve stanji:

- **Ukazno stanje / Command mode**: v to stanje se vstopi s pritiskom [escape],
- **Stanje za urejanje / Edit mode** v to stanje se vstopi s pritiskom [enter].

V obeh primerih se celico izvrši s pritiskom [shift + enter].

Celice so lahko različnega tipa; tukaj bomo uporabljali dva tipa:

- **Code**: v tem primeru celica vsebuje programsko kodo (bližnjica v ukaznem stanju je tipka [y]),
- **Markdown**: v tem primeru celica vsebuje besedilo oblikovano po standardu *Markdown* (bližnjica v ukaznem stanju je tipka [m]).

1.4.1 Markdown

Jupyter notebook omogoča zapis v načinu *markdown*, ki ga je v letih 2003-2004 predstavil John Gruber[29]. Namen markdowna je, da se pisec osredotoča na vsebino, pri tem pa za oblikovanje uporablja nekaj preprostih pravil. Pozneje se markdown lahko prevede v spletno obliko, v pdf (npr. prek LaTeX-a) itd. Bistvena pravila (vezana predvsem na uporabu na Githubu) so podana tukaj[30]; spodaj jih bomo na kratko ponovili.

[26] https://jupyter.org/
[27] https://try.jupyter.org/
[28] https://www.google.si/search?q=add+program+to+path+windows+10
[29] https://daringfireball.net/projects/markdown/
[30] https://help.github.com/articles/basic-writing-and-formatting-syntax/

Naslove definiramo z znakom #:

- # pomeni naslov prve stopnje,
- ## pomeni naslov druge stopnje itd*.

* Če pogledate kodo opazite, da se znak pojavi pred znakom # zato, da se *prikaže* sam znak # in ne uporabi njegovo funkcijo za naslov prve stopnje. Podobno je tudi pri drugih posebnih znakih.

Oblikovanje besedila definirajo posebni znaki; če želimo besedo (ali več besed) v *poševni* obliki, moramo tako besedilo vstaviti med dva znaka *: tekst v stanju za urejanje v obliki *primer* bi se prikazal kot *primer*.

Pregled simbolov, ki definirajo obliko:

- *poševno*:*
- **krepko**: **
- ~~prečrtano~~: ~~
- `simbol`: `
- `funkcije`: “
- `blok kode`: “‘.

Opomba: za primer bloka kode Python glejte uvodno poglavje, primer Fibonacci.

V okviru markdowna lahko uporabljamo tudi LaTeX sintakso. LaTeX je zelo zmogljiv sistem za urejanje besedil; na Fakulteti za strojništvo, Univerze v Ljubljani, so predloge za diplomske naloge pripravljene v LaTeXu[31] in Wordu. Tukaj bomo LaTeX uporabljali samo za pisanje matematičnih izrazov.

Poznamo dva načina:

- vrstični način (angl. *inline*): se začne in konča z znakom $, primer: $a^3 = \sqrt{365}$,
- blokovni način (angl. *block mode*): se začne in konča z znakoma $$, primer:

$$\sum F = m\,\ddot{x}$$

Zgoraj smo že uporabljali sezname, ki jih začnemo z znakom * v novi vrstici:

- lahko pišemo sezname
  - z več nivoji
    - 3. nivo
  - 2. nivo
- lahko pišemo *poševno*
- lahko pišemo **krepko**.

Če želimo oštevilčene sezname, začnemo z "1."nato pa nadaljujemo z *:

1. prvi element

- drugi
- naslednji ...

V dokument lahko vključimo tudi *html* sintakso (angl. *Hypertext Markup Language* ali po slovensko jezik za označevanje nadbesedila). Kot html najenostavneje vključimo slike.

Uporabimo:

[31] http://lab.fs.uni-lj.si/ladisk/?what=incfl&flnm=latex.php

```
![](./fig/ladisk.png)
```

Za prikaz:

Univerza *v Ljubljani*
Fakulteta za *strojništvo*
Laboratorij za dinamiko strojev in konstrukcij

Vključimo lahko tudi zunanje vire (npr. video posnetek iz Youtube). Ker gre za morebitno varnostno tveganje, tega ne naredimo v celici tipa *markdown* kakor zgoraj, ampak v celici tipa *code*, pri tem pa uporabimo t. i. magični ukaz (angl. *magic*) `%%html`, ki definira, da bo celoten blok v obliki html sintakse (privzeto so celice tipa *code* Python koda):

```
In [3]: %%html
        <iframe width="560" height="315" src="https://www.youtube.com/embed/VLcPd07-gnk" frameborder="0

<IPython.core.display.HTML object>
```

Magičnih ukazov je v notebooku še veliko (glejte dokumentacijo[32]) in nekatere bomo spoznali pozneje, ko jih bomo potrebovali!

Doslej smo že večkrat uporabili sklicevanje na zunanje vire, to naredimo tako:

```
[Ime povezave](naslov do vira)
```

Primer: `[Spletna stran laboratorija Ladisk](www.ladisk.si)` se oblikuje kot: Spletna stran laboratorija Ladisk[33].

V okviru markdowna lahko pripravimo preproste tabele. Priprava je relativno enostavna: med besede enostavno vstavimo navpične znake `|` tam, kjer naj bi bila navpična črta. Po naslovni vrstici sledi vrstica, ki definira poravnavo v tabeli.

Primer:

```
|   | Masa [kg]  | Višina [cm]  |
|:-|-:|:-:|
| Lakotnik  | 80  | 140  |
| Trdonja | 30  | 70  |
| Zvitorepec  | 40  | 100  |
```

Rezultira v:

	Masa [kg]	Višina [cm]
Lakotnik	80	140
Trdonja	30	70
Zvitorepec	40	100

Opazimo, da dvopičje definira poravnavo teksta v stolpcu (levo `:-`, desno `-:` ali sredinsko `:-:`).

[32] http://ipython.readthedocs.io/en/stable/interactive/magics.html
[33] http:/www.ladisk.si

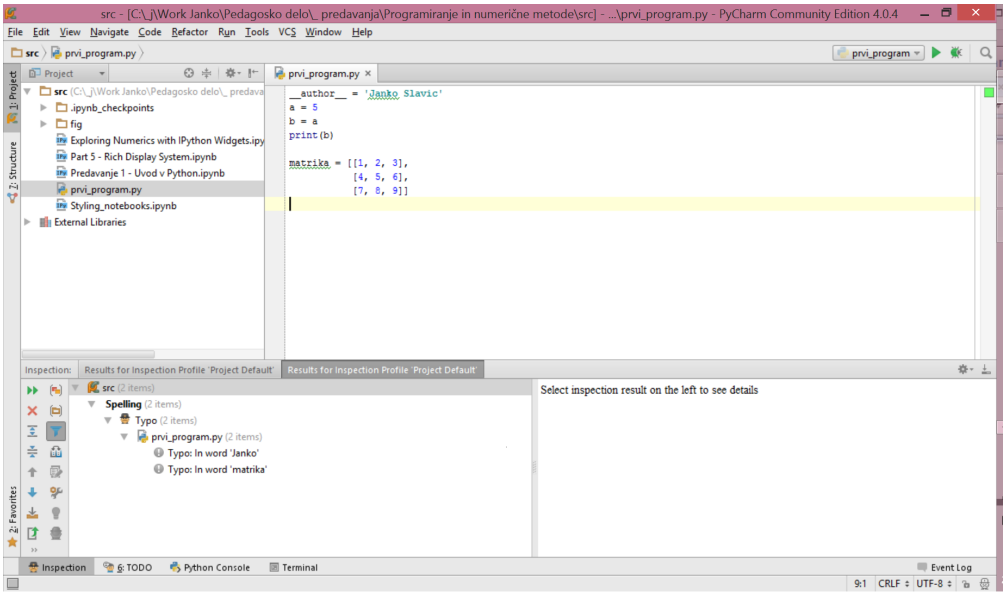
1.5 Prvi program in PEP

Sedaj smo pripravljeni na prvi program napisan v Pythonu:

```
a = 5.0
print(a)
```

Program lahko zapišemo v poljubnem urejevalniku tekstovnih datotek (npr. *Beležka/Notepad*) in ga shranimo v datoteko `ime.py`.

Primer kode[34] v okolju *pyCharm*:



To datoteko nato poženemo v ukazni vrstici (predpostavimo, da je pot do datoteke python.exe dodana v spremenljivke okolja; distribucija *Anaconda* s privzetimi nastavitvami to uredi pravilno, sicer glejte tale vir[35]):

```
python ime.py
>>> 5.0
```

Z znaki >>> označimo rezultat, ki ga dobimo v ukazni vrstici.

Zgornji primer izvajanja Python programa predstavlja t. i. *klasični način poganjanja programov* od začetka do konca. Tukaj bomo bolj pogosto uporabljali t. i. *interaktivni način*, ko kodo izvajamo znotraj posamezne celice tipa *code*:

```
In [4]: a = 5.0  # to je komentar
        print(a) # izpiše vrednost a
```

```
5.0
```

[34] ./moduli/prvi_program.py
[35] https://docs.python.org/3/using/windows.html#configuring-python

1.5.1 PEP - Python Enhancements Proposal

Python ima visoke standarde glede kakovosti in estetike. Že zgodaj v razvoju jezika se je uveljavil princip predlaganja izboljšav prek t. i. *Python Enhancements Proposal* (PEP na kratko).

PEP8

Eden od bolj pomembnih je PEP8[36], ki definira stil:

- zamik: 4 presledki,
- en presledek pri operatorju `=`, torej: `a = 5` in ne `a=5`,
- spremenljivke in imena funkcij naj bodo opisne, pišemo jih s podčrtajem: `sirina_valja = 5`,
- za razrede uporabljamo t. i. format CamelCase.

- pri aritmetičnih operaterjih postavimo presledke po občutku: `x = 3*a + 4*b` ali `x = 3 * a + 4 * b`,
- za oklepajem in pred zaklepajem ni presledka, prav tako ni presledka pred oklepajem ob klicu funkcije: `x = sin(x)`, ne pa `x = sin( x )` ali `x = sin (x)`,
- presledek za vejico: `range(5, 10)` in ne `range(5,10)`,
- brez presledkov na koncu vrstice ali v prazni vrstici,

- znotraj funkcije lahko občasno dodamo po *eno* prazno vrstico,
- med funkcije postavimo dve prazni vrstici,
- vsak modul uvozimo (`import`) v ločeni vrstici,
- najprej uvozimo standardne pythonove knjižnice, nato druge knjižnice (npr. numpy) in na koncu lastne.

Nujno: uporabite *inspect code* v pyCharm!

PEP20

PEP20[37] je naslednji pomemben PEP, ki se imenuje tudi *The zen of Python*, torej *zen Pythona*; to so vodila, ki se jih poskušamo držati pri programiranju. Tukaj navedimo samo nekatera vodila:

- Lepo je bolje kot grdo.
- Preprosto je bolje kot zakomplicirano.
- Premočrtno je bolje kot gnezdeno.
- Berljivost je pomembna.
- Posebni primeri niso dovolj posebni, da bi prekršili pravilo
- Če je implementacijo težko pojasniti, je verjetno slaba ideja; če jo je lahko, je mogoče dobra.

1.6 Osnove Pythona

1.6.1 Osnovni podatkovni tipi

Dinamično tipiziranje

Python je dinamično tipiziran jezik. Tipi spremenljivk se ne preverjajo. Posledično nekatere operacije niso mogoče na nekaterih tipih oz. so prilagojene tipu podatka (pozneje bomo na primer pogledali množenje

[36] https://www.python.org/dev/peps/pep-0008/
[37] https://www.python.org/dev/peps/pep-0020/

niza črk). Tip se določi pri dodelitvi vrednosti, poglejmo primer:

```
In [5]: a = 1 # to je komentar: a je celo število (integer)
        a

Out[5]: 1
```

Tip prikažemo z ukazom `type`:

```
In [6]: type(a)

Out[6]: int
```

Vrednost `a` lahko prepišemo z novo vrednostjo:

```
In [7]: a = 1.0  # to je sedaj število s plavajočo vejico (float)
        a        # če `a` tako zapišemo v zadnji vrstici, se bo izpisala vrednost

Out[7]: 1.0

In [8]: type(a)

Out[8]: float
```

Za izpis vrednosti sicer lahko uporabimo tudi funkcijo `print()`, ki jo bomo podrobneje spoznali pozneje:

```
In [9]: print(a)

1.0
```

Dokumentacijo funkcije lahko kličemo s pritiskom [shift + tab]. Npr: napišemo `pri` in pritisnemo [shift + tab] ter se bo v pojavnem oknu prikazala pomoč; če pridržimo [shift], se z dodatnim pritiskom na [tab] prikaže razširjena pomoč, z nadaljnjim dvojnim pritiskom [tab] (skupaj torej 4-krat) se pomoč prikaže v novem oknu.

Do pomoči dostopamo tudi, tako, da pred funkcijo postavimo `?` (za pomoč) ali `??` (prikaže tudi izvorno kodo, če je ta na voljo).

Primer:

```
?print
??print
```

Logični operatorji

Logični operatorji so (vir[38]): * `or`, primer: `a or b`; opomba: drugi element se preverja samo, če prvi ni res, * `and`, primer: `a and b`; opomba: drugi element se preverja samo, če prvi je res, * `not`, primer: `not a`; opomba: ne-logični operatorji imajo prednost: `not a == b` se interpretira kot `not (a == b)`.

[38] https://docs.python.org/3/library/stdtypes.html#boolean-operations-and-or-not

Primerjalni operatorji

Primerjalni operatorji (vir[39]): * < manj kot, * <= manj ali enako, * > večje kot, * >= večje ali enako, * == enako, * != neenako, * `is` istost objekta primerjanih operandov, * `is not` ali operanda nista isti objekt.

Primer:

```
In [10]: 5 < 6
```

```
Out[10]: True
```

Lahko naredimo niz logičnih operatorjev:

```
In [11]: 1 < 3 < 6 < 7
```

```
Out[11]: True
```

preverimo istost (ali gre za isti objekt):

```
In [12]: a = 1
         a is 1
```

```
Out[12]: True
```

Uporabimo še logični operator:

```
In [13]: 1 < 3 and 6 < 7
```

```
Out[13]: True
```

```
In [14]: 1 < 3 and not 6 < 7
```

```
Out[14]: False
```

Tukaj velja izpostaviti, da je v logičnih izrazih poleg `False` neresnična tudi vrednost `0`, `None`, prazen seznam (ali niz, terka, ...); **vse ostalo je** `True`.

Poglejmo nekaj primerov:

```
In [15]: 1 and True
```

```
Out[15]: True
```

```
In [16]: 0 and True
```

```
Out[16]: 0
```

```
In [17]: (1,) and True
```

```
Out[17]: True
```

[39] https://docs.python.org/3/library/stdtypes.html#comparisons

Podatkovni tipi za numerične vrednosti

Pogledali si bomo najbolj pogoste podatkovne tipe, ki jih bomo pozneje uporabili (vir[40]):

- `int` za zapis poljubno velikega celega števila,
- `float` za zapis števil s plavajočo vejico,
- `complex` za zapis števil v kompleksni obliki.

Bolj podrobno bomo podatkovne tipe (natančnost zapisa in podobno) spoznali pri uvodu v numerične metode.

Primeri:

```
In [18]: celo_število = -2**3000          # `**` predstavlja potenčni operator
         racionalno_število = 3.141592
         kompleksno_število = 1 + 3j      # `3j` predstavlja imaginarni del
```

Operatorji numeričnih vrednosti

Podatkovni tipi razvrščeni po naraščajoči prioriteti (vir[41]):

1. `x + y` vsota,

- `x - y` razlika,
- `x * y` produkt,
- `x / y` deljenje,
- `x // y` celoštevilsko deljenje (rezultat je celo število zaokroženo navzdol),
- `x % y` ostanek pri celoštevilskem deljenju,
- `-x` negiranje,
- `+x` nespremenjen,
- `abs(x)` absolutna vrednost,
- `int(x)` celo število tipa `int`,
- `float(x)` racionalno število tipa `float`,
- `complex(re, im)` kompleksno število tipa `complex`,
- `c.conjugate()` kompleksno konjugirano število,
- `divmod(x, y)` vrne par `(x // y, x % y)`,
- `pow(x, y)` vrne `x` na potenco `y`,
- `x ** y` vrne `x` na potenco `y`.

Pri aritmetičnih operatorjih velja izpostaviti različico, pri kateri se rezultat priredi operandu (vir[42]). To imenujemo *razširjena dodelitev* (angl. *augmented assignement).

Poglejmo si primer. Namesto:

[40] https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex
[41] https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex
[42] https://docs.python.org/3/reference/simple_stmts.html#augmented-assignment-statements

```
In [19]: a = 1
         a = a + 1 # prištejemo vrednost 1
         a

Out[19]: 2
```

lahko zapišemo:

```
In [20]: a = 1
         a += 1
         a

Out[20]: 2
```

Operatorji na nivoju bitov

Operatorje na nivoju bitov bomo v inženirski praksi redko uporabljali, vseeno jih navedimo po naraščajoči prioriteti (vir[43]):

- `x | y` *ali* na nivoju bitov `x` in `y`,
- `x ^ y` *ekskluzivni ali* (samo en, ne oba) na nivoju bitov `x` in `y`,
- `x & y` bitni *in* na nivoju bitov `x` in `y`,

- `x << n` premik bitov `x` za `n` bitov levo,
- `x >> n` premik bitov `x` za `n` bitov desno,
- `~x` negiranje bitov `x`.

Poglejmo si primer (vir[44]):

```
In [21]: a = 21  # 21 v bitni obliki: 0001 0101
         b = 9   # 9 v bitni obliki:  0000 1001

         c = a & b; # 0000 0001 v bitni obliki predstavlja število 1 v decimalni
         c

Out[21]: 1
```

Prikažimo število v bitni obliki:

```
In [22]: bin(c)

Out[22]: '0b1'
```

1.6.2 Sestavljene podatkovne strukture

Niz (string)

Niz je sestavljen iz znakov po standardu *Unicode*. V podrobnosti se ne bomo spuščali; pomembno je navesti, da je Python od verzije 3 naprej naredil velik napredek (vir[45]) in da lahko poljubne znake po standardu *Unicode* uporabimo tako v tekstu, kakor tudi v programskih datotekah .py.

[43] https://docs.python.org/3/library/stdtypes.html#bitwise-operations-on-integer-types
[44] https://www.tutorialspoint.com/python/bitwise_operators_example.htm
[45] https://docs.python.org/3/howto/unicode.html

Niz se začne in zaključi z dvojnim "ali enojnim ' narekovajem.

```
In [23]: b = 'tekst' # b je niz (string)
         b

Out[23]: 'tekst'
```

Pri tem ni težav z imeni, ki smo jih bolj vajeni:

```
In [24]: π = 3.14
```

Znak π se v notebooku zapiše, da napišemo `\pi` (imena so ponavadi ista kot se uporabljajo pri zapisu v LaTeXu) in nato pritisnemo [tab].

Nekatere operacije nad nizi:

```
In [25]: print('kratek ' + b) # seštevanje
         print(3 * 'To je množenje niza. ')      # množenje niza
         print('Celo število:', int('5') + int('5'))      # pretvorba niza '5' v celo število
         print('Število s plavajočo vejico:', float('5') + float('5'))  # pretvorba niza '5' v število

kratek tekst
To je množenje niza. To je množenje niza. To je množenje niza.
Celo število: 10
Število s plavajočo vejico: 10.0
```

Terka (tuple)

Tuples (vir[46]) ali po slovensko terke so seznami poljubnih **objektov**, ki jih ločimo z vejico in pišemo znotraj okroglih oklepajev:

```
In [26]: terka_1 = (1, 'programiranje', 5.0)

In [27]: terka_1

Out[27]: (1, 'programiranje', 5.0)
```

Večkrat smo že uporabili besedo *objekt*. Kaj je objekt? Objekte in razrede si bomo podrobneje pogledali pozneje; zaenkrat se zadovoljimo s tem, da je objekt več kot samo *ime* za določeno vrednost; objekt ima tudi metode. Npr. drugi element je tipa `str`, ki ima tudi metodo `replace` za zamenjavo črk:

```
In [28]: terka_1[1].replace('r', '8')

Out[28]: 'p8og8ami8anje'
```

Če ima terka samo en objekt, potem jo moramo zaključiti z vejico (sicer se ne loči od objekta v oklepaju):

```
In [29]: terka_2 = (1,)
         type(terka_2)
```

[46] https://docs.python.org/tutorial/datastructures.html#tuples-and-sequences

```
Out[29]: tuple
```

Terke lahko zapišemo tudi brez okroglih oklepajev:

```
In [30]: terka_3 = 1, 'brez oklepajev', 5.

In [31]: terka_3

Out[31]: (1, 'brez oklepajev', 5.0)
```

Do objektov v terki dostopamo prek indeksa (**se začnejo z 0**):

```
In [32]: terka_1[0]

Out[32]: 1
```

Terke **ni mogoče spreminjati** (angl: *immutable*), elementov ne moremo odstranjevati ali dodajati. Klicanje spodnjega izraza bi vodilo v napako:

```
terka_1[0] = 3
>>> TypeError: 'tuple' object does not support item assignment
```

Seznam (list)

Seznami (vir[47]) se zapišejo podobno kot terke, vendar se uporabijo oglati oklepaji:

```
In [33]: seznam = [1., 2, 'd']

In [34]: seznam

Out[34]: [1.0, 2, 'd']
```

Za razliko od terk, se sezname lahko spreminja:

```
In [35]: seznam[0] = 10
         seznam

Out[35]: [10, 2, 'd']
```

Seznami so spremenljivi (angl. *mutable*) in treba se je zavedati, da **ime** v bistvu **kaže** na mesto v pomnilniku računalnika:

```
In [36]: b = seznam
         b

Out[36]: [10, 2, 'd']
```

Sedaj tudi `b` kaže na isto mesto v pomnilniku.

Poglejmo, kaj se zgodi s `seznam`, če spremenimo element seznama `b`:

[47] https://docs.python.org/tutorial/datastructures.html#more-on-lists

```
In [37]: b[0] = 9
         b

Out[37]: [9, 2, 'd']

In [38]: seznam

Out[38]: [9, 2, 'd']
```

Če želimo narediti kopijo podatkov, potem moramo narediti tako:

```
In [39]: b = seznam.copy()
         b[0] = 99
         b

Out[39]: [99, 2, 'd']

In [40]: seznam

Out[40]: [9, 2, 'd']
```

Primer seznama seznamov:

```
In [41]: matrika = [[1, 2, 3],
                    [4, 5, 6],
                    [7, 8, 9]]

In [42]: matrika

Out[42]: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Več o matrikah bomo izvedeli pozneje.

Izbrane operacije nad seznami Tukaj si bomo pogledali nekatere najbolj pogoste operacije, za več, glejte dokumentacijo[48].

Najprej si poglejmo dodajanje elementa:

```
In [43]: seznam = [0, 1, 'test']
         seznam.append('dodajanje elementa')
         seznam

Out[43]: [0, 1, 'test', 'dodajanje elementa']
```

Potem vstavljanje na določeno mesto (z indeksom 1, ostali elementi se zamaknejo):

```
In [44]: seznam.insert(1,'na drugo mesto')
         seznam
```

[48] https://docs.python.org/tutorial/datastructures.html

```
Out[44]: [0, 'na drugo mesto', 1, 'test', 'dodajanje elementa']
```

Potem lahko določeni element odstranimo. To naredimo s `seznam.pop(i)` ali `del seznam[2]`. Prvi način vrne odstranjeni element, drugi samo odstrani element:

```
In [45]: odstranjeni_element = seznam.pop(2) # odstani element z indeksom 2 in ga vrne
         # podoben učinek bi imel ukaz: del seznam[2]
         print('seznam:', seznam)
         print('odstranjeni element:', odstranjeni_element)
```

```
seznam: [0, 'na drugo mesto', 'test', 'dodajanje elementa']
odstranjeni element: 1
```

Z metodo `.index(v)` najdemo prvi element vrednosti `v`:

```
In [46]: seznam.index('test')
```

```
Out[46]: 2
```

Zakaj uporabiti terko in ne seznama? Terk ne moremo spreminjati, zato je njihova numerična implementacija bistveno bolj lahka (v numeričnem smislu) in spomin se lahko bolje izrabi. Posledično so terke **hitrejše** od seznamov; poleg tega jih ne moretmo po nerodnosti **spremeniti**. Za širši odgovor glejte vir[49].

Množice (Sets)

Množice so v principu podobne *množicam* kot jih poznamo iz matematike. Množice v Pythonu (dokumentacija[50]) za razliko od ostalih sestavljenih struktur:

- **ne dovoljujejo podvojenih elementov** in
- **nimajo urejenega vrstnega reda** (pomembno, npr. za pravilno uporabo v zankah).

Vredno je poudariti, da sta dodajanje in ugotavljanje pripadnosti elementa množici zelo hitri operaciji.

Kreiramo jih z zavitimi oklepaji `{}` ali ukazom `set()`:

```
In [47]: A = {'a', 4, 'a', 2, 'ž', 3, 3}
         A
```

```
Out[47]: {2, 3, 4, 'a', 'ž'}
```

Če se želimo v seznamu znebiti podvojenih elementov, je zelo enostaven način, da kreiramo množico iz seznama:

```
In [48]: B = set([6, 4, 6, 6])
         B
```

```
Out[48]: {4, 6}
```

[49] http://getpython3.com/diveintopython3/native-datatypes.html#tuples
[50] https://docs.python.org/tutorial/datastructures.html#sets

Uporabimo lahko tipične matematične operacije nad množicami:

```
In [49]: B - A   # elementi v B brez elementov, ki so tudi v A

Out[49]: {6}

In [50]: A | B   # elementi v A ali B

Out[50]: {2, 3, 4, 6, 'a', 'ž'}

In [51]: A & B   # elementi v A in B

Out[51]: {4}

In [52]: A ^ B   # elementi v A ali B, vendar ne v obeh

Out[52]: {2, 3, 6, 'a', 'ž'}
```

Slovar (dictionary)

Slovar (dokumentacija[51]) lahko definiramo eksplicitno s pari *ključ : vrednost*, ločenimi z vejico znotraj zavitih oklepajev:

```
In [53]: parametri = {'višina': 5.2, 'širina': 43, 'g': 9.81}
         parametri

Out[53]: {'g': 9.81, 'višina': 5.2, 'širina': 43}
```

Do vrednosti (angl. *value*) elementa dostopamo tako, da uporabimo ključ (angl. *key*):

```
In [54]: parametri['višina']

Out[54]: 5.2
```

Pogosteje uporabljamo metode: * `values()`, ki vrne vrednosti slovarja, * `keys()`, ki vrne ključe slovarja, * `items()`, ki seznam terk (ključ, vrednost); to bomo pogosto rabili pri zankah.

Primer:

```
In [55]: parametri.items()

Out[55]: dict_items([('višina', 5.2), ('širina', 43), ('g', 9.81)])
```

Slovarje lahko spreminjamo; npr. dodamo element:

```
In [56]: parametri['lega_težišča'] = 99999
         parametri

Out[56]: {'g': 9.81, 'lega_težišča': 99999, 'višina': 5.2, 'širina': 43}
```

[51] https://docs.python.org/tutorial/datastructures.html#dictionaries

Elemente odstranimo (podobno kot zgoraj `pop` odstrani in vrne vrednost `del` samo odstrani):

```
In [57]: parametri.pop('lega_težišča')

Out[57]: 99999

In [58]: parametri

Out[58]: {'g': 9.81, 'višina': 5.2, 'širina': 43}
```

Operacije nad sestavljenimi podatkovnimi strukturami

Večina sestavljenih podatkovnih struktur dovoljuje sledeče operacije (dokumentacija[52]):

- `x in s` vrne `True`, če je kateri element iz `s` enak `x`, sicer vrne `False`,
- `x not in s` vrne `False`, če je kateri element iz `s` enak `x`, sicer vrne `True`,
- `s + t` sestavi `s` in `t`,
- `s * n` ali `n * s` rezultira v `n`-krat ponovljen `s`,
- `s[i]` vrne `i`-ti element iz `s`,
- `s[i:j]` vrne `s` od elementa `i` (vključno) do `j` (ni vključen),
- `s[i:j:k]` rezanje `s` od elementa `i` do `j` po koraku `k`,
- `len(s)` vrne dolžino `s`,
- `min(s)` vrne najmanjši element iz `s`,

- `max(s)` vrne največji element iz `s`,
- `s.index(x[, i[, j]])` vrne indeks prve pojave `x` v `s` (od indeksa `i` do `j`),
- `s.count(x)` vrne število elementov.

Nekateri primeri:

```
In [59]: 'a' in 'abc' # in primerja ali je objekt v nizu, seznamu,...

Out[59]: True
```

Zgoraj smo imeli seznam `parametri`:

```
{'g': 9.81, 'višina': 5.2, 'širina': 43}
```

Preverimo, ali vsebuje element s ključem `širina`:

```
In [60]: 'širina' in parametri

Out[60]: True
```

Štetje in rezanje sestavljenih podatkovnih struktur

Predpostavimo spodnji niz:

```
In [61]: abeceda = 'abcčdefghijklmnoprsštuvzž'
```

[52] https://docs.python.org/library/stdtypes.html#common-sequence-operations

V Pythonu začnemo šteti indekse z 0! Na spletu lahko najdete razprave o tem ali je prav, da se indeksi začnejo z 0 ali 1. Zakaj začeti z 0 zelo lepo pojasni prof. dr. Janez Demšar v knjigi Python za programerje (vir[53], stran 28) ali pa tudi prof. dr. Edsger W. Dijkstra v zapisu na spletu[54].

Rezanje imenujemo operacijo, ko iz seznama izrežemo določene člene.

Splošno pravilo je:

```
s[i:j:k]
```

kar pomeni, da se seznam `s` razreže od elementa `i` do `j` po koraku `k`. Če katerega elementa ne podamo potem se uporabi (logična) vrednost:

- `i` je privzeto 0, kar pomeni prvi element,
- `j` je privzeto -1, kar pomeni zadnji element,
- `k` je privzeto 1.

Prvi znak je torej:

```
In [62]: abeceda[0]
```

```
Out[62]: 'a'
```

Prvi trije znaki so:

```
In [63]: abeceda[:3]
```

```
Out[63]: 'abc'
```

Od prvih 15 znakov, vsak tretji je:

```
In [64]: abeceda[:15:3]
```

```
Out[64]: 'ačfil'
```

Zadnjih petnajst znakov, vsak tretji:

```
In [65]: abeceda[-15::3]
```

```
Out[65]: 'jmpšv'
```

Ko želimo obrniti vrstni red, lahko to naredimo tako (beremo od začetka do konca po koraku -1, kar pomeni od konca proti začetku s korakom 1:)):

```
In [66]: abeceda[::-1]
```

```
Out[66]: 'žzvutšsrponmlkjihgfedčcba'
```

[53] https://ucilnica.fri.uni-lj.si/file.php/166/Python%20za%20programerje.pdf
[54] http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html

1.6.3 Kontrola toka programa

Pogledali si bomo nekatera osnovna orodja za kontrolo toka izvajanja programa (dokumentacija[55]).

Stavek `if`

Tipična oblika stavka `if` (dokumentacija[56]):

```
if pogoj0:
    print('Izpolnjen je bil pogoj0')
elif pogoj1:
    print('Izpolnjen je bil pogoj1')
elif pogoj2:
    print('Izpolnjen je bil pogoj2')
else:
    print('Noben pogoj ni bil izpolnjen')
```

Primer:

```
In [67]: a = 6
         if a > 5:
             print('a je več od 5') # se izvede, če je pogoj res.
             print(13*'-')

a je več od 5
-------------
```

Zgoraj smo funkcijo `print` zamaknili, saj pripada bloku kode, ki se izvede v primeru `a > 0`. Nekateri programski jeziki kodo znotraj bloka dajo v zavite ali kakšne druge oklepaje in dodatno še zamikajo. Pri Pythonu se samo zamikajo. Zamik je tipično **4 presledke**. Lahko je tudi **tabulator** (manj pogosto), vendar pa smemo v eni datoteki .py uporabljati le en način zamikanja.

Stavke lahko zapišemo tudi v eno vrstico, vendar je taka uporaba odsvetovana, saj zmanjšuje preglednost kode:

```
In [68]: if a > 5: print('taka oblika je odsvetovana'); print(40*'-'); print('uporaba podpičja namesto n

taka oblika je odsvetovana
----------------------------------------
uporaba podpičja namesto nove vrstice
```

Izraz `if`

Poleg stavka `if` ima Python tudi izraz `if` (angl. *ternary operator*, glejte dokumentacijo[57]).

Sintaksa je:

```
[izvedi, če je True] if pogoj else [izvedi, če je False]
```

[55] https://docs.python.org/tutorial/controlflow.html
[56] https://docs.python.org/tutorial/controlflow.html#if-statements
[57] https://docs.python.org/faq/programming.html#is-there-an-equivalent-of-c-s-ternary-operator

Preprost primer:

```
In [69]: 'Je res' if 5==3 else 'Ni res'

Out[69]: 'Ni res'
```

If izrazi so zelo uporabni in jih bomo pogosto uporabljali pri izpeljevanju seznamov!

Zanka `while`

Sintaksa zanke while (dokumentacija[58]) je:

```
while pogoj:
    [koda za izvajanje]
else:
    [koda za izvajanje]
```

Pri tem je treba poudariti, da je del `else` opcijski in se izvede ob izhodu iz zanke `while`. Ukaz `break` prekine zanko in ne izvede `else` dela. Ukaz `continue` prekine izvajanje trenutne kode in gre takoj na testiranje pogoja.

Preprost primer:

```
In [70]: a = 0
         while a < 3:
             print(a)
             a += 1
         else:
             print('--')

0
1
2
--
```

Zanka `for`

V Pythonu bomo **zelo pogosto** uporabljali zanko `for` (dokumentacija[59]). Sintaksa je:

```
for element in seznam_z_elementi:
    [koda za izvajanje]
else:
    [koda za izvajanje]
```

Podobno kakor pri `while` je tudi tukaj `else` del opcijski.

Poglejmo primer:

[58] https://docs.python.org/reference/compound_stmts.html#the-while-statement
[59] https://docs.python.org/3/tutorial/controlflow.html#for-statements

```
In [71]: imena = ['Jaka', 'Miki', 'Luka', 'Anja']
         for ime in imena: # tukaj se skriva prirejanje; ``ime`` vsakič pridobi novo vrednost
             print(ime)
```

```
Jaka
Miki
Luka
Anja
```

Zelo uporabno je zanke delati čez slovarje:

```
In [72]: print('Slovar je:', parametri)
         for key, val in parametri.items():
             print(key, '=', val)
```

```
Slovar je: {'višina': 5.2, 'širina': 43, 'g': 9.81}
višina = 5.2
širina = 43
g = 9.81
```

Funkcija `zip`

Ko želimo kombinirati več seznamov v zanki, si pomagamo s funkcijo `zip` (dokumentacija[60]); primer:

```
In [73]: x = [1, 2, 3]
         y = [10, 20, 30]

         for xi, yi in zip(x, y):
             print('Pari so: ', xi, 'in', yi)
```

```
Pari so:  1 in 10
Pari so:  2 in 20
Pari so:  3 in 30
```

Funkcija `range`

Pogosto bomo uporabljali zanko `for` v povezavi s funkcijo `range` (dokumentacija[61]). Sintaksa je:

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

Če torej funkcijo kličemo z enim parametrom, je to `stop`: **do** katere številke naštevamo. V primeru klica z dvema parametroma `start, stop`, naštevamo od, do! Tretji parameter je opcijski in definira korak `step`.

Poglejmo primer:

[60] https://docs.python.org/library/functions.html#zip
[61] https://docs.python.org/tutorial/controlflow.html#the-range-function

```
In [74]: for i in range(3):
             print(i)

0
1
2
```

Funkcija `enumerate`

Zanko `for` pogosto uporabljamo tudi s funkcijo `enumerate`, ki elemente oštevilči (dokumentacija[62]).

Poglejmo primer:

```
In [75]: for i, ime in enumerate(imena):
             print(i, ime)

0 Jaka
1 Miki
2 Luka
3 Anja
```

1.7 Za konec: plonk list :)

Veliko jih je; tukaj je povezava[63] do enega.

1.8 Nekaj vprašanj za razmislek!

1. Namestite *Anaconda*.
2. Namestite *pyCharm Community Edition* in GitHub ter prenesite predavanja.
3. V poljubnem delovnem direktoriju zaženite Jupyter notebook.
4. Prikažite uporabo stilov, uporabo poudarjenega, poševnega teksta, uporabo seznamov, enačbe ...
5. Definirajte razliko med statičnim in dinamičnim tipiziranjem.
6. Poiščite pomoč poljubnega ukaza (znotraj Pythona in na uradni domači strani).
7. Prikažite uporabo *niza*, *celega števila* in *števila z uporabo plavajoče vejice*.
8. Prikažite uporabo *terke* in njenih bistvenih lastnosti.
9. Prikažite uporabo *seznama* in njegovih bistvenih lastnosti.
10. Komentirajte tipične operacije nad seznami.
11. Komentirajte uporabo *množic* in tipične uporabe.
12. Prikažite uporabo slovarjev.
13. Katere aritmetične operatorje poznamo v Pythonu? Prikažite uporabo.
14. Katere primerjalne operatorje poznamo v Pythonu? Prikažite uporabo.
15. Katere logične operatorje poznamo v Pythonu? Prikažite uporabo.
16. Prikažite uporabo stavka ‘`if`.
17. Kakšna je razlika med stavkom `if` in izrazom `if`. Prikažite!
18. Prikažite uporabo zanke `while`.
19. Prikažite uporabo zanke `for`.

[62] https://docs.python.org/library/functions.html#enumerate
[63] http://perso.limsi.fr/pointal/_media/python:cours:mementopython3-english.pdf

20. Prikažite uporabo zanke `for` v povezavi s funkcijami `range`, `enumerate`, `zip`

Še nekaj branja: automatetheboringstuff.com[64].

In en twit:

```
In [76]: %%html
         <blockquote class="twitter-tweet" data-lang="en"><p lang="en" dir="ltr">The most important sli
         <script async src="//platform.twitter.com/widgets.js" charset="utf-8"></script>
```

```
<IPython.core.display.HTML object>
```

1.8.1 Vključevanje lokalnega video posnetka s transformacijami

```
In [77]: from IPython.core.display import HTML
```

```
In [78]: HTML('<style>video{\
           -moz-transform:scale(0.75) rotate(-90deg);\
           -webkit-transform:scale(0.75) rotate(-90deg);\
           -o-transform:scale(0.75) rotate(-90deg);\
           -ms-transform:scale(0.75) rotate(-90deg);\
           transform:scale(0.75) rotate(-90deg);\
         }</style>')
```

```
Out[78]: <IPython.core.display.HTML object>
```

(Ne prikaže na github-u)

[64] https://automatetheboringstuff.com/

Poglavje 2

Print, delo z datotekami, funkcije, moduli

2.1 Funkcija print

`print` predstavlja eno od najbolj pogosto uporabljenih funkcij. Vse podane argumente izpiše, mednje da presledek, na koncu pa gre v novo vrstico.

Poglejmo primer:

```
In [1]: print('prvi argument', 'drugi', '=', 5.)

prvi argument drugi = 5.0
```

Za bolj splošno uporabo glejmo dokumentacijo[1]:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False),
```

kjer so argumenti:

- `*objects` predstavlja argumente, ki jih želimo izpisati (celovitost zvezdice bo jasna pri obravnavi *funkcij*, spodaj,
- `sep` predstavlja delilni niz (angl. *sep*arator),
- `end` predstavlja zaključni niz,
- `file` predstavlja *datoteko* izpisa (`sys.stdout` predstavlja *standard output*, v konkretnem primeru to pomeni *zaslon* oz ukazna vrstica; pozneje bomo to spremenili),
- `flush` v primeru `True` izpis zaključi (sicer lahko zaradi optimizacije ostane v medpolnilniku, angl. *buffer*).

Pri argumentu `end` smo zgoraj uporabili t. i. **izhodni znak** (angl. *escape character*): `\`.

Namen izhodnega znaka je, da sledečemu znaku da poseben pomen, nekatere pogoste uporabe so: * **\n** vstavi *prelomi vrstico*, * **\t** vstavi *tabulator*, * **\'** izpiše enojni narekovaj ', * **\"** izpiše dvojni narekovaj ", * **** prikaže *izhodni znak* (angl. *backslash*), * \ nova vrstica v večvrstičnem nizu.

Primer (kako ponavadi ne programiramo):

[1] https://docs.python.org/library/functions.html#print

```
In [2]: print('pet','et', sep='-', end='=')
        print('p:)')

pet-et=p:)
```

In še primer z izhodnimi znaki:

```
In [3]: print('Nizi v Pythonu (po PEP8) naj ne bi bili daljši od 80 znakov. \
        Razlog je v tem, da nam ni treba pomikati okna levo in desno. \
        Da pa vseeno lahko napišemo nize, ki so daljši od 80 znakov \
        Uporabimo enojni delilni znak \\ (bodite pozorni na to, \
        kako se izpiše: \\)',)

Nizi v Pythonu (po PEP8) naj ne bi bili daljši od 80 znakov. Razlog je v tem, da nam ni treba pomikati
```

2.2 Oblikovanje nizov

Pri oblikovanju/formatiranju nizov imamo na voljo več orodij:

- **f-niz** (angl. *f-string*, *f* zaradi "formatirani"), dokumentacija[2],
- **metoda** `.format()`, dokumentacija[3],
- % oblikovanje, dokumentacija[4].

Poglejmo si primere:

```
In [4]: a = 3.14
        print(f'pi = {a}')                # f-niz
        print('pi = {a}'.format(a = a))   # .format
        print('pi = %(a)3.2f' % {'a': a}) # %

pi = 3.14
pi = 3.14
pi = 3.14
```

f-niz je implementiran od Python verzije 3.6 naprej (PEP 498[5]) in predstavlja najbolj enostaven način oblikovanja. Ostala načina navajamo zgolj zato, da se bralec seznani z njimi in da naredimo povezavo na dokumentacijo.

f-niz se vedno začne s **f** pred prvim narekovajem in omogoča zelo splošno oblikovanje (glejte dokumentacijo[6]).

Vrednost, ki jo želimo znotraj niza oblikovati, damo v zavite oklepaje:

```
f'besedilo... {ime_arg1:fmt1} ... besedilo...'
```

[2] https://docs.python.org/reference/lexical_analysis.html#f-strings
[3] https://docs.python.org/library/stdtypes.html#str.format
[4] https://docs.python.org/library/stdtypes.html#string-formatting-operations
[5] https://www.python.org/dev/peps/pep-0498/
[6] https://docs.python.org/reference/lexical_analysis.html#f-strings

V zavitih oklepajih je: * pred : podamo ime vrednosti, ki jo želimo oblikovati * za : oblikujemo izpis spremenljivke; najpogosteje bomo uporabili *fmt* oblike: * w.d**f** - predstavitev s plavajočo vejico (**f**loat) * w.d**e** - predstavitev z eksponentom (**e**xponent) * w.d**g** - splošni format (**g**eneral) * w**s** - niz znakv (**s**tring).

`w` predstavlja (minimalno) skupno širino, `d` pa število mest za decimalno piko.

Poglejmo si primer:

```
In [5]: višina = 1.84
        ime = 'Marko'
        tekst = f'{ime} je visok {višina} m ali tudi: {višina:e} m, {višina:7.3f} m.'
        tekst

Out[5]: 'Marko je visok 1.84 m ali tudi: 1.840000e+00 m,   1.840 m.'
```

Podobno bi lahko naredili z metodo *format* in se sklicali na indeks argumenta:

```
In [6]: tekst = '{1} je visok {0} m ali tudi: {0:e} m, {0:7.3f} m.'.format(višina, ime)
        tekst

Out[6]: 'Marko je visok 1.84 m ali tudi: 1.840000e+00 m,   1.840 m.'
```

Primer z uporabo slovarja:

```
In [7]: parametri = {'visina': 5., 'gostota':100.111, 'ime uporabnika': 'Janko'}

In [8]: f'višina = {parametri["visina"]:7.3f}, gostota = {parametri["gostota"]:7.3e}'

Out[8]: 'višina =   5.000, gostota = 1.001e+02'
```

Oblikovanje s *fmt* (angl. *Format Specification Mini-Language*) je izredno bogato. Del za dvopičjem je v splošnem (glejte dokumentacijo[7]):

```
[[fill]align][sign][#][0][width][grouping_option][.precision][type]
```

kjer so v oglatih oklepajih opcijski parametri (vsi so opcijski):

- `fill` je lahko katerikoli znak,
- `align` označuje poravnavo, možnosti: `"<" | ">" | "=" | "^"`,
- `sign` označuje predznak, možnosti: ::= `"+" | -" | " "`,
- `width` definira minimalno skupno širino,
- `grouping_option` možnosti združevanja, možnosti: `"_" | ","`
- `precision` definira število števk po decimalnem ločilu,
- `type` definira, kako naj bodo podatki prikazani, možnosti: `"b" | č" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | o" | š" | "x" | "X" | "%"`

Besedili poravnavamo levo z `"<"`, desno z `">"` in sredinsko `"^"`. Primer sredinske/desne poravnave (širina 20 znakov):

```
In [9]: f'{višina:-^20}' # pred ^ je znak s katerim zapolnimo levo in desno stran
```

[7] https://docs.python.org/library/string.html#formatspec

```
Out[9]: '--------1.84--------'

In [10]: f'{parametri["visina"]:_>20}'

Out[10]: '_________________5.0'

In [11]: c = 4-2j
         f'Kompleksno število {c} je sestavljeno iz realnega ({c.real})\
         in imaginarnega ({c.imag}) dela.'

Out[11]: 'Kompleksno število (4-2j) je sestavljeno iz realnega (4.0)in imaginarnega (-2.0) dela.'
```

Oblikovanje datuma, ure si lahko pogledate v dodatku spodaj.

2.3 Funkcije

V Pythonu je veliko funkcij že vgrajenih (glejte dokumentacijo[8]); funkcije pa lahko tudi napišemo sami ali jih uvozimo iz t. i. modulov (zgoraj smo uvozili `datetime`, več si bomo pogledali pozneje).

Generični primer funkcije je (dokumentacija[9]):

```
def ime_funckije(parametri):
    '''docstring'''
    [koda]
    return vsebina
```

Pri tem izpostavimo: * `def` označuje definicijo funkcije, * `ime_funkcije` **imena funkcije po PEP8 pišemo z malo, večbesedna povežemo s podčrtajem**, * `parametri` parametri funkcije, več bomo povedali spodaj, * `docstring` (opcijsko) dokumentacija funkcije, * `[koda]` koda funkcije, * `return` (opcijsko) ukaz za izhod iz funkcije, kar sledi se vrne kot rezultat (če ni `return` ali po return ni ničesar, se vrne `None`)

Primer preproste funkcije:

```
In [12]: def površina(dolžina, širina):
             S = dolžina * širina
             return S

In [13]: površina(1, 20)

Out[13]: 20
```

2.3.1 Posredovanje argumentov v funkcije

Python pozna dva načina posredovanja argumentov: * glede na **mesto** (*positional*) * glede na **ime**.

Poglejmo si oba načina na primeru:

```
In [14]: površina(3, 6) # pozicijsko
```

[8] https://docs.python.org/library/functions.html
[9] https://docs.python.org/tutorial/controlflow.html#defining-functions

```
Out[14]: 18

In [15]: površina(širina = 3, dolžina = 6) # glede na ime

Out[15]: 18
```

Priporočeno je, da argumente posredujemo z *imenom*; s tem zmanjšamo možnost napake!

Če ne podamo imena, potem se privzame ime glede na vrstni red argumentov; primer:

```
In [16]: površina(3, širina=4)

Out[16]: 12
```

V kolikor mešamo *pozicijsko* in *poimensko* posredovanje argumentov, morajo **najprej biti navedeni pozicijski argumenti, šele nato poimenski**. Pri tem poimenski ne sme ponovno definirati pozicijskega; ker je `višina` prvi argument, bi klicanje take kode:

```
površina(3, višina=4)
```

povzročilo napako.

Primer dobre prakse pri definiranju argumentov

Pogosto funkcije dopolnjujemo in dodajamo nove argumente ali pa število argumentov funkciji sploh ne moremo vnaprej definirati!

Python s tem nima težav. Argumente, ki niso eksplicitno definirani obvladamo z:

- `*[ime0]` terka `[ime0]` vsebuje vse **pozicijske argumente**, ki niso eksplicitno definirani,
- `**[ime1]` slovar `[ime1]` vsebuje vse **poimenske argumente**, ki niso eksplicitno definirani.

Poglejmo primer:

```
In [17]: def neka_funkcija(student, izpit = True):
             ''' Prva verzija funkcije'''
             print(f'Študent {student} se je prijavil na izpit: {izpit}.')

In [18]: neka_funkcija('Janez', izpit=True)

Študent Janez se je prijavil na izpit: True.
```

Sedaj pa pripravimo drugo, nadgrajeno, verzijo (omogoča enako uporabo kot prva verzija):

```
In [19]: def neka_funkcija(*terka, **slovar):
             ''' Druga verzija funkcije '''
             student = terka[0]
             izpit = slovar.get('izpit', True) # True je privzeta vrednost
             izpisi_sliko = slovar.get('izpisi_sliko', False) # False je privzeta vrednost

             print(f'Študent {student} se je prijavil na izpit: {izpit}.')
             if izpisi_sliko:
                 print('slika:)')
```

```
In [20]: neka_funkcija('Janez', izpit=True)

Študent Janez se je prijavil na izpit: True.


In [21]: neka_funkcija('Janez', izpit=True, izpisi_sliko=True, neobstoječi_argument='ni')

Študent Janez se je prijavil na izpit: True.
slika:)
```

Posredovanje funkcij kot argument

Tukaj bi želeli izpostaviti, da so argumenti lahko tudi druge funkcije.

Poglejmo si primer:

```
In [22]: def oblika_a(vrednost):
             return f'Prikaz vrednosti: {vrednost}'

         def oblika_b(vrednost):
             return f'Prikaz vrednosti: {vrednost:3.2f}'

         def izpis(oblika, vrednost):
             print(oblika(vrednost))
```

Imamo torej dve funkciji `oblika_a` in `oblika_b`, ki malenkost drugače oblikujeta numerično vrednost.

Poglejmo uporabo:

```
In [23]: izpis(oblika_a, 3)

Prikaz vrednosti: 3


In [24]: izpis(oblika_b, 3)

Prikaz vrednosti: 3.00
```

2.3.2 Docstring funkcije

docstring je neobvezen del definicije funkcije, ki dokumentira funkcijo in njeno uporabo. Iz tega stališča je zelo priporočeno, da ga uporabimo.

Dokumentacija[10] za docstring navaja bistvene elemente:

- prva vrstica naj bo kratek povzetek funkcije,
- če je vrstic več, naj bo druga prazna (da se vizualno loči prvo vrstico od preostalega teksta),
- uporabimo tri narekovaje (dvojne "ali enojne '), ki označujejo večvrstični niz črk.

[10] https://docs.python.org/tutorial/controlflow.html#documentation-strings

Primer dokumentirane funkcije:

```
In [25]: def površina(dolžina = 1, širina = 10):
             """ Izračun površine pravokotnika

             Funkcija vrne vrednost površine.

             Argumenti:
             dolžina: dolžina pravokotnika
             širina: dolžina pravokotnika
             """
             return dolžina * širina
```

Primer klica s privzetimi argumenti (med klicem funkcije lahko s pritiskom [shift]+[tab] dostopamo do pomoči):

```
In [26]: površina()
```

```
Out[26]: 10
```

2.3.3 Lokalna/globalna imena

Preprosto vodilo pri funkcijah je: **funkcija vidi ven, drugi pa ne vidijo noter**. Funkcija ima notranja imena, ki se ne prekrivajo z istimi imeni v drugih funkcijah.

Poglejmo si primer:

```
In [27]: zunanja = 3
         def prva():
             notranja = 5
             print(f'Tole je zunanja vrednost: {zunanja}, tole pa notranja: {notranja}')
```

```
In [28]: prva()
```

```
Tole je zunanja vrednost: 3, tole pa notranja: 5
```

Ker ime `notranja` zunaj funkcije `prva` ni definirano, bi klicanje:

```
notranja
```

zunaj funkcije vrnilo napako (v funkcijo ne vidimo).

Tukaj velja omeniti, da je **posredovanje zunanjih imen v funkcijo mimo argumentov funkcije odsvetovano**.

2.3.4 `return`

Ukaz `return` vrne vrednosti iz funkcije. Doslej smo spoznali rezultate v obliki ene spremenljivke; ta spremenljivka pa je lahko tudi terka.

Poglejmo si primer:

```
In [29]: def vrnem_terko():
             return 1, 2, 3 # to je ekvivalenten zapis za terko: (1, 2, 3)

In [30]: vrnem_terko()

Out[30]: (1, 2, 3)
```

Rezultat funkcije lahko ujamemo tako:

```
In [31]: rezultat = vrnem_terko()
         rezultat

Out[31]: (1, 2, 3)
```

Ali tako, da vrednosti razpakiramo (to sicer velja v splošnem za terke):

```
In [32]: i1, i2, i3 = vrnem_terko()
         i1 # izpis samo ene vrednosti

Out[32]: 1
```

2.3.5 Anonimna funkcija/izraz

Anonimna funkcija (glejte dokumentacijo[11]) je funkcija, kateri ne damo imena (zato je *anonimna*), ampak jo definiramo z ukazom `lambda`. Tipična uporaba je:

```
lambda [seznam parametrov]: izraz
```

`lambda` funkcijo si bomo pogledali na primeru razvrščanja:

```
In [33]: seznam = [-4, 3, -8, 6]
```

od najmanjše do največje vrednosti. To lahko izvedemo z vgrajeno metodo `sort` (dokumentacija[12]):

```
sort(*, key=None, reverse=False)
```

Metoda ima dva opcijska argumenta:

- `key` ključ, po katerem razvrstimo elemente
- `reversed`, če je `False`, se med vrednostmi ključa uporabi < sicer >.

Opomba: `list.sort()` izvede razvrščanje na obstoječem seznamu in ne vrne seznama. Funkcija `sorted()` (glejte dokumentacijo[13]) pa vrne seznam urejenih vrednosti.

Poglejmo primer:

```
In [34]: seznam = [-4, 3, -8, 6]
         seznam.sort()
         seznam
```

[11] https://docs.python.org/tutorial/controlflow.html#lambda-expressions
[12] https://docs.python.org/3/library/stdtypes.html#list.sort
[13] https://docs.python.org/3/library/functions.html#sorted

```
Out[34]: [-8, -4, 3, 6]
```

Sedaj uporabimo ključ, kjer se bo za primerjavo vrednosti razvrščanja uporabila kvadratna vrednost:

```
In [35]: seznam.sort(key = lambda a: a**2)
         seznam

Out[35]: [3, -4, 6, -8]
```

2.4 Delo z datotekami

Delo z datotekami je pomembno, saj podatke pogosto shranjujemo v datoteke ali jih beremo iz njih. Datoteko, iz katere želimo brati ali vanjo pisati, moramo najprej odpreti. To izvedemo z ukazom `open` (dokumentacija[14]):

```
open(file, mode='r', buffering=-1, encoding=None,
    errors=None, newline=None, closefd=True, opener=None)
```

Od vseh argumentov je nujen samo `file`, ki definira pot do datoteke.

Argument `mode` je lahko:

- `'r'` za branje (*read*, privzeto),
- `'w'` za pisanje (*write*, datoteka se najprej pobriše),
- `'x'` za ekskluzivno pisanje; če datoteka že obstaja vrne napako,
- `'a'` za dodajanje na koncu obstoječe datoteke (*append*),
- `'b'` binarna oblika zapisa,
- `'t'` tekstnovna oblika zapisa (privzeto),
- `'+'` datoteka se odpre za branje in pisanje.

Zadnji trije zanki (`'b'`, `'t'`, `'+'`) se uporabljajo v kombinaciji s prvimi tremi (`'r'`, `'w'`, `'x'`). Primer: `mode='r+b'` pomeni branje in pisanje binarne datoteke.

Podrobneje bomo spoznali samo branje in pisanje tekstovnih datotek.

Ostali (opcijski) argumenti funkcije `open` so podrobno opisani v dokumentaciji[15].

Privzeto je `mode='r'` (oz. `'rt'`, ker je privzet tekstovni način), kar pomeni, da se datoteko odpre za branje v tekstovni obliki.

Poglejmo si najprej pisanje v (tekstovno) datoteko (`mode='w'`):

```
In [36]: datoteka = open('data/prikaz vpisa.txt', mode='w')
```

`'data/prikaz vpisa.txt'` predstavlja relativno pot (glede na mesto, kjer se poganja *Jupyter notebook*) do datoteke.

Zapišimo prvo vrstico s pomočjo metode `write`:

```
In [37]: datoteka.write('test prve vrstice\n')

Out[37]: 18
```

[14] https://docs.python.org/library/functions.html#open
[15] https://docs.python.org/library/functions.html#open

Metoda `write` vrne število zapisanih zankov. Če boste v tem trenutku pogledali vsebino datoteke, je velika možnost, da je še prazna. Razlog je v tem, da je vsebina še v medpolnilniku, ki se izprazni na disk samo občasno ali ob zaprtju datoteke (razlog za tako delovanje je v hitrosti zapisa na disk; za podrobnosti glejte dokumentacijo argumenta `buffering` funkcije `open`).

Zapišimo drugo vrstico:

```
In [38]: datoteka.write('druga vrstica\n', )

Out[38]: 14
```

Vpišimo sedaj še nekaj številčnih vrednosti:

```
In [39]: for d in range(5):
             datoteka.write(f'{d:7.2e}\t {d:9.4e}\n')
```

Datoteko zapremo:

```
In [40]: datoteka.close()
```

Sedaj datoteko odpremo za branje in pisanje? `mode='r+'`:

```
In [41]: datoteka = open('data/prikaz vpisa.txt', mode='r+')
```

Preverimo vse vrstice s `for` zanko; v spodnji kodi nam `datoteka` v vsakem klicu vrne eno vrstico:

```
In [42]: for line in datoteka:
             print(line, end='')

test prve vrstice
druga vrstica
0.00e+00         0.0000e+00
1.00e+00         1.0000e+00
2.00e+00         2.0000e+00
3.00e+00         3.0000e+00
4.00e+00         4.0000e+00
```

Pri uvodu v funkcijo `print` smo omenili argument `file`, ki definira datoteko, v katero se piše (privzet je standardni izhod oz. *zaslon*). Če kot argument `file` posredujemo datoteko, ki je odprta za pisanje:

```
In [43]: print('konec', file=datoteka)
         datoteka.close()
```

se vsebina zapiše v datoteko. Rezultat lahko preverite v datoteki!

Tukaj je lepa priložnost, da spoznamo stavek `with` (dokumentacija[16]). Celovitost stavka `with` presega namen tega predmeta, je pa zelo preprosta uporaba pri delu z datotekami, zato si poglejmo primer:

```
In [44]: with open('data/prikaz vpisa.txt') as datoteka:
             for line in datoteka:
                 print(line, end='')
```

[16] https://docs.python.org/reference/compound_stmts.html#with

```
test prve vrstice
druga vrstica
0.00e+00         0.0000e+00
1.00e+00         1.0000e+00
2.00e+00         2.0000e+00
3.00e+00         3.0000e+00
4.00e+00         4.0000e+00
konec
```

Stavek `with` rezultat funkcije `open` shrani v (*as*) ime `datoteka`, potem pa za vsako vrstico izpišemo vrednosti. Pri izhodu iz stavka `with` se samodejno pokliče metoda `datoteka.close()`. Stavek `with` nam torej omogoča pisanje pregledne in kompaktne kode.

2.5 Obravnavanje izjem

Pri programiranju se relativno pogosto pojavijo izjeme; na primer: deljenje z nič.

```
1/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-50-05c9758a9c21> in <module>()
----> 1 1/0

ZeroDivisionError: division by zero
```

V primeru izjeme Python opozori na to in prekine izvajanje. Ni dobro, da se program prekine ali zruši; iz tega razloga bi zgornjo situacijo programer začetnik verjetno reševal s stavkom `if`. Takšen pristop pa ni preveč uporaben niti splošen in zato je v večini modernih programskih jezikih uveljavljeno obravnavanje izjem.

Pogledali si bomo nekatere osnove, ki bodo študentom predvsem olajšale branje in razumevanje kode od drugih avtorjev.

Izjeme obvladujemo s stavkom `try`:

```
try:
    [koda0]
except:
    [koda1]
finally:
    [koda2]
```

V stavki `try` se `except` izvede, če se v kodi `[koda0]` pojavi izjema, `finally` je opcijski in se izvede v vsakem primeru ob izhodu iz stavka `try`.

Poglejmo primer, ko enostavno nadaljujemo z izvajanjem:

```
In [45]: try:
             1/0
         except:
             pass
```

Ukaz `pass` uporabimo, ko sintaksa zahteva stavek (kodo) za izvajanje, vendar ne želimo nobenega ukrepa (glejte dokumentacijo[17]).

S pomočjo `except` lahko lovimo tudi različne izjeme, to naredimo tako:

```
In [46]: try:
             1/0
         except ZeroDivisionError:
             print('Deljenje z ničlo!') # ali kakšna druga akcija
         except:
             print('Nepričakovana napaka.') # odsvetovano; dobro je predvideti napako!
```

```
Deljenje z ničlo!
```

Sedaj uporabimo isto kodo za primer deljenja števila z nizom:

```
In [47]: try:
             1/'to pa ne gre'
         except ZeroDivisionError:
             print('Deljenje z ničlo!') # ali kakšna druga akcija
         except:
             print('Nepričakovana napaka.') # odsvetovano; dobro je predvideti napako!
```

```
Nepričakovana napaka.
```

2.5.1 Proženje izjem

Izjeme pa lahko tudi sami prožimo; to naredimo z ukazom `raise` (dokumentacija[18]).

Primer proženja napake je:

```
raise Exception('Opis izjeme')
```

preprost, vendar celovit, primer, ko bi pričakovali, da je `ime_osebe` tipa `str`:

```
In [48]: ime_osebe = 1
         try:
             if type(ime_osebe) != str:
                 raise Exception(f'Ime ni pričakovanega tipa `str`.')
         except Exception as izjema:
             print(izjema)
```

```
Ime ni pričakovanega tipa `str`.
```

[17] https://docs.python.org/tutorial/controlflow.html#pass-statements
[18] https://docs.python.org/tutorial/errors.html#raising-exceptions

2.6 Izpeljevanje seznamov

Pri programiranju se pogosto srečujemo s tem, da moramo izvajati operacije na posameznem elementu seznama. Predhodno smo že spoznali zanko `for`, ki jo lahko uporabimo v takem primeru.

Oglejmo si primer, kjer izračunamo kvadrat števil:

```
In [49]: seznam = [1, 2, 3] # izvorni seznam
         rezultat = [] # pripravimo prazen seznam v katerega bomo dodajali rezultate
         for element in seznam:
             rezultat.append(element**2)
         rezultat

Out[49]: [1, 4, 9]
```

Mnogo enostavneje lahko isti rezultat dosežemo s t. i. **izpeljevanjem seznamov** (angl. *list comprehensions*, glejte dokumentacijo[19]).

Zgornji primer bi bil:

```
In [50]: seznam = [1, 2, 3] # izvorni seznam
         rezultat = [element**2 for element in seznam] # <<< na desni strani = je izpeljevanje seznamov
         rezultat

Out[50]: [1, 4, 9]
```

Izpeljevanje seznamov predstavlja koda `[element**2 for element in seznam]`, ki v bistvu pove to, kar je napisano: izračunaj kvadrat za vsak element v seznamu. Da je rezultat seznam, nakazujejo oglati oklepaji.

Izpeljevanje seznamov ima sledečo sintakso:

```
[koda for element in seznam]
```

in ima predvsem dve prednosti:

- **preglednost / kompaktnost** kode in
- malo **hitrejše** izvajanje.

Preglednost/kompaktnost smo lahko že presodili. Kako preverimo hitrost? Najlažje to naredimo s t. i. magičnim ukazom `%%timeit`, torej **meri čas** (glejte dokumentacijo[20]); spodaj bomo uporabili blokovni magični ukaz (označuje ga `%%`):

```
%%timeit -n1000
```

kjer parameter `-n1000` omejuje merjenje časa na 1000 ponovitev.

Najprej daljši način:

```
In [51]: seznam = range(1000)
```

[19] https://docs.python.org/tutorial/datastructures.html#list-comprehensions
[20] http://ipython.readthedocs.io/en/stable/interactive/magics.html#magic-timeit

```
In [52]: %%timeit -n1000
         rezultat = []
         for element in seznam:
             rezultat.append(element**2)
```

```
766 µs ± 52 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Nato izpeljevanje seznamov (bolj pregledno in malenkost hitreje):

```
In [53]: %%timeit -n1000
         rezultat = [element**2 for element in seznam]
```

```
887 µs ± 162 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Poglejmo si še dva uporabna primera:

- uporabo izraza `if` za izračun nove vrednosti,
- uporabo izraza `if`, če se nova vrednost sploh izračuna.

Pri izračunu nove vrednosti izraz `if` vstavimo v del pred `for`:

```
In [54]: seznam = [1, 2, 3]
         [el+10 if el<2 else el**2 for el in seznam]
```

```
Out[54]: [11, 4, 9]
```

Pogosto pa za določene elemente izračuna sploh ne želimo izvesti, v tem primeru izraz `if` vstavimo za seznam:

```
In [55]: [el**2 for el in seznam if el >1]
```

```
Out[55]: [4, 9]
```

2.7 Osnove modulov

Z moduli lahko na enostaven način uporabimo že napisano kodo. Preprosto povedano, moduli nam v Pythonu omogočajo relativno enostavno ohranjanje preglednosti in reda. Ponavadi v modul zapakiramo neko zaključeno funkcijo, skupino funkcij ali npr. objekt (pozneje bomo spoznali, kaj je objekt).

Za podroben opis uvažanja modulov glejte dokumentacijo[21].

V mapi `moduli` se nahaja datoteka `prvi_modul`, ki ima sledečo vsebino:

```
def kvadrat(x=1):
    return x**2
```

[21] https://docs.python.org/reference/import.html#the-import-system

Pomembno: pogosto module hierarhično razporejamo po direktorijih. **Samo** če je v določenem direktoriju datoteka (lahko tudi prazna) **__init__.py**, potem se tak direktorij obnaša kot modul. V zgornjem primeru taka datoteka obstaja in posledično se mapa `moduli` obnaša kot *modul*, v tem modulu je pa *podmodul* `prvi_modul`.

Module tipično uvažamo na dva načina:

Prvi način:

```
from moduli import prvi_modul
```

v tem primeru uvozimo imenski prostor ***prvi_modul***; to pomeni, da funkcijo v `kvadrat()` kličemo tako: `prvi_modul.kvadrat()`.

Drugi način uvažanja modula:

```
import moduli
```

Uvozimo *imenski prostor* ***moduli***; to pomeni, da funkcijo `kvadrat()` kličemo tako: `moduli.prvi_modul.kvadrat()`.

V kolikor želimo uvoziti funkcije v modulu `prvi_modul`, to naredimo tako:

```
In [56]: from moduli import prvi_modul
```

Sedaj lahko funkcijo `kvadrat` kličemo tako:

```
In [57]: prvi_modul.kvadrat(5)
```

```
Out[57]: 25
```

Z ukazom `from` lahko uvozimo samo del modula. Primer:

```
In [58]: from moduli.prvi_modul import kvadrat
```

Sedaj kličemo neposredno funkcijo `kvadrat`:

```
In [59]: kvadrat(6)
```

```
Out[59]: 36
```

Funkcije ali module lahko tudi preimenujemo. Primer:

```
In [60]: from moduli.prvi_modul import kvadrat as sqr
         sqr(7)
```

```
Out[60]: 49
```

Kje Python išče module?

1. V trenutni mapi (to je tista, v kateri ste prožili `jupyter notebook`),

- v mapah, definiranih v `PYTHONPATH`,
- v mapah, kjer je nameščen Python (poddirektorij `site-packages`).

2.7.1 Uvoz modulov in Jupyter notebook

V tej knjigi bomo, zaradi pedagoških razlogov, module vedno uvažali takrat, ko jih bomo prvič potrebovali. **Pravila dobre prakse in pregledne kode priporočajo, da se uvoze vseh modulov izvede na vrhu Jupyter notebooka!**

2.7.2 Modul `pickle`

Modul `pickle` bomo velikokrat uporabljali za zelo hitro in kompaktno shranjevanje in branje podatkov. `pickle` vrednosti spremenljivke shrani v taki obliki, kot so zapisane v spominu. V primeru racionalnih števil tako **ne izgublja na natančnosti** (zapis v tekstovno obliko izgublja natančnost, saj definiramo število izpisanih števk, ki pa je lahko manjše od števila decimalnih mest v spominu). Poleg tega pa je zapis v **spominu ponavadi manj zahteven** kakor zapis v tekstovni obliki. Ker vrednosti ni treba pretvarjati v/iz tekstovno/e obliko/e, je pisanje/branje tudi **zelo hitro**! Tukaj si bomo pogledali osnovno uporabo, celovito je `pickle` opisan v dokumentaciji[22].

`pickle` ima dve pomembni metodi:

- `dump(obj, file, protocol=None, *, fix_imports=True)` za shranjevanje objekta `obj`,
- `load(file, *, fix_imports=True, encoding="ASCII", errors=štrict")` za brane datoteke `file`.

Uporabo si bomo pogledali na primeru. Najprej uvozimo modul:

```
In [61]: import pickle
```

Zapišimo vrednosti slovarja `a` v datoteko `data/stanje1.pkl`:

```
In [62]: a = {'število': 1, 'velikost': 10}

         with open('data/stanje1.pkl', 'wb') as datoteka:
             pickle.dump(a, datoteka, protocol=-1) # pazite: uporabite protocol=-1, da boste uporabili
```

Opazimo, da so datoteko odprli za pisanje v binarni obliki `wb`. Pomembno je tudi, da uporabimo zadnjo verzijo protokola (parameter `protocol`); privzeta ni optimirana za hitrost in ne zapiše v binarni obliki.

Sedaj vrednosti preberimo nazaj v `b` in ga prikažemo:

```
In [63]: with open('data/stanje1.pkl', 'rb') as datoteka:
             b = pickle.load(datoteka)

         b

Out[63]: {'velikost': 10, 'število': 1}
```

Zgoraj smo trdili, da je pristop zelo hiter; preverimo to.

Najprej pripravimo podatke (seznam 50000 vrednosti tipa `ind`):

```
In [64]: data = list(range(50000))
```

Pisanje v tekstovni obliki

[22] https://docs.python.org/library/pickle.html

```
In [65]: %%timeit
         datoteka = open('data/data.txt', mode='w')
         for d in data:
             datoteka.write(f'{d:7.2e}\n' )
         datoteka.close()
```

```
269 ms ± 5.75 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Pisanje s pomočjo `pickle`:

```
In [66]: %%timeit
         with open('data/data.pkl', 'wb') as datoteka:
             pickle.dump(data, datoteka, protocol=-1)
```

```
11 ms ± 440 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Gre skoraj za 20 kratno pohitritev!

2.8 Nekaj vprašanj za razmislek!

1. Odprite datoteko za zapis in vanjo vpišite formatirani (skupaj 7 mest, 3 decimalna mesta) seznam v dveh stolpcih. Razločevalni znak med stolpci naj bo prazen znak ' '.
2. Odprite pripeto datoteko `zapis_labview.lvm` (gre za zapis iz programa LabView) in preberite glavo v obliki slovarja ('ime polja': vrednost).
3. V datoteki iz prejšnje točke preberite podatke (uporabite funkcijo `replace()` za zamenjavo decimalne vejice v piko in nato pretvorite niz v število. Najdite ustrezno dokumentacijo/help).
4. Na poljubnem seznamu besed prikažite uporabo izpeljevanja seznamov tako, da zamenjate poljuben samoglasnik.
5. Napišite funkcijo. Če se v funkcijo vstavi seznam besed, potem naj vrne dolžino besed. Če se v funkcjo vstavi seznam numeričnih vrednosti, potem naj vrne njihovo vrednost povečano za ena.
6. Zgornjo funkcijo nadgradite: če se v funkcijo vstavi prazen seznam (dolžine 0), potem naj sproži izjemo `Exception` z ustreznim opisom.
7. Pripravite novo funkcijo, ki bo klicala funkcijo iz točke 6 in lovila izjeme.
8. Za obe funkciji pripravite *docstring*.
9. Prikažite uporabo argumentov s privzetimi vrednostmi.
10. Prikažite uporabo argumentov glede na mesto in glede na ime.
11. V funkcijo pošljite nepredvidene vrednosti (nize in numerične vrednosti).
12. V funkcijo pošljite nepredvidene poimenske vrednosti.
13. Definirajte *lambda* funkcijo in jo uporabite v povezavi s funkcijo `max` ali `min`.
14. Zgornji funkciji shranite v modul poljubnega imena (npr.: prve_funkcije.py).
15. Uvozite samo eno od funkcij kot funkcijo z novim imenom (drugačno ime od izvornega).
16. Poljuben slovar shranite in odprite s `pickle`.

2.9 Dodatno

2.9.1 Oblikovanje datuma in časa

Najprej uvozimo modul za delo z datumom in časom `datetime` in prikažemo trenutni čas z datumom (uporabimo funkcijo `now()`[23]):

```
In [67]: import datetime
         sedaj = datetime.datetime.now()
         sedaj

Out[67]: datetime.datetime(2017, 11, 15, 7, 43, 33, 90131)
```

Podatek o času oblikujemo:

```
In [68]: f'{sedaj:%Y-%m-%d %H:%M:%S}'

Out[68]: '2017-11-15 07:43:33'
```

Oblikovanje datuma in ure sledi metodam:

- `strftime` (dokumentacija[24]) se uporablja za pretvorbo iz `datetime` v niz črk,
- `strptime` (dokumentacija[25]) se uporablja za pretvorbo iz niza črk v tip `datetime`.

Kako izpisati ustrezno obliko (torej kako uporabiti znake `%Y`, `%m` itd.) je prav tako podrobno pojasnjeno v dokumentaciji[26].

Poglejmo si primer pretvorbe tipa `datetime` v niz (glede na zgoraj smo dodali še %a, da se izpiše ime dneva):

```
In [69]: sedaj.strftime('%Y-%m-%d %H:%M:%S %a')

Out[69]: '2017-11-15 07:43:33 Wed'
```

Sedaj v obratno smer, torej iz niza v `datetime`:

```
In [70]: datetime.datetime.strptime('2017-09-28 05:31:45', '%Y-%m-%d %H:%M:%S')

Out[70]: datetime.datetime(2017, 9, 28, 5, 31, 45)
```

2.9.2 Uporabljajte www.stackoverflow.com!

http://www.stackoverflow.com

[23] https://docs.python.org/3/library/datetime.html#datetime.datetime.now
[24] https://docs.python.org/library/datetime.html#datetime.datetime.strftime
[25] https://docs.python.org/library/datetime.html#datetime.datetime.strptime
[26] https://docs.python.org/library/datetime.html#strftime-strptime-behavior

2.9.3 Nekateri moduli

1. `openpyxl` za pisanje in branje Excel xlsx/xlsm datotek: vir[27],

- Regularni izrazi (zelo močno orodje za iskanje po nizih): vir[28].

2.9.4 Modul `sys`

Python ima veliko število vgrajenih modulov (in še veliko se jih lahko prenese s spleta, npr. tukaj: https://pypi.python.org/pypi), ki dodajo različne funkcionalnosti.

Modul `sys` omogoča dostop do objektov, ki jih uporablja interpreter.

Poglejmo si primer; najprej uvozimo modul:

```
In [71]: import sys
```

Mape, kjer se iščejo moduli, lahko preverimo z ukazom `sys.path` (dokumentacija[29]):

```
In [72]: sys.path

Out[72]: ['',
          'C:\\Users\\Janko\\Anaconda3\\python36.zip',
          'C:\\Users\\Janko\\Anaconda3\\DLLs',
          'C:\\Users\\Janko\\Anaconda3\\lib',
          'C:\\Users\\Janko\\Anaconda3',
          'C:\\Users\\Janko\\Anaconda3\\lib\\site-packages',
          'C:\\Users\\Janko\\Anaconda3\\lib\\site-packages\\Babel-2.5.0-py3.6.egg',
          'C:\\Users\\Janko\\Anaconda3\\lib\\site-packages\\win32',
          'C:\\Users\\Janko\\Anaconda3\\lib\\site-packages\\win32\\lib',
          'C:\\Users\\Janko\\Anaconda3\\lib\\site-packages\\Pythonwin',
          'C:\\Users\\Janko\\Anaconda3\\lib\\site-packages\\IPython\\extensions',
          'C:\\Users\\Janko\\.ipython']
```

2.9.5 Modul `os`

Modul `os` je namenjen delu z operacijskim sistemom in skrbi za združljivost z različnimi operacijskimi sistemi (če npr. napišete program na sistemu *Windows*, bo delal tudi na sistemu *linux*).

Uvozimo modul:

```
In [73]: import os
```

Pogljemo trenutno mapo:

```
In [74]: os.path.curdir

Out[74]: '.'
```

ali trenutno mapo v absolutni obliki:

[27] http://openpyxl.readthedocs.org/en/latest/index.html#
[28] https://docs.python.org/library/re.html
[29] https://docs.python.org/2/library/sys.html#sys.path

```
In [75]: os.path.abspath(os.path.curdir)
```

```
Out[75]: 'c:\\_j\\Work Janko\\Pedagosko delo\\_ predavanja\\Programiranje in numerične metode\\pypinm'
```

Kako najdemo vse datoteke in mape v direktoriju?

```
In [76]: seznam = os.listdir()
```

Prikažimo prve štiri s končnico `ipynb`:

```
In [77]: i = 0
         for ime in seznam:
             if os.path.isfile(ime):
                 if ime[-5:] == 'ipynb':
                     print(f'Našel sem datoteko: {ime:s}')
                     i +=1
                 if i>3:
                     break
```

```
Našel sem datoteko: NM2017.ipynb
Našel sem datoteko: PiNM2016-17.ipynb
Našel sem datoteko: Predavanje 01 - Uvod v Python.ipynb
Našel sem datoteko: Predavanje 02 - Print, delo z datotekami, funkcije, moduli.ipynb
```

Za več glejte dokumentacijo[30].

[30] https://docs.python.org/3.7/library/os.path.html#module-os.path

Poglavje 3

Moduli, numpy, matplotlib

3.1 Moduli (nadaljevanje)

Poleg vgrajenih modulov in tistih, ki jih pripravimo sami, obstaja še velika množica modulov, ki jih lahko najdemo na spletu in Pythonu dodajajo nove funkcionalnosti. Šele ti moduli naredijo ekosistem Pythona tako uporaben.

Anaconda[1] je najbolj popularna distribucija Pythona in ima vključenih že veliko modulov (za podroben seznam glejte to povezavo[2]); predvsem pa ima vključene vse bistvene. Najbolj pomembne na področju inženirskih ved bomo spoznali v okviru te knjige.

Modul je tehnično gledano ena datoteka, kadar nek večji modul vsebuje več modulov, pa lahko začnemo govoriti o *paketih*.

Module oz pakete lahko *posodabljamo* ali nameščamo *nove*, pri tem nam pomagajo t. i. *upravljalniki paketov* (angl. *package manager*). Najbolj pogosto uporabljamo:

- `pip`: dokumentacija[3],
- `conda`: dokumentacija[4].

Upravljalnika paketov nista povsem ekvivalentna, pogosto uporabo si bomo pogledali spodaj.

3.1.1 Upravljalnik paketov `conda`

`conda` uporabljamo predvsem za module/pakete vključene v distribucijo *Anaconda*, in ima še nekatere dodatne sposobnosti (npr. kreiranje navideznega okolja, angl. *virtual environment*).

Če želimo posodobiti vse nameščene pakete znotraj distribucije Anaconda, najprej posodobimo samo `conda`, nato pa pakete. V ukazni vrstici izvedemo sledeča ukaza:

```
conda update conda
conda update --all
```

Kateri paketi so nameščeni, preverimo z ukazom (v ukazni vrstici):

[1] https://www.anaconda.com/download/
[2] https://docs.anaconda.com/anaconda/packages/pkg-docs
[3] https://pip.pypa.io/en/stable/
[4] https://conda.io/docs/

```
conda list
```

Namesto izhoda v ukazno vrstico bomo večkrat uporabljali možnost *Jupyter notebooka*, ko s pomočjo klicaja (!) v celici s kodo izvedemo ukaz v ukazni vrstici (gre za kratko obliko magičnega ukaza `%se` - shell execute, glejte dokumentacijo[5]). Opomba: dva klicaja bi vrnila celoten rezultat neposredno v notebook (ker je izpis zelo dolg, smo se temu izognili).

Poglejmo primer:

```
In [1]: rezultat = !conda list
        rezultat[:5]

Out[1]: ['# packages in environment at c:\\Users\\Janko\\Anaconda3:',
         '#',
         '_ipyw_jlab_nb_ext_conf    0.1.0            py36he6757f0_0  ',
         'alabaster                 0.7.10           py36hcd07829_0  ',
         'anaconda                  custom           py36h363777c_0  ']
```

Pakete namestimo z ukazom (v ukazni vrstici):

```
conda install [ime paketa]
```

in odstranimo z:

```
conda remove [ime paketa]
```

Za več glejte dokumentacijo[6].

3.1.2 Upravljalnik paketov `pip`

`pip` je upravljalnik paketov z daljšo zgodovino kot `conda`; podpira bistveno več paketov (pypi.python.org/pypi[7]), vendar ni tako odporen na nezdružljivosti kakor `conda`. Razlik je še več, vendar tukaj ne bomo šli v podrobnosti; uporabljate tistega, ki vam namesti željeni paket!

Podobno kot pri `conda`, tudi tukaj že nameščene pakete najdemo z ukazom v ukazni vrstici (dokumentacija[8]):

```
pip list
```

Pakete s `pip` namestimo z:

```
pip install [ime_paketa]
```

posodobimo z:

```
pip install [ime_paketa] --upgrade
```

in odstranimo z:

```
pip uninstall [ime_paketa]
```

[5] http://ipython.readthedocs.io/en/stable/interactive/magics.html#magic-sx
[6] https://conda.io/docs/user-guide/getting-started.html
[7] https://pypi.python.org/pypi
[8] https://pip.pypa.io/en/stable/reference/pip_list/

Primer namestitve paketa

Sicer pa pakete najpogosteje najdemo na spletu (pypi.python.org[9]). Pojdite na omenjeni portal in poiščite pakete na temo snemanja posnetkov iz portala www.youtube.com[10]. Z iskanjem "youtube download"najdemo obetaven paket `youtube_dl`, ki ga namestimo:

```
In [2]: !!pip install "youtube_dl" --upgrade
```

```
Out[2]: ['Collecting youtube_dl',
         '  Downloading youtube_dl-2017.11.15-py2.py3-none-any.whl (1.7MB)',
         'Installing collected packages: youtube-dl',
         '  Found existing installation: youtube-dl 2017.11.6',
         '    Uninstalling youtube-dl-2017.11.6:',
         '      Successfully uninstalled youtube-dl-2017.11.6',
         'Successfully installed youtube-dl-2017.11.15']
```

Sedaj modul uporabimo (za podrobnosi uporabe glejte dokumentacijo[11]). Najprej uvozimo celotni paket:

```
In [3]: import youtube_dl
        yt = youtube_dl.YoutubeDL() # kreiramo instanco objekta (kaj to točno pomeni, spoznamo pozneje)
```

Prenesemo poljubni video:

```
In [4]: yt.download(url_list = ['7VGiJN-sCKk'])
```

```
[youtube] 7VGiJN-sCKk: Downloading webpage
[youtube] 7VGiJN-sCKk: Downloading video info webpage
[youtube] 7VGiJN-sCKk: Extracting video information
[youtube] 7VGiJN-sCKk: Downloading MPD manifest
[download] HOZENTREGARJI- JUTR TOČIMO ZASTONJ-7VGiJN-sCKk.mp4 has already been downloaded
[download] 100% of 8.14MiB


Out[4]: 0
```

3.2 Modul `numpy`

Kakor je omenjeno zgoraj, so v okviru *Anaconda* distribucije Pythona že nameščeni praktično vsi pomembni moduli. V kolikor bi namestili Python *the hard way*, bi sicer morali modul `numpy` namestiti posebej (glejte spletno stran `numpy`[12]).

3.2.1 Osnove modula `numpy`

Najprej uvozimo modul (uveljavljeno je, da ga uvozimo v kratki obliki `np`):

```
In [5]: import numpy as np
```

[9] https://pypi.python.org/pypi
[10] https://www.youtube.com
[11] http://rg3.github.io/youtube-dl/
[12] http://www.numpy.org

Gre za enega najbolj pomembnih modulov. Na kratko: gre za visoko optimiran modul za numerične izračune!

Poglejmo si najprej sintakso za vektor ničel (dokumentacija[13]):

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

argumenti so:

- `shape` definira obliko (lahko večdimenzijsko numerično polje),
- `dtype` definira tip podatka,
- `order` definira vrstni red (lahko je C ali F kot Fortran).

Poglejmo primer:

```
In [6]: np.zeros(3)

Out[6]: array([ 0.,  0.,  0.])
```

ali pa:

```
In [7]: np.zeros((3,5))

Out[7]: array([[ 0.,  0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.,  0.],
               [ 0.,  0.,  0.,  0.,  0.]])
```

Podobno kot `zeros` se obnaša `ones`, vendar je namesto ničel vrednost 1 (dokumentacija[14]).

Poglejmo si primer, kjer definiramo tudi tip `int` (privzeti tip je `float`):

```
In [8]: np.ones(4, dtype=int)

Out[8]: array([1, 1, 1, 1])
```

Pogosto bomo tudi uporabljali razpon vrednosti `arange` (dokumentacija[15]):

```
numpy.arange([start, ]stop, [step, ]dtype=None)
```

kjer so argumenti:

- `start` začetna vrednost razpona (privzeto 0),
- `stop` končna vrednost razpona,
- `step` korak in
- `dtype` tip vrednosti (če tip ni podan, se vzame tip ostalih argumetnov, npr. `step`).

Poglejmo primer razpona od 0 do 9 (kakor vedno pri Pythonu *od* je vključen, *do* pa ni):

```
In [9]: np.arange(9)
```

[13] https://docs.scipy.org/doc/numpy/reference/generated/numpy.zeros.html
[14] https://docs.scipy.org/doc/numpy/reference/generated/numpy.ones.html#numpy.ones
[15] https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html

```
Out[9]: array([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

ali pa od 7 do 12 po koraku 2, vendar število s plavajočo vejico:

```
In [10]: np.arange(7, 12, 2, dtype=float)
```

```
Out[10]: array([  7.,   9.,  11.])
```

Še eno funkcijo bomo pogosto uporabili, to je `linspace` (dokumentacija[16]):

```
numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
```

ki definiranemu razponu vrne numerično polje vrednosti na enaki razdalji (ekvidistanten razmik).

Argumenti so:

- `start` začetna vrednost razpona,
- `stop` končna vrednost razpona,
- `num` število točk/vozlišč,
- `endpoint` ali je vrednost pri `stop` vključena ali ne,
- `retstep` v primeru `True` vrne funkcija terko `(rezultat, korak)`
- `dtype` tip vrednosti (če tip ni podan, se vzame tip ostalih argumetnov, npr. `step`).

Primer generiranja 10 točk na razponu od $-\pi$ do vključno $+\pi$:

```
In [11]: np.linspace(-np.pi, np.pi, 10)
```

```
Out[11]: array([-3.14159265, -2.44346095, -1.74532925, -1.04719755, -0.34906585,
                 0.34906585,  1.04719755,  1.74532925,  2.44346095,  3.14159265])
```

Mimogrede smo zgoraj spoznali, da ima `numpy` vgrajene konstante (npr. π):

Poglejmo še vgrajene funkcije za generiranje naključnih števil. Te najdemo v `numpy.random` (dokumentacija[17]).

Najprej si poglejmo funkcijo `numpy.random.seed()`, ki se uporablja za ponastavitev generatorja naključnih števil (dokumetnacija[18]). To pomeni, da lahko z istim semenom (angl. *seed*) različni uporabniki generiramo ista naključna števila!

Spodnja vrstica:

```
In [12]: np.random.seed(0)
```

bo povzročila, da bo klic generatorja naključnih števil z enakomerno porazdelitvijo `numpy.random.rand()` (dokumentacija[19]) vedno rezultiral v iste vrednosti:

```
In [13]: np.random.rand(3)
```

```
Out[13]: array([ 0.5488135 ,  0.71518937,  0.60276338])
```

[16] https://docs.scipy.org/doc/numpy/reference/generated/numpy.linspace.html
[17] https://docs.scipy.org/doc/numpy/reference/routines.random.html
[18] https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.seed.html#numpy.random.seed
[19] https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.rand.html#numpy.random.rand

Preizkusimo ali je res, kar smo zapisali, in ponastavimo seme in kličimo generator:

```
In [14]: np.random.seed(0)
         np.random.random(3)

Out[14]: array([ 0.5488135 ,  0.71518937,  0.60276338])
```

Zapis matrik in vektorjev

Matrika je dimenzije `m x n`, kjer je na prvem mestu `m` število vrstic in `n` število stolpcev. Primer definiranja matrike dimenzije `m x n = 3 x 2` je:

```
In [15]: a = np.zeros((3, 2))
         a

Out[15]: array([[ 0.,  0.],
                [ 0.,  0.],
                [ 0.,  0.]])
```

Vektor je lahko zapisan kot **vrstični vektor**:

```
In [16]: b = np.zeros(3) # (1 x 3)
         b

Out[16]: array([ 0.,  0.,  0.])
```

ali kot **stolpični vektor**:

```
In [17]: c = np.zeros((3, 1)) # 3 x 1
         c

Out[17]: array([[ 0.],
                [ 0.],
                [ 0.]])
```

V modulu `numpy` lahko vektorje in matrike zapisujemo kot:

- `numpy.array` (priporočeno, dokumentacija[20]),
- `numpy.matrix` (dokumentacija[21]).

Priporočena je prva oblika (`numpy.array`), ki pa ne sledi povsem matematičnemu zapisu (več o tem pozneje), nam pa omogoča **enostavnejše** programiranje in je tudi numerično **bolj učinkovit** pristop (vir[22]). Pristopa `numpy.matrix` tukaj ne bomo obravnavali.

V angleškem jeziku bomo *array* prevajali kot **večdimenzijsko numerično polje** ali včasih **večdimenzijske sezname** (ker imajo nekatere podobnosti z navadnimi seznami). Nekatere knjige *array* tukaj prevajajo kot *tabela*.

[20] https://docs.scipy.org/doc/numpy/reference/generated/numpy.array.html
[21] https://docs.scipy.org/doc/numpy/reference/generated/numpy.matrix.html
[22] http://wiki.scipy.org/NumPy_for_Matlab_Users#head-e9a492daa18afcd86e84e07cd2824a9b1b651935

Rezanje

Rezanje (angl. *slicing*) seznamov smo si že pogledali v poglavju Uvod v Python. Podobno rezanje, vendar bolj splošno, velja tudi za numerična polja modula `numpy`.

Sintaksa rezanja (dokumentacija[23]) je:

```
numpy_array[od:do:korak]
```

pri tem velja:

- indeksiranje se začne z 0 (kot sicer pri Pythonu),
- `od` pomeni **>=**,
- `do` pomeni **<**,
- `od`, `do`, `korak` so opcijski parametri,
- če parameter `od` ni podan, pomeni od *začetka*,
- če parameter `do` ni podan, pomeni do vključno zadnjega,
- če parameter `korak` ni podan, pomeni korak 1.

Primer od elementa 3 do elementa 8 po koraku 2:

```
In [18]: b = np.arange(10)
         b[3:8:2]

Out[18]: array([3, 5, 7])
```

Primer od elementa 3 naprej:

```
In [19]: b[3:]

Out[19]: array([3, 4, 5, 6, 7, 8, 9])
```

Primer od elementa 3 naprej, vendar vsak tretji:

```
In [20]: b[3::3]

Out[20]: array([3, 6, 9])
```

Primer zadnjih 5 elementov, vendar vsak drugi:

```
In [21]: b[-5::2]

Out[21]: array([5, 7, 9])
```

Primer zadnjih 5 elementov brez zadnjih 2, vendar vsak drugi:

```
In [22]: b[-5:-2:2]

Out[22]: array([5, 7])
```

[23] http://docs.scipy.org/doc/numpy/reference/arrays.indexing.html

Poglejmo si še rezanje večdimenzijskega numeričnega polja. Večdimenzijsko rezanje izvedemo tako, da dimenzije ločimo z vejico:

```
numpy_array[rezanje0, rezanje1,...]
```

kjer `rezanje0` reže indeks 0 (prvo dimenzijo) v obliki `od:do:korak`, `rezanje1` reže indeks 1 (drugo dimenzijo) in tako naprej.

Poglejmo si primer; najprej pripravimo seznam 15 števil (`np.arange`), nato z metodo `reshape()` (dokumentacija[24]) spremenimo obliko v matriko `3 x 5`:

```
In [23]: a = np.arange(15)
         a = a.reshape((3,5))
         a

Out[23]: array([[ 0,  1,  2,  3,  4],
                [ 5,  6,  7,  8,  9],
                [10, 11, 12, 13, 14]])
```

Obliko numeričnega polja lahko preverimo z atributom `shape` (dokumentacija[25]).

```
In [24]: a.shape

Out[24]: (3, 5)
```

Atribute bomo sicer podrobno spoznali pri obravnavi razredov.

Prikažimo vrstice z indeksom 0:

```
In [25]: a[0]

Out[25]: array([0, 1, 2, 3, 4])
```

Isti rezultat bi dobili z prikazom vrstice z indeksom 0 in vseh stolpcev:

```
In [26]: a[0,:]

Out[26]: array([0, 1, 2, 3, 4])
```

Pogosto želimo dostopati do stoplcev, npr. stolpca z indeksom 1 (torej režemo vse vrstice in stolpec z indeksom 1):

```
In [27]: a[:,1]

Out[27]: array([ 1,  6, 11])
```

Poglejmo si še primer rezanja prvih dveh vrstic in zadnjih dveh stolpcev:

```
In [28]: a[:2, -2:]

Out[28]: array([[3, 4],
                [8, 9]])
```

Podobna logika se uporabi pri dimenzijah višjih od 2.

[24] https://docs.scipy.org/doc/numpy/reference/generated/numpy.reshape.html
[25] https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.shape.html

Operacije nad numeričnimi polji

Poglejmo si sedaj bolj podrobno nekatere osnovne prednosti numeričnega polja `numpy.array` v primerjavi z navadnim seznamom Python.

Najprej pripravimo navaden Pythonov seznam:

```
In [29]: a = [1, 2, 3, 4, 5, 6, 7]
         a

Out[29]: [1, 2, 3, 4, 5, 6, 7]
```

In nato še numerično polje `numpy.array` (kar iz seznama `a`):

```
In [30]: b = np.array(a)
         b

Out[30]: array([1, 2, 3, 4, 5, 6, 7])
```

Ko izpišemo `b`, smo opozorjeni, da gre za `array([...])`.

Pogljemo, kako se obnaša Pythonov seznam pri množenju:

```
In [31]: 2*a

Out[31]: [1, 2, 3, 4, 5, 6, 7, 1, 2, 3, 4, 5, 6, 7]
```

Opazimo, da se podvoji seznam, ne pa vrednosti, kar bi morebiti pričakovali.

Poglejmo, kako se pri množenju obnaša numerično polje `numpy.array`:

```
In [32]: 2*b

Out[32]: array([ 2,  4,  6,  8, 10, 12, 14])
```

Opazimo, da se podvojijo vrednosti; tako kakor bi pričakovali, ko množimo na primer skalarno vrednost in vektor!

Aritmetične operacije

Izbor aritmetičnih operacij, ki jih `numpy` izvaja na nivoju posameznega elementa, je (po naraščajoči prioriteti):

- `x + y` vsota,
- `x - y` razlika,
- `x * y` produkt,
- `x / y` deljenje,
- `x // y` celoštevilsko deljenje (rezultat je celo število zaokroženo navzdol),

- `x % y` ostanek pri celoštevilskem deljenju,
- `x ** y` vrne `x` na potenco `y`.

Primer:

```
In [33]: b + 3*b - b**2

Out[33]: array([  3,   4,   3,   0,  -5, -12, -21])
```

Vse aritmetične operacijo so sicer navedene v dokumentaciji[26] in namesto kratkih oblik imamo tudi *dolge*, npr: `numpy.power(x, y)` namesto `x**y`; primer:

```
In [34]: np.power(b, 2)

Out[34]: array([ 1,  4,  9, 16, 25, 36, 49], dtype=int32)
```

Mimogrede opazimo, da je rezultat tipa `int32` (integer). Ko smo ustvarili ime `b`, smo namreč ustvarili numerično polje z elementi tipa `int32`.

Matematične funkcije

`numpy` ponuja praktično vse potrebne matematične (in druge) operacije, navedimo jih po skupinah, kot so strukturirane v dokumentaciji[27]:

- trigonometrične funkcije[28],
- hiperbolične[29],
- funkcije za zaokroževanje[30],
- funkcije za vsoto, produkt in odvod[31],
- eksponenti in logaritmi[32],
- posebne[33] ter preostale[34] funkcije.

Poglejmo primer funkcije sin():

```
In [35]: a = np.linspace(0, 2*np.pi, 5)
         np.sin(a)

Out[35]: array([  0.00000000e+00,   1.00000000e+00,   1.22464680e-16,
                 -1.00000000e+00,  -2.44929360e-16])
```

Poglejmo še hitrost izvajanja:

```
In [36]: %timeit -n100 np.sin(a)

1.36 µs ± 338 ns per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

[26] https://docs.scipy.org/doc/numpy/reference/routines.math.html#arithmetic-operations
[27] https://docs.scipy.org/doc/numpy/reference/routines.math.html#mathematical-functions
[28] https://docs.scipy.org/doc/numpy/reference/routines.math.html#trigonometric-functions
[29] https://docs.scipy.org/doc/numpy/reference/routines.math.html#hyperbolic-functions
[30] https://docs.scipy.org/doc/numpy/reference/routines.math.html#rounding
[31] https://docs.scipy.org/doc/numpy/reference/routines.math.html#sums-products-differences
[32] https://docs.scipy.org/doc/numpy/reference/routines.math.html#exponents-and-logarithms
[33] https://docs.scipy.org/doc/numpy/reference/routines.math.html#other-special-functions
[34] https://docs.scipy.org/doc/numpy/reference/routines.math.html#miscellaneous

Podatkovni tipi

numpy ima vnaprej definirane podatkovne tipe (statično). Celoten seznam možnih tipov je naveden v dokumentaciji[35].

Osredotočili se bomo predvsem na sledeče tipe:

- `int` - celo število (poljubno veliko)
- `float` - število s plavajočo vejico (dokumentacija[36])
- `complex` - kompleksno število s plavajočo vejico
- `object` - python objekt.

Poglejmo si nekaj primerov (cela števila, število s plavajočo vejico in kompleksna števila):

```
In [37]: np.arange(5, dtype=int)

Out[37]: array([0, 1, 2, 3, 4])

In [38]: np.arange(5, dtype=float)

Out[38]: array([ 0.,  1.,  2.,  3.,  4.])

In [39]: np.arange(5, dtype=complex)

Out[39]: array([ 0.+0.j,  1.+0.j,  2.+0.j,  3.+0.j,  4.+0.j])
```

Spreminjanje elementov numeričnega polja (`numpy.array`)

Podatke spreminjamo na podoben način kakor pri navadnih seznamih; v kolikor uporabljamo rezanje, moramo paziti, da so na levi in desni strani enačaja podatki iste oblike (`array.shape`).

Poglejmo si primer, ko matriki ničel dimenzije `3 x 4` spremenimo element z indeksom `[2, 3]`:

```
In [40]: a = np.zeros((3, 4))
         a[2, 3] = 100
         a

Out[40]: array([[   0.,    0.,    0.,    0.],
                [   0.,    0.,    0.,    0.],
                [   0.,    0.,    0.,  100.]])
```

Sedaj spremenimo še elemente prvih dveh vrstic in prvih dveh stolpcev v vrednost 1:

```
In [41]: a[:2, :2] = np.ones((2, 2))
         a

Out[41]: array([[   1.,    1.,    0.,    0.],
                [   1.,    1.,    0.,    0.],
                [   0.,    0.,    0.,  100.]])
```

[35] http://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html
[36] https://docs.python.org/dev/library/functions.html#float

Z 2 pomnožimo stolpec z indeksom 1:

```
In [42]: a[:,1] = 2 * a[:,1]
         a

Out[42]: array([[   1.,    2.,    0.,    0.],
                [   1.,    2.,    0.,    0.],
                [   0.,    0.,    0.,  100.]])
```

Bodite pozorni na to, da na tak način naredimo *pogled* (view) na podatke (**ne naredimo kopije podatkov**). Za primer najprej naredimo novo ime `pogled`:

```
In [43]: pogled = a[:, 2]
         pogled

Out[43]: array([ 0.,  0.,  0.])
```

Sedaj spremenimo izbrane vrednosti numeričnega polja `a`:

```
In [44]: a[:, 2] = 5
         a

Out[44]: array([[   1.,    2.,    5.,    0.],
                [   1.,    2.,    5.,    0.],
                [   0.,    0.,    5.,  100.]])
```

Vrednosti `pogled` nismo spreminjali. Ker pa ime kaže na isto mesto kakor `a[:, 2]`, so vrednosti spremenjene:

```
In [45]: pogled

Out[45]: array([ 5.,  5.,  5.])
```

Če želimo kopijo, potem moramo narediti tako:

```
In [46]: kopija = a[:, 2].copy()
         kopija

Out[46]: array([ 5.,  5.,  5.])
```

in rezultat `kopija` ostane nespremenjen:

```
In [47]: a[:, 2] = 2
         kopija

Out[47]: array([ 5.,  5.,  5.])
```

3.2.2 Osnove matričnega računanja

Če želite ponoviti matematične osnove matričnega računanja, potem sledite tej povezavi[37] (gre za kratek in dober pregled prof. dr. T. Koširja). Pogledali bomo, kako matrične račune izvedemo s pomočjo paketa `numpy`.

Najprej definirajmo matriki **A** in **B**:

```
In [48]: A = np.array([[1, 2], [3, 2]])
         B = np.array([[1, 1], [2, 2]])
```

ter vektorja **x** in **y**.

```
In [49]: x = np.array([1, 2])
         y = np.array([3, 4])
```

Skalarni produkt dveh vektorjev izvedemo s funkcijo `dot()` (dokumentacija[38]):

```
numpy.dot(a, b, out=None)
```

kjer argumenta `a` in `b` predstavljata numerični polji `numpy.array` (poljubne dimenzije), ki jih želimo množiti. Če sta `a` in `b` dimenzije 1 se izvede skalarni produkt. Pri dimenziji 2 (matrike) se izračuna produkt matrik. Za uporabo funkcije `dot()` pri dimenzijah več kot 2: glejte dokumentacijo[39].

Poglejmo primer množenja dveh vektorjev, to lahko izvedemo tako:

```
In [50]: np.dot(x, y)

Out[50]: 11
```

ali tudi tako:

```
In [51]: x.dot(y)

Out[51]: 11
```

Zgoraj smo omenili, da `numpy.array` ne sledi dosledno matematičnemu zapisu. Če bi, bi namreč eden od vektorjev moral biti vrstični, drugi stolpični. `numpy` to poenostavi in zato je koda lažje berljiva in krajša.

Poglejmo sedaj množenje matrike z vektorjem (opazimo, da transponiranje `x` ni potrebno):

```
In [52]: np.dot(A, x)

Out[52]: array([5, 7])
```

Lahko pa seveda pripravimo matematično korektno transponirano obliko vektorja (ampak vidimo, da je zapis neroden):

```
In [53]: A.dot(np.transpose([x]))
```

[37] http://www.fmf.uni-lj.si/~kosir/poucevanje/skripta/matrike.pdf
[38] https://docs.scipy.org/doc/numpy/reference/generated/numpy.dot.html
[39] https://docs.scipy.org/doc/numpy/reference/generated/numpy.dot.html

```
Out[53]: array([[5],
                [7]])
```

Transponiranje ima sicer tudi kratko obliko, prek atributa `T`, npr. za matriko **A**:

```
In [54]: A.T

Out[54]: array([[1, 3],
                [2, 2]])
```

Poglejmo si primer množenja dveh matrik:

```
In [55]: np.dot(A, B)

Out[55]: array([[5, 5],
                [7, 7]])
```

Od Pythona 3.5 naprej se za množenje matrik (in vektorjev) uporablja tudi operator `@` (dokumentacija[40]), ki omogoča kratek in pregleden zapis.

Zgornji primeri zapisani z operatorjem `@`:

```
In [56]: x @ y

Out[56]: 11

In [57]: A @ x

Out[57]: array([5, 7])

In [58]: A @ B

Out[58]: array([[5, 5],
                [7, 7]])
```

Vektorski produkt izračunamo s funkcijo `numpy.cross()` (dokumentacija[41]):

```
numpy.cross(a, b, axisa=-1, axisb=-1, axisc=-1, axis=None)
```

kjer `a` in `b` definirata komponente vektorjev. Če sta podani samo dve komponenti (x in y), se izračuna skalarna vrednost (komponenta z); če so podane tri komponente, je rezultat tudi vektor s tremi komponentami. Uporaba funkcije je možna tudi na večdimenzijskih numeričnih poljih in temu so namenjeni preostali argumenti (glejte dokumentacijo).

Primer vektorskega produkta ravninskih vektorjev:

```
In [59]: x = np.array([1, 0])
         y = np.array([0, 1])
         np.cross(x, y)
```

[40] https://docs.scipy.org/doc/numpy/reference/generated/numpy.matmul.html#numpy.matmul
[41] https://docs.scipy.org/doc/numpy/reference/generated/numpy.cross.html

```
Out[59]: array(1)
```

Primer vektorskega produkta prostorskih vektorjev:

```
In [60]: x = np.array([1, 0, 0])
         y = np.array([0, 1, 0])
         np.cross(x, y)

Out[60]: array([0, 0, 1])
```

Nekatere funkcije knjižnice `numpy`

Pogledali si bomo še nekatere funkcije, ki jih bolj ali manj pogosto potrebujemo.
Enotsko matriko definiramo s funkcijo `numpy.identity()` (dokumentacija[42]):

```
numpy.identity(n, dtype=None)
```

kjer argument `n` definira število vrstic in stolpcev pravokotne matrike. Tip `dtype` je privzeto `float`.
Primer enotske matrike:

```
In [61]: A = np.identity(3)
         A

Out[61]: array([[ 1.,  0.,  0.],
                [ 0.,  1.,  0.],
                [ 0.,  0.,  1.]])
```

Do diagonalnih elementov matrike dostopamo s pomočjo funkcije `numpy.diagonal()` (dokumentacija[43])

```
numpy.diagonal(a, offset=0, axis1=0, axis2=1)
```

Če je matrika dvodimenzijska, potem funkcija s privzetimi argumenti vrne diagonalno os. Če je dimenzija višja od 2, se uporabi osi `axis1` in `axis2`, da se izloči dvodimenzijsko polje, nato pa določi diagonalo glede na elemente `[i, i+offset]`.
Poglejmo primer izločanja diagonale:

```
In [62]: np.diagonal(A)

Out[62]: array([ 1.,  1.,  1.])
```

in uporabe `offset=1` za sosednjo diagonalo (najprej pripravimo nesimetirčno matriko):

```
In [63]: A[0, 1] = 10
         np.diagonal(A, offset=1)

Out[63]: array([ 10.,   0.])
```

[42] https://docs.scipy.org/doc/numpy/reference/generated/numpy.identity.html
[43] https://docs.scipy.org/doc/numpy/reference/generated/numpy.diagonal.html

Podobno sintakso kot `numpy.diagonal()` ima funkcija `numpy.trace()`, ki izračuna vsoto (sled) diagonalnih elementov (dokumentacija[44]):

```
numpy.trace(a, offset=0, axis1=0, axis2=1,
            dtype=None, out=None)
```

Primer sledi diagonale:

```
In [64]: np.trace(A)

Out[64]: 3.0
```

in potem sosednje diagonale:

```
In [65]: np.trace(A, offset=1)

Out[65]: 10.0
```

Pogosto nas zanimata največji ali najmanjši element nekega numeričnega polja. `numpy` je tukaj zelo splošen. Poglejmo si na primeru funkcije `numpy.max()` (dokumentacija[45]):

```
numpy.max(a, axis=None, out=None)
```

Izpostavimo argument `axis`, ki pove, čez kateri indeks iščemo maksimalno vrednost. Če je `axis=None` se določi največja vrednost v celotnem polju.

Primer izračuna največje vrednosti celotnega polja (prej poglejmo `A`):

```
In [66]: A

Out[66]: array([[  1.,  10.,   0.],
                [  0.,   1.,   0.],
                [  0.,   0.,   1.]])

In [67]: np.max(A)

Out[67]: 10.0
```

Primer izračuna največje vrednosti čez vrstice (torej po stolpcih):

```
In [68]: np.max(A, axis=0)

Out[68]: array([  1.,  10.,   1.])
```

Primer izračuna največje vrednosti čez stolpce (torej po vrsticah):

```
In [69]: np.max(A, axis=1)

Out[69]: array([ 10.,   1.,   1.])
```

Par funkcije `max()` je `numpy.argmax()`, kateri določi indekse največje vrednosti.

Primer uporabe:

```
In [70]: np.argmax(A, axis=0)

Out[70]: array([0, 0, 2], dtype=int64)
```

[44] https://docs.scipy.org/doc/numpy/reference/generated/numpy.trace.html
[45] https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.amax.html

Linearna algebra z `numpy`

Pozneje bomo linearno algebro bolj podrobno spoznali in bomo sami pisali algoritme. Tukaj si poglejmo nekatere osnove, ki so vgrajene v modul `numpy`.

Za primer najprej definirajmo matriko in vektor:

```
In [71]: A = np.array([[4, -2],
                       [-2, 4]])
         b = np.array([1, 2])
```

Inverzno matriko izračunamo z uporabo funkcije `numpy.linalg.inv()` (dokumentacija[46]):

```
numpy.linalg.inv(a)
```

Primer:

```
In [72]: np.linalg.inv(A)
```

```
Out[72]: array([[ 0.33333333,  0.16666667],
                [ 0.16666667,  0.33333333]])
```

Sistem linearnih enačb, ki ga definirata matrika koeficientov `a` in vektor konstant `b`, rešimo s pomočjo funkcije `numpy.linalg.solve()` (dokumentacija[47]):

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

Primer:

```
In [73]: rešitev = np.linalg.solve(A, b)
         rešitev
```

```
Out[73]: array([ 0.66666667,  0.83333333])
```

Enakost elementov numeričnega polja `a` in `b` (znotraj določene tolerance) preverimo s funkcijo `numpy.isclose()` (dokumentacija[48]):

```
numpy.isclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)
```

Primer:

```
In [74]: np.isclose(np.dot(A, rešitev), b)
```

```
Out[74]: array([ True,  True], dtype=bool)
```

[46] https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.inv.html
[47] https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.solve.html
[48] https://docs.scipy.org/doc/numpy/reference/generated/numpy.isclose.html

3.2.3 Vektorizacija algoritmov

V tem poglavju želimo izpostaviti vektorizacijo algoritmov. Glede na to kako Python in `numpy` delujeta se je potrebno izogibati zankam. Bistveno hitreje lahko izvajamo izračune, če jih uspemo vektorizirati; to pomeni, da izračune izvajamo na nivoju vektorjev (oz. numeričnih polj) in ne elementov.

Za primer si najprej pripravimo podatke (dva vektorja dolžine 1000)

```
In [75]: N = 1000
         a = np.arange(N)
         b = np.arange(N)
```

Izračunajmo skalarni produkt vektorjev z uporbo zanke `for`:

```
In [76]: c = 0
         for i in range(N):
             c += a[i] * b[i]
         c

Out[76]: 332833500
```

Izmerimo hitrost:

```
In [77]: %%timeit -n100
         c = 0
         for i in range(N):
             c += a[i] * b[i]

869 μs ± 56.6 μs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Isti rezultat pridobimo še v vektorski obliki:

```
In [78]: %%timeit -n100
         c = a @ b

The slowest run took 4.12 times longer than the fastest. This could mean that an intermediate result is
5.75 μs ± 4.02 μs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Vidimo, da je vektorski način bistveno hitrejši (za še hitrejši način glejte `numba` v dodatku)!

3.3 Modul `matplotlib`

V Pythonu imamo več možnosti za prikaz rezultatov v grafični obliki. Najbolj uporabni paketi so:

- `matplotlib`[49] za visoko kakovostne, visoko prilagodljive slike (relativno počasno),
- `pyqtgraph`[50] za kakovostne in prilagodljive uporabniške vmesnike (zelo hitro),

[49] http://matplotlib.org/
[50] http://www.pyqtgraph.org/

- bokeh[51] za interaktiven prikaz v brskalniku (relativno hitro).

Obstaja še veliko drugih; dober pregled je naredil Jake VanderPlas na konferenci PyCon 2017[52] (sicer avtor paketa za deklerativno vizualizacijo: *Altair*).

3.3.1 Osnovna uporaba

Najbolj razširjen in najbolj splošno uporabljen je paket matplotlib[53]:

Sposobnosti paketa najbolje prikazuje galerija[54]. Gre za zelo sofisticiran paket in tukaj si bomo na podlagi primerov pogledali nekatere osnove.

Tipično uvozimo `matplotlib.pyplot` kot `plt`:

```
In [79]: import matplotlib.pyplot as plt
```

Znotraj *Jupyter notebooka* obstajata dva načina prikaza slike (v oglatem oklepaju je magic ukaz za proženje):

1. [`%matplotlib inline`]: slike so vključene v notebook (**medvrstični** način),
2. [`%matplotlib notebook`]: slike so vključene interaktivno v notebook (**medvrstični interaktivni** način).

Opomba: *interaktivni način se v pasivni, spletni/pdf verziji te knjige ne prikaže pravilno.*

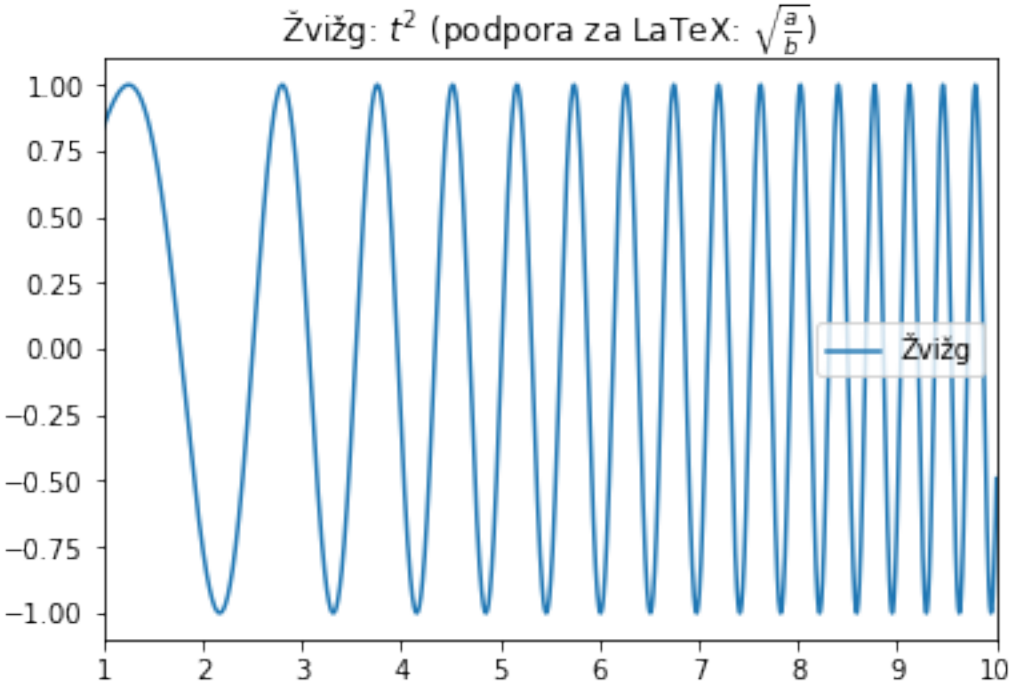
Tukaj bomo najpogosteje uporabljali *medvrstični* način:

```
In [80]: %matplotlib inline
```

Kratek primer:

```
In [81]: t = np.linspace(1, 130, 44000)
         žvižg = np.sin(t**2)
         plt.plot(t, žvižg, label='Žvižg')
         plt.xlim(1, 10)
         plt.title('Žvižg: $t^2$ (podpora za LaTeX: $\\sqrt{\\frac{a}{b}}$)')
         plt.legend();
```

[51] https://bokeh.pydata.org/en/latest/
[52] https://www.youtube.com/watch?v=FytuB8nFHPQ
[53] http://matplotlib.org/
[54] http://matplotlib.org/gallery.html#



Mimogrede, zakaj to imenujemo žvižg (oz. kvadraten žvižg)? Da dobimo odgovor, podatke predvajamo na zvočnik:

```
In [82]: from IPython.display import Audio, display
         display(Audio(data=žvižg, rate=44000))
```

```
<IPython.lib.display.Audio object>
```

Aktivirajmo sedaj interaktivni način (glejte tudi `%matplotlib`):

```
In [83]: %matplotlib notebook
```

```
In [84]: plt.plot([1,2,3], [2,4,5]);
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Zgornjo sliko lahko sedaj interaktivno klikamo in tudi dopolnjujemo s kodo:

```
In [85]: plt.title('Pozneje dodani naslov!')
         plt.xlabel('Čas [$t$]');
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

3.3.2 Interaktivna uporaba

Pri tem predmetu bomo večkrat uporabljali interaktinivnost pri delu z grafičnimi prikazi. V ta namen najprej uvozimo `interact` iz paketa `ipywidgets`:

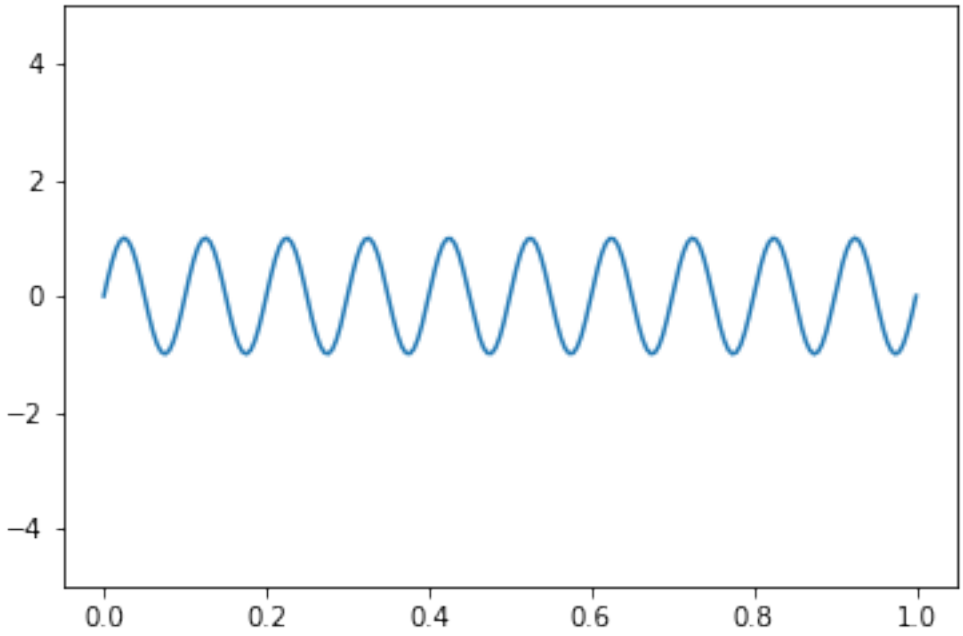
```
In [86]: from ipywidgets import interact
```

Potem definiramo sliko kot funkcijo z argumenti `amplituda`, `fr`, `faza` in `dušenje`:

```
In [87]: def slika(amplituda=1, fr=10, faza=0, dušenje=0.):
             t = np.linspace(0, 1, 200)
             f = amplituda * np.sin(2*np.pi*fr*t - faza) * np.exp(-dušenje*2*np.pi*fr*t)
             plt.plot(t, f)
             plt.ylim(-5, 5)
             plt.show()
```

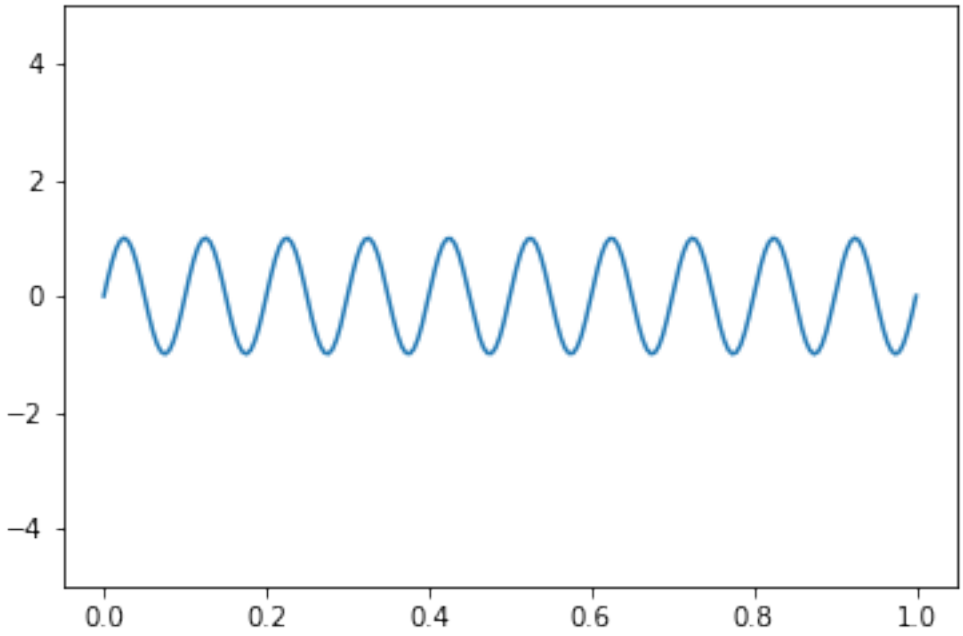
Gremo nazaj na medvrstično uporabo in kličemo funkcijo za izris slike (privzeti argumenti):

```
In [88]: %matplotlib inline
         slika()
```



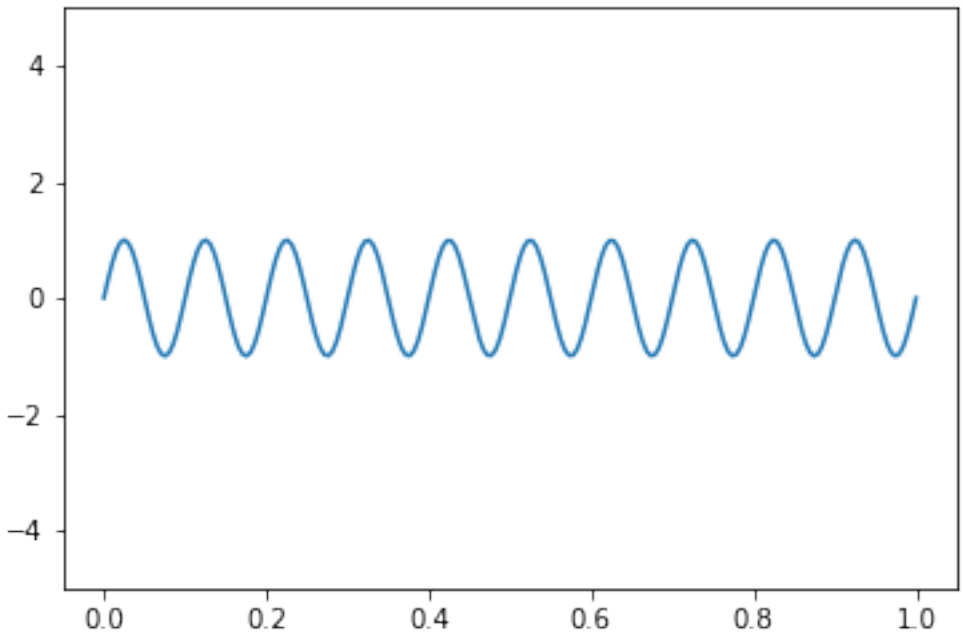
Če funkcijo za izris slike pošljemo v funkcijo `interact`, slednja poskrbi za interaktivne gumbe, s katerimi lahko spreminajmo parametre klicanja funkcije `slika`:

```
In [89]: interact(slika);
```



Razpon parametov lahko tudi sami definiramo:

```
In [90]: interact(slika, amplituda=(1, 5, 1), dušenje=(0, 1, 0.05), fr=(10, 100, 1), faza=(0, 2*np.pi,
```

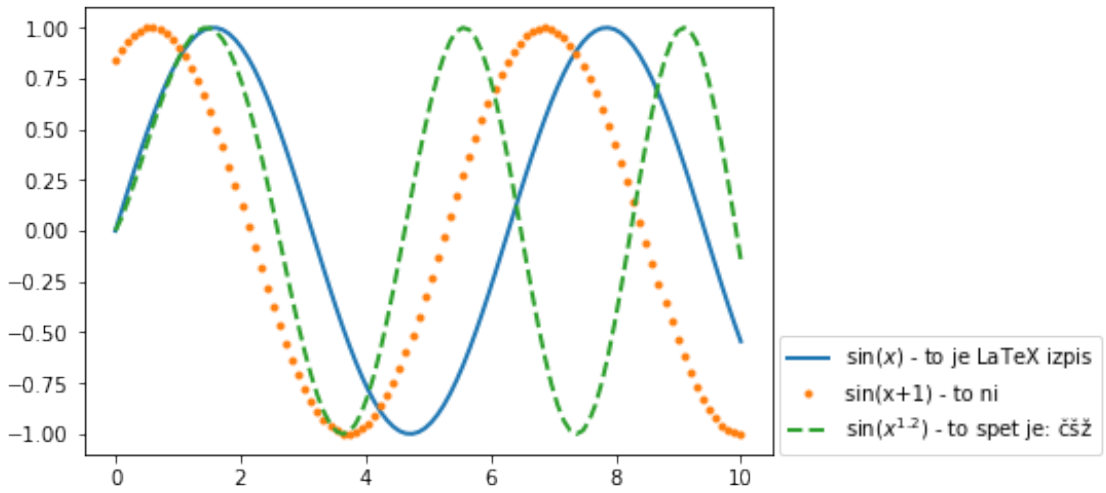


3.3.3 Napredna uporaba

Prikaz več funkcij

Poglejmo si preprost primer prikaza več funkcij:

```
In [91]: x = np.linspace(0, 10, 100)

         y1 = np.sin(x)
         y2 = np.sin(x+1)
         y3 = np.sin(x**1.2)

         plt.plot(x, y1, '-', label='$\sin(x)$ - to je LaTeX izpis', linewidth = 2);
         plt.plot(x, y2, '.', label='sin(x+1) - to ni', linewidth = 2);
         plt.plot(x, y3, '--', label='$\sin(x^{1.2})$ - to spet je: čšž', linewidth = 2);
         plt.legend(loc=(1.01,0));
         plt.savefig('data/prvi plot.pdf')
```



Prikaz več slik

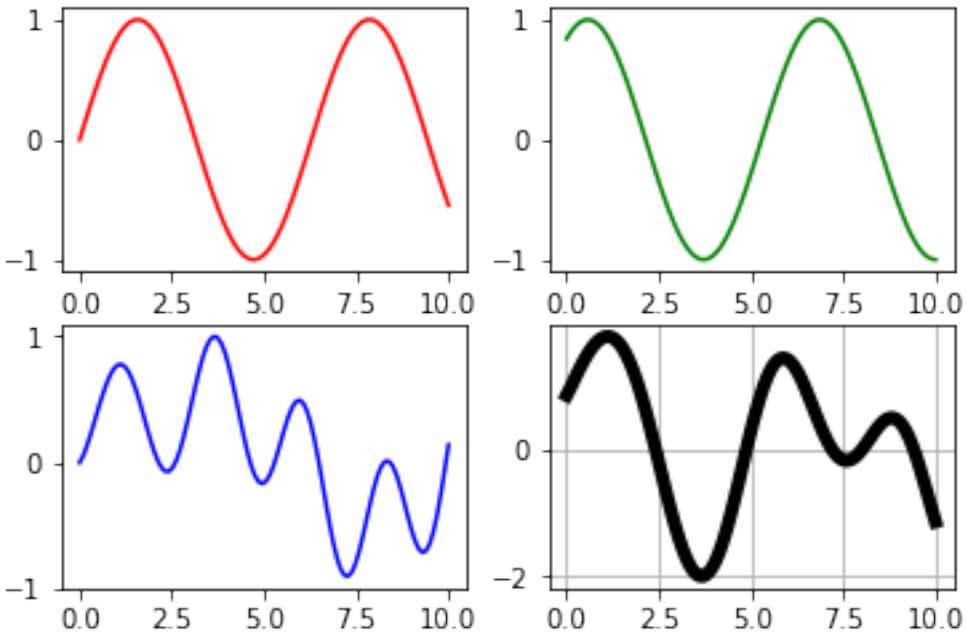
Več slik prikažemo s pomočjo metode `subplot`, ki definira mrežo in lego naslednjega izrisala.

```
plt.subplot(2, 2, 1)
plt.plot(x, y1, 'r')
plt.subplot(2, 2, 2)
plt.plot(x, y2, 'g')
```

V primeru zgoraj `plt.subplot(2, 2, 1)` pomeni: mreža naj bo `2 x 2`, riši v sliko 1 (levo zgoraj). Naslednjič se kliče `plt.subplot(2, 2, 2)` kar pomeni mreža `2 x 2`, riši v sliko 2 (desno zgoraj) in tako naprej. Opazimo, da se indeks slik začne z 1; lahko bi rekli, da gre za nekonsistentnost s Pythonom, razlog pa je v tem, da je bil `matplotlib` rojen v ideji, da bi uporabnikom Pythona uporabil čimbolj podoben način izrisa, kakor so ga poznali v Matlabu.

Delujoči primer je:

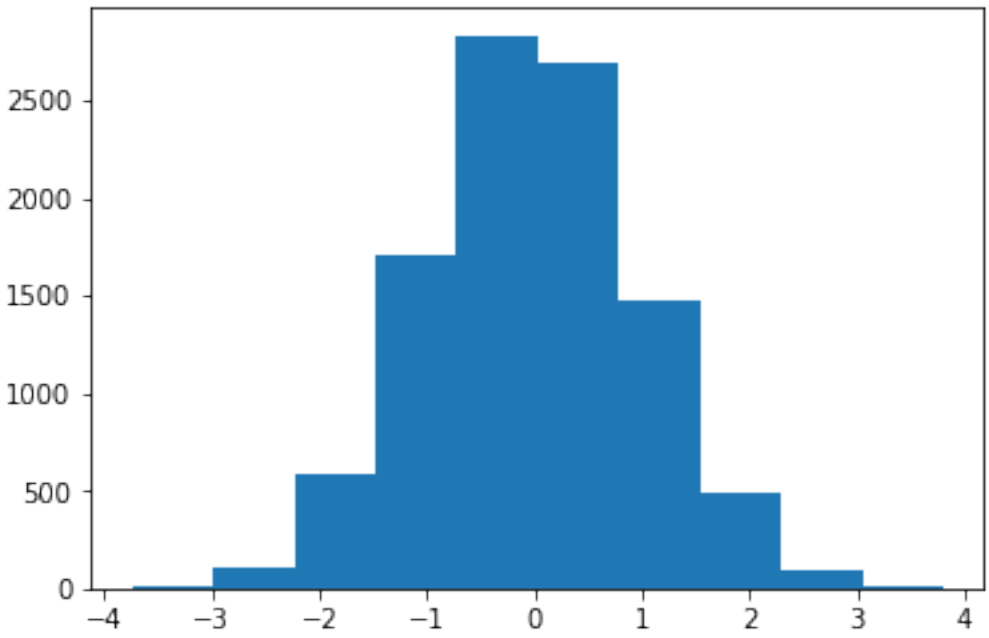
```
In [92]: plt.subplot(2, 2, 1)
         plt.plot(x, y1, 'r')
         plt.subplot(2, 2, 2)
         plt.plot(x, y2, 'g')
         plt.subplot(2, 2, 3)
         plt.plot(x, y2*y3, 'b')
         plt.subplot(2, 2, 4)
         plt.plot(x, y2+y3, 'k', linewidth=5);
         plt.grid()
```



Histogram

Generirajmo 10000 normalno porazdeljenih vzorcev in jih prikažimo v obliki histograma:

```
In [93]: np.random.seed(0)
         x = np.random.normal(size=10000)
         plt.hist(x);
```



3.3.4 Uporaba primerov iz `matplotlib.org`

Primere iz galerije[55] lahko uvozimo s pomočjo magične funkcije `%load`.

Poskusite:

- `%load http://matplotlib.org/mpl_examples/lines_bars_and_markers/fill_demo.py`
- `%load http://matplotlib.org/examples/widgets/slider_demo.py`

3.4 Nekaj vprašanj za razmislek!

1. Naredite slovar lokalno nameščenih modulov (uporabite izpeljevanje slovarjev, ključ naj bo ime modula, vrednost naj bo verzija).
2. S pomočjo slovarja iz prejšnje točke čimbolj preprosto preverite, ali so nameščeni sledeči moduli: ['numpy', 'scipy', 'matplotlib', 'pandas', 'pyjamas', 'openpyxl'].
3. Namestite poljubni modul iz https://pypi.python.org/pypi in ga preizkusite.
4. Pretvorite navaden Pythonov seznam v `numpy` numerično polje. Preverite tip enega in drugega.
5. Raziščite funkcije `np.ones`, `np.zeros_like`, `np.arange`, `np.linspace` (ali zadnja funkcija lahko vrne korak?).
6. Prikažite uporabo rezanja.
7. Prikažite razliko med vrstičnim in stolpičnim vektorjem. Prikažite tipične matematične operacije med vektorji in matrikami.
8. Ustvarite matriko ničel dimenzije 3 x 2 in drugi stolpec zapolnite z vektorjem enic.
9. Ustvarite enotsko matriko dimenzije 5 podatkovnega tipa `complex`.

[55] http://matplotlib.org/gallery.html

10. Ustvarite enotsko matriko dimenzije `N` in izračunajte vsoto poljubnga stolpca. Poskusite najti najhitrejši (vektoriziran) način in ga primerjajte s pristopom v zanki. Namig: np.sum().
11. V matriki iz prejšnje točke zamenjajte poljubna stolpca, nato še poljubni vrstici. Preverite hitrost vaše implementacije.
12. S pomočjo funkcije `np.random.rand` ustvarite dvorazsežno matriko poljubne dimenzije in najdite največjo in najmanjšo vrednost. Preverite možnost `axis` v funkciji `np.max` ali `np.min`.
13. V matriki iz prejšnje točke najdite indeks, kjer se nahaja največja vrednost.
14. Na primeru preprostega diagrama prikažite razliko med *inline* in *interaktivno* uporabo knjižnice `matplotlib`.
15. Na primeru preprostega diagrama prikažite uporabo vsaj 5 različnih tipov črte in 5 različnih barv.
16. Raziščite primere http://matplotlib.org/gallery.html. Za primer si podrobneje poglejte enega od zgledov na temo zaznamb *annotation*. Izbrani primer namestite na vaš računalnik.
17. Na primeru preprostega diagrama prikažite uporabo zaznamb.
18. Dodatno: Naredite preprosto animacijo.
19. Dodatno: Izrišite več krogov naključne lege in velikosti ter poljubne barve. Ob kliku se krogom naj spremeni barva.

3.5 Dodatno

3.5.1 `numba`

Paket `numba` se v zadnjem obdobju zelo razvija in lahko numerično izvajanje še dodatno pohitri (tudi v povezavi z grafičnimi karticami oz. GPU procesorji).

Za zgled tukaj uporabimo `jit` (just-in-time compilation[56]) iz paketa `numba`:

```
In [94]: from numba import jit
```

`jit` uporabimo kot dekorator[57] funkcije, ki jo želimo pohitriti:

```
In [95]: @jit
         def skalarni_produkt(a, b):
             c = 0
             for i in range(N):
                 c += a[i] * b[i]
             return c
```

Sedaj definirajmo vektorja:

```
In [96]: N = 1000
         a = np.arange(N)
         b = np.arange(N)
```

Preverimo hitrost `numpy` skalarnega produkta:

```
In [97]: %%timeit -n1000
         a @ b
```

```
5.54 µs ± 2.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

[56] http://numba.pydata.org/numba-doc/dev/reference/jit-compilation.html
[57] https://docs.python.org/3/glossary.html#term-decorator

Preverimo še hitrost `numba` pohitrene verzije. Prvi klic izvedemo, da se izvede kompilacija, potem merimo čas:

```
In [98]: skalarni_produkt(a, b)

Out[98]: 332833500

In [99]: %%timeit -n1000
         skalarni_produkt(a, b)

1.74 µs ± 344 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Vidimo, da smo še izboljšali hitrost!

3.5.2 `matplotlib`: animacije, povratni klic, XKCD stil

Z `matplotlib` lahko pripravimo tudi **animacije**. Dva primera lahko najdete tukaj:

- Preprosta animacija[58],
- Dvojno nihalo[59].

Več v dokumentaciji[60].

Slika s **povratnim klicem** (angl. *call back*):

- Risanje črte[61].

Pripravite lahko tudi *na roko* narisane slike (**XKCD stil**):

- XKCD stil[62].

3.5.3 Za najbolj zagrete

1. Naučite se še kaj novega na chrisalbon.com[63].

[58] `./moduli/matplotlib_animacija.py`
[59] `./moduli/animacija_dvojno_nihalo.py`
[60] `https://matplotlib.org/api/animation_api.html#animation`
[61] `./moduli/matplotlib_klik.py`
[62] `./moduli/xkcd_stil.py`
[63] `http://chrisalbon.com/`

Poglavje 4

Objektno programiranje, simbolno računanje

4.1 Objektno programiranje

Pri programiranju poznamo različne pristope, dokumetnacija Python-a (docs.python.org[1]) omenja npr:

1. **proceduralni**: seznam navodil, kaj je treba izvesti (npr.: `C, Pascal`)

- **deklarativni**: opišemo kaj želimo, programski jezik pa izvede (npr., `SQL`)
- **funkcijski**: programiranje temelji na funkcijah (npr.: `Haskell`)
- **objektni**: program temelji na objektih, ki imajo lastnosti, funkcije ... (npr.: `Java, Smalltalk`)

Python je objektno orientiran programski jezik, vendar pa nas ne sili v uporabo objektov v vseh primerih. Kot bomo videli pozneje, ima objektno programiranje veliko prednosti, vendar pa je lahko mnogokrat okorno in bi po nepotrebnem naredilo program kompleksen. Iz tega razloga se eksplicitnemu objektnemu programiranju izognemo, če se le da.

Objektno programiranje v Pythonu temelji na **razredih** (*class*), objekti so pa **instance** (*instance*) razreda. Pogledali si bomo zgolj nekatere osnove objektnega programiranja (da boste lažje razumeli kodo drugih avtorjev in jo prirejali svojim potrebam).

Razred definiramo z ukazom `class` (dokumentacija[2]):

```
class ImeRazreda:
    '''docstring'''
    [izraz 1]
    [izraz 2]
    .
    .
```

kjer ime razreda (torej `ImeRazreda`) po PEP8 pišemo z veliko začetnico. Če je ime sestavljeno iz več besed, vsako pišemo z veliko (t. i. principi *CamelCase*).

Poglejmo si primer:

[1] https://docs.python.org/howto/functional.html
[2] https://docs.python.org/tutorial/classes.html

```
In [1]: class Pravokotnik:
            """Razred za objekt pravokotnik"""

            def __init__(self, širina=1, višina=1): # to je konstruktor objekta. Se izvede, ko kličemo
                self.širina = širina
                self.višina = višina # višina je atribut objekta

            def površina(self):
                return self.širina * self.višina

            def set_širina(self, širina=1):
                self.širina = širina
```

Preden gremo v podrobnosti razumevanja kode, naredimo instanco razreda (torej objekt):

```
In [2]: moj_pravokotnik = Pravokotnik()
```

Funkcije definirane znotraj razreda poimenujemo **metode**, ko jih kličemo na objektih.

V zgornjem primeru metodo `površina` uporabimo tako:

```
In [3]: moj_pravokotnik.površina()

Out[3]: 1
```

Ime `self` je referenca na instanco razreda (objekt, ki bo ustvarjen). Imena znotraj razreda postanejo **atributi** objekta.

Primer atributa 'višina':

```
In [4]: moj_pravokotnik.višina

Out[4]: 1
```

Od kje pride rezultat `1`? Ko ustvarimo objekt, se najprej izvede inicializacijska funkcija `__init__()`, pri tem se kot argumenti funkcije `__init__` uporabijo argumenti, ki jih posredujemo v razred.

Primer:

```
In [5]: tvoj_pravokotnik = Pravokotnik(višina=5, širina=3)
        tvoj_pravokotnik.površina()

Out[5]: 15
```

Pripravili smo tudi metodo, ki spremeni atribut `širina`:

```
In [6]: moj_pravokotnik.set_širina(širina=100)
        moj_pravokotnik.površina()

Out[6]: 100
```

Atribute lahko spreminjamo tudi neposredno, vendar se temu (zaradi možnosti napake in napačne uporabe) ponavadi izogibamo.

Primer:

```
In [7]: moj_pravokotnik.višina = 3
        moj_pravokotnik.površina()
```

```
Out[7]: 300
```

4.1.1 Dedovanje

Pomembna lastnost razredov je dedovanje; samo ime pove bistvo: tako kot ljudje dedujemo od svojih staršev, podobno velja tudi za razrede. Vsak razred (`class`) tako lahko deduje lastnosti kakega drugega razreda (dokumentacija[3]). Lahko ima celo več staršev (v te podrobnosti tukaj ne bomo šli).

Sintaksa razreda, ki deduje, je:

```
class Otrok(Starš):
    [izraz]
    .
    .
```

Opomba: tudi, če razredu ne definiramo *starša*, deduje razred `class`.

Primer, ko novi razred `Kvadrat` podeduje obstoječega (`Pravokotnik`):

```
In [8]: class Kvadrat(Pravokotnik):
            "Razred kvadrat"

            def __init__(self, širina=1):
                # kličimo iniciacijo razreda Pravokotnik
                super().__init__(širina=širina, višina=širina)

            def set_širina(self, širina):
                self.širina = širina
                self.višina = širina
```

Poglejmo sedaj uporabo:

```
In [9]: moj_kvadrat = Kvadrat(širina=4)
```

Razred `Kvadrat` nima definicije metode za izračun površine, vendar pa jo je podedoval od razreda `Pravokotnik` in zato *ima metodo* za izračun površine:

```
In [10]: moj_kvadrat.površina()
```

```
Out[10]: 16
```

V kolikor spremenimo širino, se ustrezno spremeni površina:

```
In [11]: moj_kvadrat.set_širina(5)
         moj_kvadrat.površina()
```

```
Out[11]: 25
```

[3] https://docs.python.org/tutorial/classes.html#inheritance

Primer dedovanja razreda `list` (seznam)

Najprej pripravimo seznam:

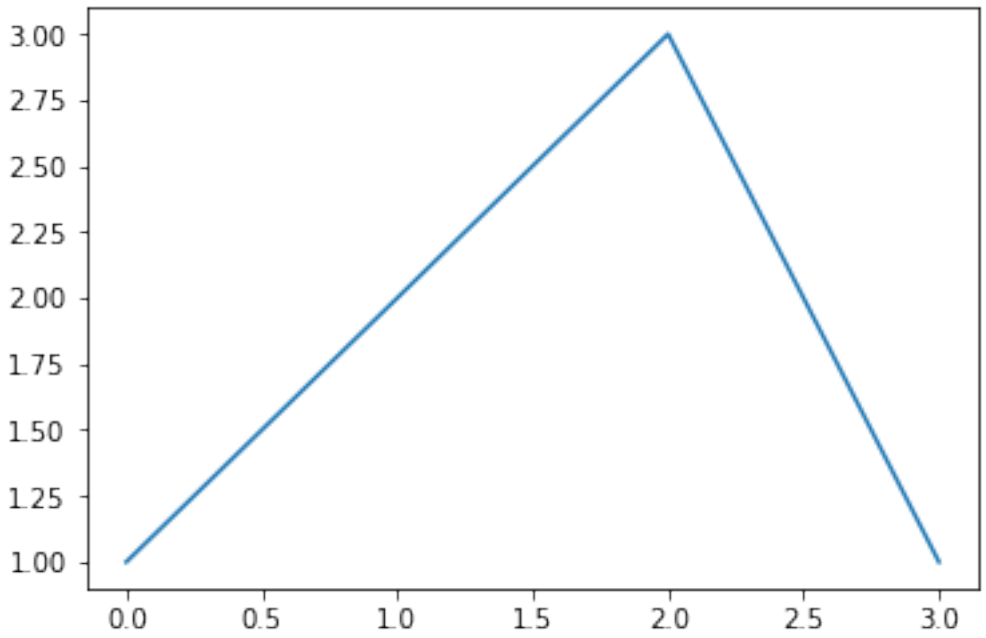
```
In [12]: seznam = list([1,2,3])
         seznam

Out[12]: [1, 2, 3]
```

Če želimo seznamu dodati vrednost, uporabimo metodo `append` (to je metoda, ki jo imajo objekti tipa `list`):

```
In [13]: seznam.append(1)
```

Nato seznam prikažemo (najprej uvozimo `matplotlib`):

```
In [14]: import matplotlib.pyplot as plt
         plt.plot(seznam)
         plt.show()
```



Če nekaj takega izvajamo pogosto, potem je bolje, da si pripravimo svoj razred `Seznam`, ki deduje od `list` in dodamo metodo za prikaz `nariši`:

```
In [15]: class Seznam(list):

             def nariši(self):
                 plt.plot(self, 'r.', label='Dolgo besedilo')
                 plt.legend()
                 plt.ylim(-5, 5)
                 plt.show()
```

Instanca objekta je:

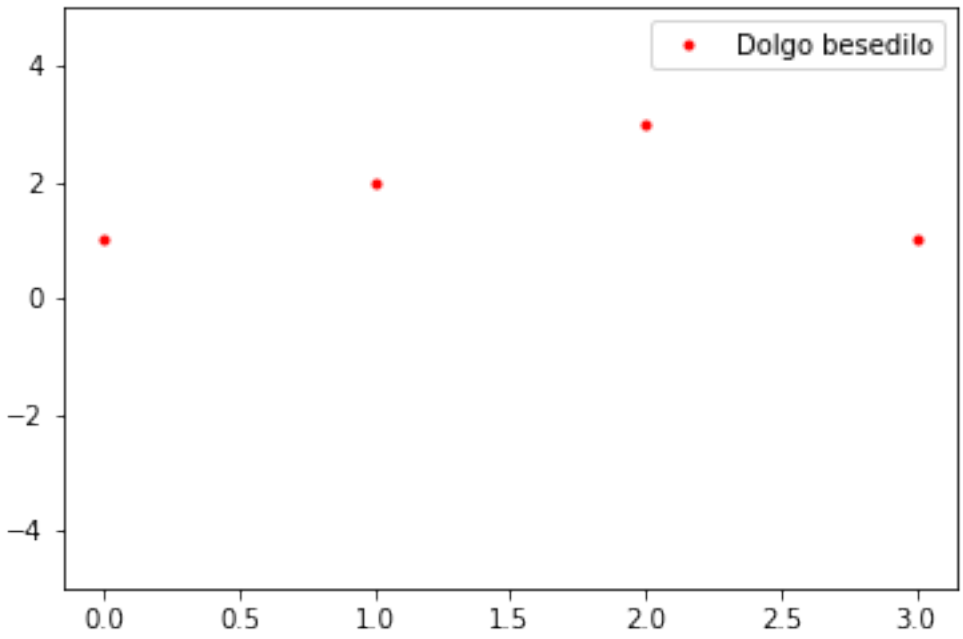
```
In [16]: moj_seznam = Seznam([1, 2, 3])
         type(moj_seznam)

Out[16]: __main__.Seznam

In [17]: moj_seznam

Out[17]: [1, 2, 3]
```

Čeprav nismo definirali metode `append`, jo je nov razred podedoval po razredu `list`:

```
In [18]: moj_seznam.append(1)
         moj_seznam

Out[18]: [1, 2, 3, 1]
```

Ima tudi metodo za izris:

```
In [19]: moj_seznam.nariši()
```



4.2 Simbolno računanje s `SymPy`

Termin **simbolno računanje** pomeni, da matematične izraze rešujemo strojno v obliki abstraktnih simbolov (in **ne numerično**). Strojno simbolno računanje nam pomaga kadar nas zanima **rezultat v simbolni obliki**

in so izrazi preobsežni za klasično reševanje na list in papir. K strojnemu reševanju se zatečemo tudi zaradi zmanjšanja možnosti napake (pri obsežnih izračunih se ljudje lahko zmotimo).

Simbolno računanje nikakor ni nadomestek numeričnih metod!

Pogledali si bomo nekatere osnove, nekateri priporočeni dodatni viri pa so:

- J.R. Johansson Scientific python lectures[4],
- SymPy[5] - uradna dokumentacija modula,
- Odličen članek nekaterih avtorjev SymPy: Meurer et. al, 2017[6].

`SymPy` je eden od sistemov za strojno algebro (CAS - *Computer Algebra Systems*), ki pa ima poleg zmogljivosti tudi to prednost, da je v celoti napisan v Pythonu (alternativni paket v Pythonu Sage[7] na primer ni v celoti napisan v Pythonu).

Nekatera namenska komercialna orodja:

- Mathematica[8]
- Maple[9]

Najprej uvozimo modul `SymPy`; tipično paket uvozimo kot `sym`:

```
In [20]: import sympy as sym
```

Opazimo lahko, da se `SymPy` uvaža tudi `from sympy import *`. Temu se praviloma izogibamo, saj tako s `SymPy` imeni po nepotrebnem zapolnimo osnovni imenski prostor programa. V slednjem primeru do funkcij paketa (npr. `sympy.Sum`) dostopamo neposredno (npr. `Sum`), kar je lahko privlačno, vendar nas začne motiti, ko dodamo še druge pakete (npr. `numpy`), kar lahko poleg zmede privede do tega, da se funkcije z enakimi imeni "povozijo".

Zato da dobimo lepo oblikovan LaTeX izpis, uporabimo:

```
In [21]: sym.init_printing()
```

4.2.1 Definiranje spremenljivk in numerični izračun

Spremenljivke definiramo takole:

```
In [22]: x, y, k = sym.symbols('x, y, k')
```

Preverimo lahko tip:

```
In [23]: type(x)

Out[23]: sympy.core.symbol.Symbol
```

Opazimo, da je spremenljivka `x` sedaj `Symbol` iz paketa `sympy`.

Sedaj lahko naredimo preprost izračun:

[4] http://github.com/jrjohansson/scientific-python-lectures
[5] http://sympy.org/en/index.html
[6] https://peerj.com/articles/cs-103/
[7] http://www.sagemath.org/
[8] http://www.wolfram.com/mathematica
[9] http://www.maplesoft.com/products/maple

```
In [24]: x**y + k
```

Out[24]:

$$k + x^y$$

Funkcijo lahko tudi poimenujemo; tukaj je primer, kjer uporabimo funkcijo sinus in konstanto π:

```
In [25]: f = sym.sin(1.2*sym.pi + x)**2
         f
```

Out[25]:

$$\sin^2(x + 1.2\pi)$$

Bralec se morebiti sprašuje, zakaj potrebujemo *novo* funkcijo `sympy.sin()`, saj imamo vendar že tisto iz paketa `numpy`! Razlog je v tem, da simbolni izračun potrebuje popolnoma drugačno obravnavo kakor numerični in zato je koda zadaj povsem drugačna.

Če želimo zapisati enačbo, torej da enačimo en izraz z drugim, to naredimo takole:

```
In [26]: enačba = sym.Eq(sym.sin(k*x),0.5)
         enačba
```

Out[26]:

$$\sin(kx) = 0.5$$

Pri definiranju spremenljivk lahko dodajamo predpostavke:

```
In [27]: x = sym.Symbol('x', positive=True)
```

```
In [28]: x.is_positive
```

```
Out[28]: True
```

Prdpostavke se potem upoštevajo pri izračunu. V splošnem vemo, da $\sqrt{(x^2)} \neq x$, če pa je x pozitiven, pa velja $\sqrt{(x^2)} = x$ in `sympy` glede na predpostavke izračuna pravilen rezultat:

```
In [29]: sym.sqrt(x**2)
```

Out[29]:

$$x$$

Z metodo `assumptions0` pogledamo predpostavke objekta:

```
In [30]: x.assumptions0
```

```
Out[30]: {'commutative': True,
          'complex': True,
          'hermitian': True,
          'imaginary': False,
          'negative': False,
          'nonnegative': True,
          'nonpositive': False,
          'nonzero': True,
          'positive': True,
          'real': True,
          'zero': False}
```

SymPy pozna tipe števil (dokumentacija[10]):

- `Real` realna števila,
- `Rational` racionalna števila,
- `Complex` kompleksna števila,
- `Integer` cela števila.

Ti tipi so pomembni, saj lahko vplivajo na način reševanja in na rešitev.

Racionalna števila

Zgoraj smo že definirali *realna števila*. Poglejmo na primeru sedaj *racionalna števila*:

```
In [31]: r1 = sym.Rational(4, 5)
         r2 = sym.Rational(5, 4)

In [32]: r1

Out[32]:
```

$$\frac{4}{5}$$

Nekateri izračuni:

```
In [33]: r1+r2

Out[33]:
```

$$\frac{41}{20}$$

```
In [34]: r1/r2

Out[34]:
```

$$\frac{16}{25}$$

[10] http://docs.sympy.org/latest/modules/core.html#module-sympy.core.numbers

Kompleksna število

Imaginarno število se zapiše z `I` (to je drugače kot pri `numpy`, kjer je imaginarni del definiran z `j`):

```
In [35]: 1+1*sym.I
```

Out[35]:

$$1 + i$$

```
In [36]: sym.I**2
```

Out[36]:

$$-1$$

Numerični izračun

Pri simbolnem izračunu najprej analitične izraze rešimo, poenostavimo itd., nato pa pogosto želimo tudi izračunati konkreten rezultat.

Poglejmo primer:

```
In [37]: x = sym.symbols('x')
         f = sym.exp(x**x**x + sym.pi)
         f
```

Out[37]:

$$e^{x^{x^x}+\pi}$$

Če želimo sedaj namesto x uporabiti vrednost, npr `0.5`, to naredimo z metodo `subs()` (dokumentacija[11]):

```
In [38]: f.subs(x, 0.5)
```

Out[38]:

$$1.84512555475729e^{\pi}$$

Zgoraj smo uporabili konstanto π (dokumentacija[12]); nekatere tipično uporabljene konstante so:

- `sympy.pi` za število π,
- `sympy.E` za naravno število e,
- `sympy.oo` za neskončnost.

Kot smo videli zgoraj, `subs()` naredi zamenjavo in potem poenostavitve, ki so očitne; števila π ni izračunal v racionalni obliki. To moramo eksplicitno zahtevati z metodo:

[11] http://docs.sympy.org/latest/tutorial/basic_operations.html#substitution
[12] http://docs.sympy.org/latest/modules/core.html?highlight=pi#sympy.core.numbers.Pi

- `evalf` (angl. *evaluate function, dokumentacija[13]) ali
- `N`,

ki imata argument `n` (število decimalnih mest).

Poglejmo primer:

```
In [39]: f.subs(x, 2).evalf(n=80)
```

```
Out[39]:
```

$$205630752.25597888373365145985724589469698354456454413064267071001422894803692 88$$

Podobno je z `N`:

```
In [40]: sym.N(f.subs(x, 2), n=80)
```

```
Out[40]:
```

$$205630752.25597888373365145985724589469698354456454413064267071001422894803692 88$$

Mimogrede smo pokazali, da pod pogojem, da v izrazu nimamo števil s plavajočo vejico, lahko rezultat prikažemo poljubno natančno (dokumentacija[14]).

V `subs` funkciji lahko uporabimo tudi slovar. Primer:

```
In [41]: x, y = sym.symbols('x, y')
         parametri = {x: 4, y: 10}
```

```
In [42]: (x + y).subs(parametri)
```

```
Out[42]:
```

$$14$$

ali seznam terk:

```
In [43]: (x + y).subs([(x, 4), (y, 10)])
```

```
Out[43]:
```

$$14$$

Podobno ima metoda `sympy.evalf()` argument `subs`, ki sprejme slovar zamenjav, primer:

```
In [44]: (y**x).evalf(subs=parametri)
```

```
Out[44]:
```

$$10000.0$$

Metoda `sympy.subs` pa lahko zamenja simbol (ali izraz) tudi z drugim izrazom:

[13] http://docs.sympy.org/latest/modules/evalf.html
[14] http://docs.sympy.org/latest/modules/evalf.html

```
In [45]: (x + y).subs(x, y + sym.oo)
```

Out[45]:

$$2y + \infty$$

4.2.2 SymPy in NumPy

Pogosto sympy povežemo z numpy. Za primer si poglejmo, kako bi izraz:

```
In [46]: x = sym.symbols('x')
         f = sym.sin(x) + sym.exp(x)
         f
```

Out[46]:

$$e^x + \sin(x)$$

numerično učinkovito izračunali pri tisoč vrednostih x.

Najprej uvozimo paket numpy:

```
In [47]: import numpy as np
```

Pripravimo numerično polje vrednosti:

```
In [48]: x_vec = np.linspace(0, 10, 1000, endpoint=False)
         x_vec[:10]
```

```
Out[48]: array([ 0.  ,  0.01,  0.02,  0.03,  0.04,  0.05,  0.06,  0.07,  0.08,  0.09])
```

Glede na zapisano zgoraj in predhodno znanje uporabimo izpeljevanje seznamov:

```
In [49]: y_vec = np.array([f.evalf(subs={x: vrednost}) for vrednost in x_vec])
```

Opazimo, da je to dolgotrajno, zato izmerimo potreben čas:

```
In [50]: %%timeit -n1
         y_vec = np.array([f.evalf(subs={x: vrednost}) for vrednost in x_vec])
```

```
337 ms ± 10.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Uporaba funkcije lambdify

Bistveno hitrejši način je uporaba pristopa lambdify, kjer se pripravi prevedena funkcija, optimirana za numerično izvajanje. Sintaksa funkcije sympy.lambdify() je (dokumentacija[15]):

```
sympy.lambdify(simboli, funkcija, modules=None)
```

[15] http://docs.sympy.org/latest/modules/utilities/lambdify.html#sympy.utilities.lambdify.lambdify

kjer so argumenti:

- `simboli` simboli uporabljeni v `funkcija`, ki se zamenjajo z numeričnimi vrednostmi,
- `funkcija` predstavlja `sympy` funkcijo,
- `modules` predstavlja, za kateri paket je prevedena oblika pripravljena. Če je `numpy` nameščen, je privzeto za ta modul.

Primer uporabe:

```
In [51]: f_hitra = sym.lambdify(x, f, modules='numpy')
         y_vec_hitra = f_hitra(x_vec)
```

Preverimo hitrost:

```
In [52]: %%timeit -n100
         f_hitra(x_vec)
```

```
34.3 μs ± 5.6 μs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Opazimo približno 10.000-kratno pohitritev!

Pogljemo še primer uporabe v primer funkcije več spremenljivk:

```
In [53]: f_hitra2 = sym.lambdify((x, y), (x + y + sym.pi)**2, 'numpy')
         x = np.linspace(0, 10, 10)
         y = x
         f_hitra2(x, y)
```

```
Out[53]: array([   9.8696044 ,   28.77051002,   57.54795885,   96.20195089,
                  144.73248614,  203.1395646 ,  271.42318627,  349.58335115,
                  437.62005924,  535.53331054])
```

4.2.3 Grafični prikaz

`SymPy` ima na `matplotlib` temelječ prikaz podatkov. Prikaz je sicer glede na `matplotlib` bolj omejen in ga uporabljamo za preproste prikaze. (dokumentacija[16]).

Pogledali si bomo preproste primere, ki se navezujejo na funkcijo `sympy.plotting.plot`; najprej uvozimo funkcijo:

```
In [54]: from sympy.plotting import plot
```

Sintaksa uporabe funkcije `sympy.plotting.plot()` (dokumentacija[17]) je:

```
plot(izraz, razpon, **kwargs)
```

kjer so argumenti:

- `izraz` je matematični izraz ali več izrazov,

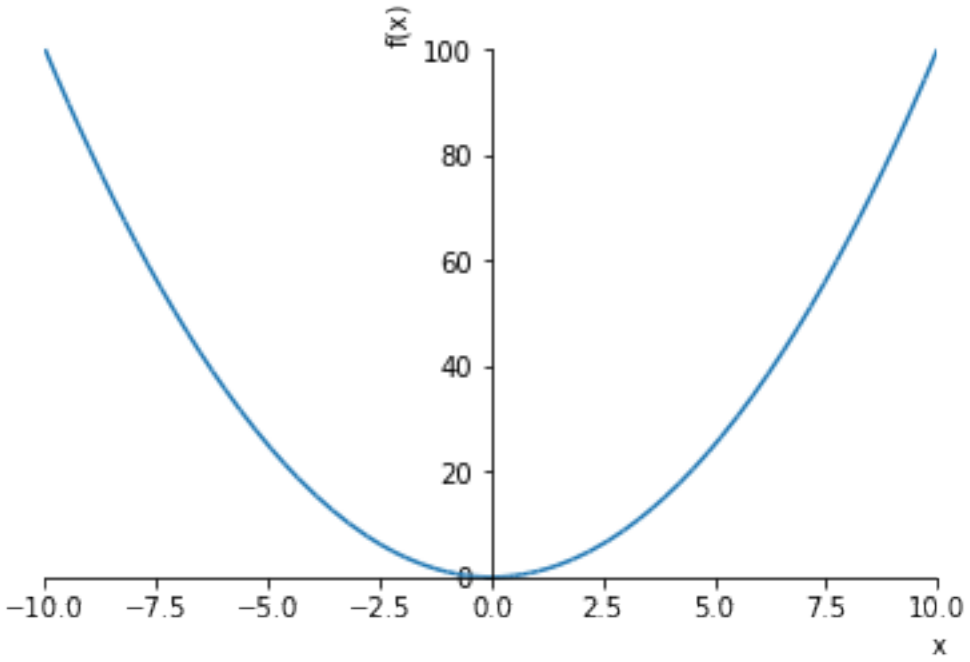
[16] http://docs.sympy.org/latest/modules/plotting.html
[17] http://docs.sympy.org/latest/modules/plotting.html#plotting-function-reference

- `razpon` je razpon prikaza (privzeti razpon je (-10, 10)),
- `**kwargs` so *keyword arguments*, torej slovar različnih možnosti.

Funkcija vrne instanco objekta `sympy.Plot()`.

Minimalni primer ene funkcije:

```
In [55]: x = sym.symbols('x')
         plot(x**2)
```




```
Out[55]: <sympy.plotting.plot.Plot at 0x1c6bb76e208>
```

Opomba: zadnja vrstica opozori na rezultat v obliki instance `Plot`; izpis objekta `<sympy.plotting.plot.Plot at 0x...>` skrijemo z uporabo podpičja:

```
plot(x**2);
```

slika pa se vseeno prikaže.

Nekateri pogosti argumenti so:

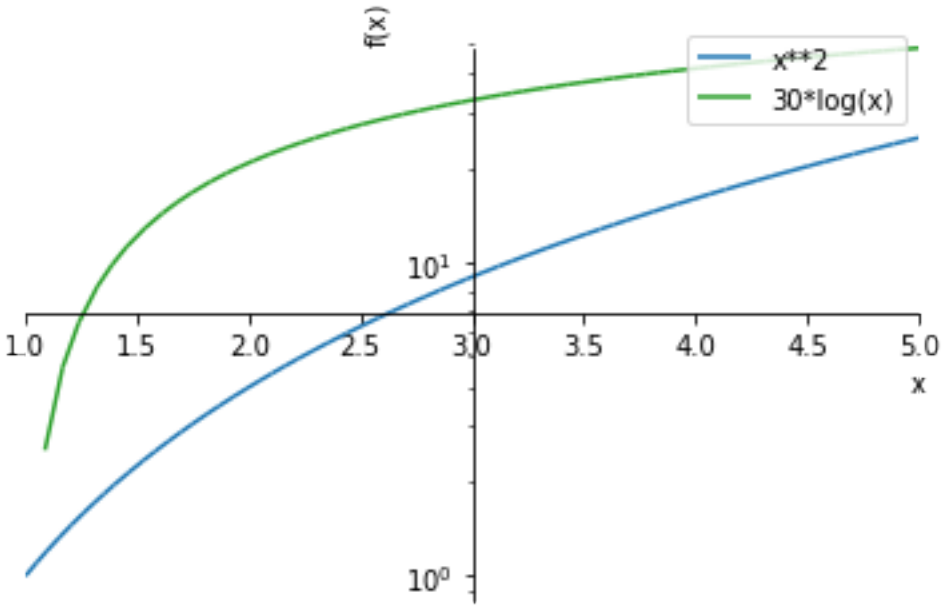
- `show` prikaže sliko (privzeto `True`),
- `line_color` barva izrisa,
- `xscale` in `yscale` način prikaza (možnosti: `linear` ali `log`),
- `xlim` in `ylim` omejitev prikaza za osi (terka dveh `(min, max)` vrednosti).

Pripravimo dve sliki, kjer bo y os logaritemska in bo razpon izrisa od 1 do 5:

```
In [56]: izris1 = plot(x**2, (x, 1, 5), show=False, legend=True, yscale='log', )
         izris2 = plot(30*sym.log(x), (x, 1, 5), show=False, line_color='C2', legend=True, yscale='log')
```

Sedaj prvo sliko razširimo z drugo in prikažemo rezultat:

```
In [57]: izris1.extend(izris2)
         izris1.show()
```



Parametrični izris

Podobno uporabljamo funkcijo `sympy.plotting.plot_parametric` za parametrični izris (dokumentacija[18]):

```
plot_parametric(izraz_x, izraz_y, range, **kwargs)
```

kjer sta nova argumenta:

- `izraz_x` in `izraz_y` definicije lege koordinate x in y,
- `**kwargs` je slovar možnosti.

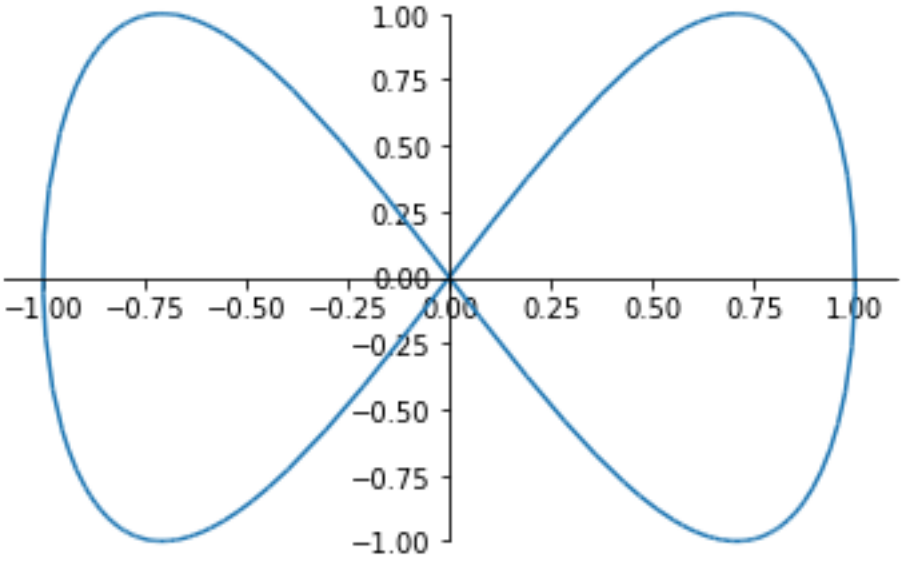
Uvozimo funkcijo:

```
In [58]: from sympy.plotting import plot_parametric
```

Prikažimo uporabo na primeru:

[18] http://docs.sympy.org/latest/modules/plotting.html#sympy.plotting.plot.plot_parametric

```
In [59]: plot_parametric(sym.sin(x), sym.sin(2*x), (x, 0, 2*sym.pi));
```



Izris v prostoru

Funkcija `sympy.plotting.plot3D` (dokumentacija[19]) za izris v prostoru ima sintakso:

```
plot3d(izraz, razpon_x, razpon_y, **kwargs)
```

kjer so argumenti:

- `izraz` definicija površine,
- `razpon_x` in `razpon_y` razpon koordinate `x` in `y`,
- `**kwargs` slovar možnosti.

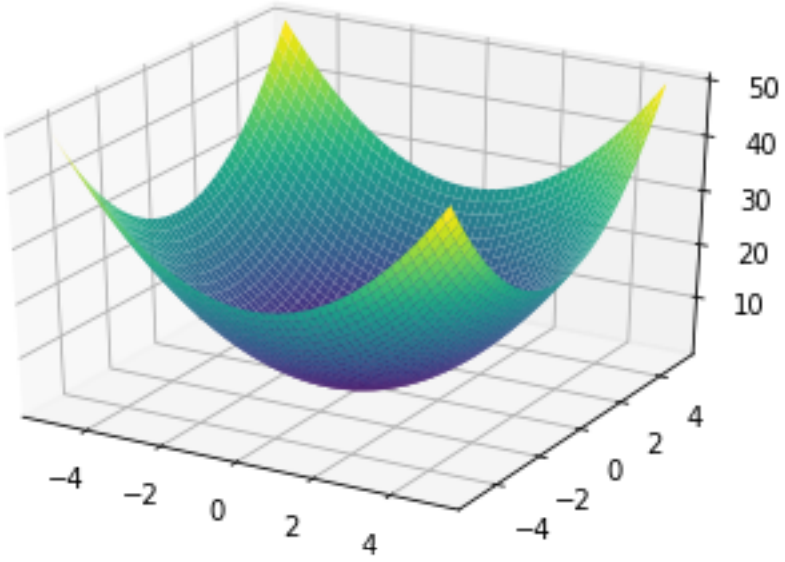
Uvozimo funkcijo:

```
In [60]: from sympy.plotting import plot3d
```

Prikažimo uporabo na primeru:

```
In [61]: x, y = sym.symbols('x y')
         plot3d(x**2 + y**2, (x, -5, 5), (y, -5, 5));
```

[19] http://docs.sympy.org/latest/modules/plotting.html#sympy.plotting.plot.plot3d



Za ostale prikaze glejte dokumentacijo[20].

4.2.4 Algebra

V tem poglavju si bomo pogledali nekatere osnove uporabe `SymPy` za algebrajske operacije.

Uporaba `expand` **in** `factor`

Definirajmo matematični izraz:

```
In [62]: x = sym.symbols('x')
         f = (x+1)*(x+2)*(x+3)
         f
```

Out[62]:

$$(x+1)\,(x+2)\,(x+3)$$

in ga sedaj **razčlenimo** (angl. *expand*, glejte dokumentacijo[21]):

```
In [63]: aa = sym.expand(f)
         aa
```

Out[63]:

$$x^3 + 6x^2 + 11x + 6$$

[20] http://docs.sympy.org/latest/modules/plotting.html#module-sympy.plotting.plot
[21] http://docs.sympy.org/latest/tutorial/simplification.html#expand

Če želimo pogledati koeficiente pred x, to naredimo z metodo `coeff()`:

```
In [64]: aa.coeff(x)
```

Out[64]:

$$11$$

Argumenti funkcije definirajo, kakšno razširitev želimo (dokumentacija[22]). Če želimo npr. trigonometrično razširitev, potem uporabimo `trig=True`:

```
In [65]: a, b = sym.symbols('a, b')
         sym.expand(sym.sin(a+b))
```

Out[65]:

$$\sin(a+b)$$

```
In [66]: sym.expand(sym.sin(a+b), trig=True)
```

Out[66]:

$$\sin(a)\cos(b)+\sin(b)\cos(a)$$

Obratna operacija od razčlenitve je **razcepitev** ali **razstavljanje** ali **faktorizacija** (angl. *factor*, dokumentacija[23]):

```
In [67]: sym.factor(x**3 + 6 * x**2 + 11*x + 6)
```

Out[67]:

$$(x+1)(x+2)(x+3)$$

Če nas zanimajo posamezni členi, potem to naredimo s funkcijo `sympy.factor_list`:

```
In [68]: sym.factor_list(x**3 + 6 * x**2 + 11*x + 6)
```

Out[68]:

$$(1,\quad [(x+1,\quad 1),\quad (x+2,\quad 1),\quad (x+3,\quad 1)])$$

Poenostavljanje izrazov s `simplify`

Funkcija `sympy.simplify()` (dokumentacija[24]) poskuša poenostaviti izraze v bolj preproste (npr. s krajšanjem spremenljivk).

Za posebne namene lahko poenostavimo tudi z:

[22] http://docs.sympy.org/latest/tutorial/simplification.html#expand
[23] http://docs.sympy.org/latest/tutorial/simplification.html#factor
[24] http://docs.sympy.org/latest/tutorial/simplification.html#simplify

- `sympy.trigsimp`[25],
- `sympy.powsimp`[26],
- `sympy.logcombine`[27].

Za več glejte dokumentacijo[28].

Primeri poenostavljanja:

```
In [69]: sym.simplify((x+1)*(x+1)*(x+3))
```

Out[69]:

$$(x+1)^2\,(x+3)$$

```
In [70]: sym.simplify(sym.sin(a)**2 + sym.cos(a)**2)
```

Out[70]:

$$1$$

```
In [71]: sym.simplify(sym.cos(x)/sym.sin(x))
```

Out[71]:

$$\frac{1}{\tan(x)}$$

4.2.5 Uporaba `apart` in `together`

Funkciji uporabljamo za delo z ulomki:

```
In [72]: f1 = 1/((1 + x) * (5 + x))
         f1
```

Out[72]:

$$\frac{1}{(x+1)\,(x+5)}$$

Razcep na parcialne ulomke (angl. *partial fraction decomposition*) izvedemo s funkcijo `sympy.apart()` (dokumentacija[29]):

```
In [73]: f2 = sym.apart(f1, x)
         f2
```

[25] http://docs.sympy.org/latest/tutorial/simplification.html#trigsimp
[26] http://docs.sympy.org/latest/tutorial/simplification.html#powsimp
[27] http://docs.sympy.org/latest/tutorial/simplification.html#logcombine
[28] http://docs.sympy.org/latest/tutorial/simplification.html
[29] http://docs.sympy.org/latest/tutorial/simplification.html#apart

Out[73]:

$$-\frac{1}{4\,(x+5)}+\frac{1}{4\,(x+1)}$$

in potem ponovno v obratni smeri s funkcijo `sympy.together()`:

```
In [74]: sym.together(f2)
```

Out[74]:

$$\frac{1}{(x+1)\,(x+5)}$$

V slednjem primeru pridemo do podobnega rezultata s `sympy.simplify()`:

```
In [75]: sym.simplify(f2)
```

Out[75]:

$$\frac{1}{(x+1)\,(x+5)}$$

4.2.6 Odvajanje

Odvajanje je načeloma relativno preprosta matematična operacija, ki jo izvedemo s funkcijo `sympy.diff()` (dokumentacija[30]):

Pripravimo primer:

```
In [76]: x, y, z = sym.symbols('x, y, z')
         f = sym.sin(x*y) + sym.cos(y*z)
         f
```

Out[76]:

$$\sin(xy)+\cos(yz)$$

Odvajajmo ga po x:

```
In [77]: sym.diff(f, x)
```

Out[77]:

$$y\cos(xy)$$

ali tudi

```
In [78]: f.diff(x)
```

[30] http://docs.sympy.org/latest/tutorial/calculus.html#derivatives

Out[78]:

$$y\cos(xy)$$

Odvode višjega reda definiramo tako:

```
In [79]: sym.diff(f, x, x, x)
```

Out[79]:

$$-y^3\cos(xy)$$

ali (isti rezultat malo drugače):

```
In [80]: sym.diff(f, x, 3)
```

Out[80]:

$$-y^3\cos(xy)$$

Odvod po več spremenljivkah $\frac{d^3 f}{dx\,dy^2}$ izvedemo takole:

```
In [81]: sym.diff(f, x, 1, y, 2)
```

Out[81]:

$$-x\left(xy\cos(xy)+2\sin(xy)\right)$$

4.2.7 Integriranje

Funkcija `integrate` lahko uporabimo za nedoločeno integriranje (dokumentacija[31]):

```
integrate(f, x)
```

ali za določeno integriranje:

```
integrate(f, (x, a, b))
```

kjer so argumenti:

- `f` funkcija, ki jo integriramo,
- `x` spremenljivka, po kateri integriramo,
- `a` in `b` meje integriranja.

Primer nedoločenega integriranja:

```
In [82]: x = sym.symbols('x')
         f = sym.sin(x*y) + sym.cos(y*z)
         sym.integrate(f, x)
```

[31] http://docs.sympy.org/latest/modules/integrals/integrals.html#module-sympy.integrals

Out[82]:

$$x\cos(yz) + \begin{cases} 0 & \text{for } y = 0 \\ -\frac{1}{y}\cos(xy) & \text{otherwise} \end{cases}$$

Opazimo, da sympy pravilno upošteva možnost, da je $y = 0$.

Še primer določenega integriranja:

```
In [83]: sym.integrate(f, (x, -1, 1))
```

Out[83]:

$$2\cos(yz)$$

Primer, ko so meje v neskončnosti (uporabimo konstanto za neskončnost sympy.oo):

```
In [84]: sym.integrate(sym.exp(-x**2), (x, -sym.oo, sym.oo))
```

Out[84]:

$$\sqrt{\pi}$$

4.2.8 Vsota in produkt vrste

Vsoto vrste definiramo s pomočju funkcije sympy.Sum() (dokumentacija[32]):

```
sympy.Sum(izraz, (spr, start, end))
```

kjer so argumenti:

- izraz izraz, katerega seštevamo,
- spr, start in end spremenljivka, ki naračša od start do end (end je vključen).

Primer vsote vrste:

```
In [85]: n = sym.Symbol('n')
         f = sym.Sum(1/x**n, (n, 1, sym.oo))
         f
```

Out[85]:

$$\sum_{n=1}^{\infty} x^{-n}$$

Šele ko uporabimo metodo doit_(), se izračun izvede:

```
In [86]: f.doit()
```

[32] http://docs.sympy.org/latest/modules/concrete.html#sympy.concrete.summations.Sum

Out[86]:

$$\begin{cases} \frac{1}{x\left(1-\frac{1}{x}\right)} & \text{for } \left|\frac{1}{x}\right| < 1 \\ \sum_{n=1}^{\infty} x^{-n} & \text{otherwise} \end{cases}$$

Poglejmo še številčni rezultat:

```
In [87]: f.subs({x: 5}).evalf()
```

Out[87]:

$$0.25$$

Produkt vrste definiramo podobno s funkcijo `sympy.Product` (dokumentacija[33]):

```
sympy.Product(izraz, (spr, start, end))
```

kjer so argumenti:

- `izraz` izraz, katerega množimo,
- `spr`, `start` in `end` spremenljivka, ki narašča od `start` do `end` (`end` je vključen).

Primer:

```
In [88]: f = sym.Product(1/n, (n, 1, 5))
         f
```

Out[88]:

$$\prod_{n=1}^{5} \frac{1}{n}$$

```
In [89]: f.doit()
```

Out[89]:

$$\frac{1}{120}$$

4.2.9 Limitni račun

Limite računamo s pomočjo funkcije `sympy.limit()` (dokumentacija[34]):

```
sympy.limit(f, x, x0)
```

kjer so argumenti:

[33] http://docs.sympy.org/latest/modules/concrete.html#sympy.concrete.products.Product
[34] http://docs.sympy.org/latest/tutorial/calculus.html#limits

- `f` izraz, katerega limito iščemo,
- `x` spremenljivka, ki limitira proti `x0`,
- `x0` limita.

Primer:

```
In [90]: x = sym.symbols('x')
         f = sym.sin(x)/x
         f
```

Out[90]:

$$\frac{1}{x}\sin(x)$$

```
In [91]: sym.limit(f, x, 0)
```

Out[91]:

$$1$$

Za primer si poglejmo uporabo limite na definiciji odvoda:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \lim_{h\to 0}\frac{f(x+h,y)-f(x,y)}{h}.$$

Pripravimo funkcijo `f` in njen odvod:

```
In [92]: x, y, z, h = sym.symbols('x, y, z, h')
         f = sym.sin(x*y) + sym.cos(y*z)
```

Odvod funkcije je:

```
In [93]: sym.diff(f, x)
```

Out[93]:

$$y\cos(xy)$$

Enak rezultat izračunamo tudi z uporabo limite:

```
In [94]: sym.limit((f.subs(x, x+h) - f)/h, h, 0)
```

Out[94]:

$$y\cos(xy)$$

4.2.10 Taylorjeve vrste

Taylorjeve vrste izračunamo s pomočjo funkcijo `sympy.series()` (dokumentacija[35]):

```
sympy.series(izraz, x=None, x0=0, n=6, dir='+')
```

kjer so argumenti:

- `izraz` izraz, katerega vrsto določamo,
- `x` neodvisna spremenljivka,
- `x0` vrednost, okoli katere določamo vrsto (privzeto 0),
- `n` red vrste (privzeto 6),
- `dir` smer razvoja vrste (+ ali -).

Primer:

```
In [95]: x = sym.symbols('x')
         sym.series(sym.exp(x), x) # privzete vrednosti x0=0, in n=6
```

Out[95]:

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \mathcal{O}\left(x^6\right)$$

Če želimo definirati drugo izhodišče (`x0=2`) in z več členi (`n=8`), to izvedemo takole:

```
In [96]: s1 = sym.series(sym.exp(x), x, x0=2, n=8)
         s1
```

Out[96]:

$$e^2 + (x-2)\,e^2 + \frac{e^2}{2}(x-2)^2 + \frac{e^2}{6}(x-2)^3 + \frac{e^2}{24}(x-2)^4 + \frac{e^2}{120}(x-2)^5 + \frac{e^2}{720}(x-2)^6 + \frac{e^2}{5040}(x-2)^7 + \mathcal{O}\left((x-2)^8; x \to 2\right)$$

Rezultat vključuje tudi red veljavnosti; na ta način lahko kontroliramo veljavnosti izvajanja ($\mathcal{O}$).
Primer:

```
In [97]: s1 = sym.cos(x).series(x, 0, 5)
         s1
```

Out[97]:

$$1 - \frac{x^2}{2} + \frac{x^4}{24} + \mathcal{O}\left(x^5\right)$$

```
In [98]: s2 = sym.sin(x).series(x, 0, 2)
         s2
```

[35] http://docs.sympy.org/latest/modules/series/series.html#id1

Out[98]:

$$x + \mathcal{O}\left(x^2\right)$$

Izračuna s1 in s2 imata različna reda veljavnosti, posledično je produkt:

```
In [99]: s1 * s2
```

Out[99]:

$$\left(x + \mathcal{O}\left(x^2\right)\right)\left(1 - \frac{x^2}{2} + \frac{x^4}{24} + \mathcal{O}\left(x^5\right)\right)$$

natančen samo do reda $\mathcal{O}(x^2)$, kar sympy ustrezno obravnava:

```
In [100]: s3 = sym.simplify(s1 * s2)
          s3
```

Out[100]:

$$x + \mathcal{O}\left(x^2\right)$$

Podatek o stopnji veljavnosti lahko odstranimo:

```
In [101]: s3.removeO()
```

Out[101]:

$$x$$

4.2.11 Linearna algebra

Matrike in vektorji

Matrike in vektorje definiramo s funkcjo Matrix. Če se pri numpy.array ni treba dosledno držati matematičnega zapisa vektorjev in matrik, je pri sympy to nujno.

Poglejmo si primer; najprej pripravimo spremenljivke:

```
In [102]: m11, m12, m21, m22 = sym.symbols('m11, m12, m21, m22')
          b1, b2 = sym.symbols('b1, b2')
```

Nato matriko in stolpični vektor:

```
In [103]: A = sym.Matrix([[m11, m12],[m21, m22]])
          A
```

Out[103]:

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix}$$

```
In [104]: b = sym.Matrix([[b1], [b2]])
          b
```

Out[104]:

$$\begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Sedaj si poglejmo nekatere tipične operacije; naprej množenje matrike in vektorja:

```
In [105]: A * b
```

Out[105]:

$$\begin{bmatrix} b_1 m_{11} + b_2 m_{12} \\ b_1 m_{21} + b_2 m_{22} \end{bmatrix}$$

Nato skalarni produkt dveh vektorjev (paziti moramo na transponiranje enega od vektorjev):

```
In [106]: b.T*b
```

Out[106]:

$$\begin{bmatrix} b_1^2 + b_2^2 \end{bmatrix}$$

Determinanta in inverzna matrika:

```
In [107]: A.det()
```

Out[107]:

$$m_{11} m_{22} - m_{12} m_{21}$$

```
In [108]: A.inv()
```

Out[108]:

$$\begin{bmatrix} \frac{m_{22}}{m_{11} m_{22} - m_{12} m_{21}} & -\frac{m_{12}}{m_{11} m_{22} - m_{12} m_{21}} \\ -\frac{m_{21}}{m_{11} m_{22} - m_{12} m_{21}} & \frac{m_{11}}{m_{11} m_{22} - m_{12} m_{21}} \end{bmatrix}$$

Množenje in potenca matrike:

```
In [109]: A*A
```

Out[109]:

$$\begin{bmatrix} m_{11}^2 + m_{12} m_{21} & m_{11} m_{12} + m_{12} m_{22} \\ m_{11} m_{21} + m_{21} m_{22} & m_{12} m_{21} + m_{22}^2 \end{bmatrix}$$

```
In [110]: A**2
```

Out[110]:

$$\begin{bmatrix} m_{11}^2 + m_{12} m_{21} & m_{11} m_{12} + m_{12} m_{22} \\ m_{11} m_{21} + m_{21} m_{22} & m_{12} m_{21} + m_{22}^2 \end{bmatrix}$$

4.2.12 Reševanje enačb

Enačbe in sistem enačb rešujemo s funkcijo `sympy.solve()` (dokumentacija[36]). Podprto je reševanje sledečih enačb:

- polinomske enačbe,
- transcendentne enačbe,
- odsekovno definirane enačbe kot kombinacija zgornjih dveh tipov,
- sistem linearnih in polinomskih enačb,
- sistem enačb z neenakostmi.

Sintaksta je:

```
sympy.solve(f, *symbols, **flags)
```

kjer so argumenti:

- `f` izraz ali seznam izrazov,
- `*symbols` simbol ali seznam simbolov, katere želimo določiti,
- `**flags` slovar možnosti.

Poglejmo primer:

```
In [111]: x = sym.symbols('x')
          f = sym.sin(x)
          en = sym.Eq(f, 1/2)
          en
```

Out[111]:

$$\sin(x) = 0.5$$

```
In [112]: sym.solve(en, x)
```

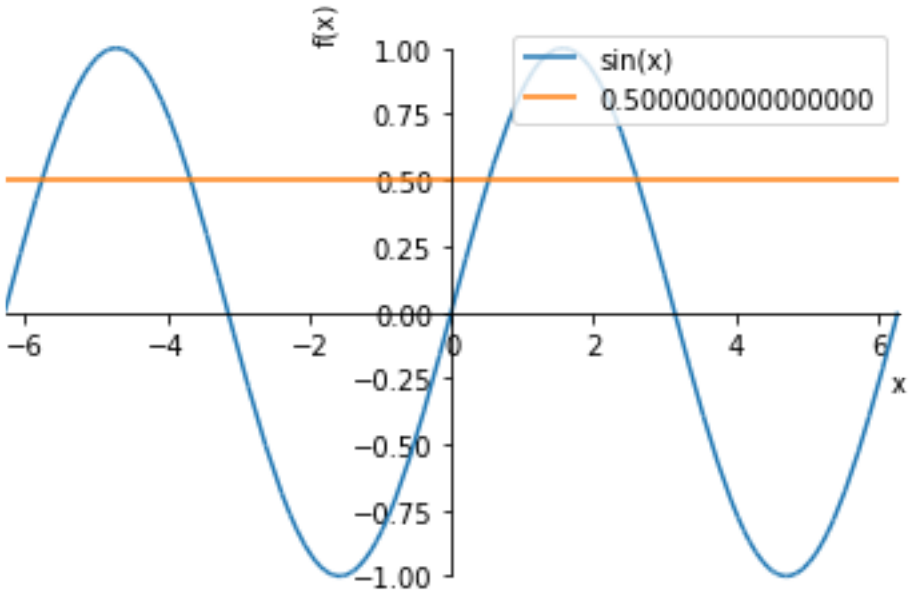
Out[112]:

$$[0.523598775598299, \quad 2.61799387799149]$$

Prikažimo rešitev (opazimo, da smo našli samo dve od neskončno rešitev):

```
In [113]: p1 = sym.plotting.plot(sym.sin(x), (x, -2*sym.pi, 2*sym.pi), line_color='C0', show=False, lege
          p2 = sym.plotting.plot(0.5, (x, -2*sym.pi, 2*sym.pi), line_color='C1', show=False, legend=True
          p1.extend(p2)
          p1.show()
```

[36] http://docs.sympy.org/latest/modules/solvers/solvers.html#algebraic-equations



Kvadratna enačba:

```
In [114]: a, b, c, x = sym.symbols('a, b, c, x')
          sym.solve(a*x**2 + b*x + c, x)
```

Out[114]:

$$\left[\frac{1}{2a}\left(-b+\sqrt{-4ac+b^2}\right),\quad -\frac{1}{2a}\left(b+\sqrt{-4ac+b^2}\right)\right]$$

Sistem enačb:

```
In [115]: x, y = sym.symbols('x y')
          sym.solve([x + y - 1, x - y - 1], [x, y])
```

Out[115]:

$$\{x:1,\quad y:0\}$$

Za nelinearne sistema pa lahko uporabimo tudi numerično reševanje s funkcijo `sympy.nsolve()` (dokumentaija[37]):

```
sympy.nsolve(f, [args,] x0, modules=['mpmath'], **kwargs)
```

kjer so argumenti:

- `f` enačba ali sistem enačb, ki ga rešujemo,

[37] http://docs.sympy.org/latest/modules/solvers/solvers.html#sympy.solvers.solvers.nsolve

- `args` spremenljivke (opcijsko),
- `x0` začetni približek (skalar ali vektor),
- `modules` paket, ki se uporabi za izračun numerične vrednosti (enakalogika kot pri funkciji `lambdify`, privzet je paket `mpmath`),
- `**kwargs` slovar opcij.

Poglejmo primer od zgoraj:

```
In [116]: x = sym.symbols('x')
          eq = sym.Eq(sym.sin(x), 0.5)
          sol = sym.nsolve(en, x, 3)
          sol
```

Out[116]:

$$2.61799387799149$$

4.2.13 Reševanje diferencialnih enačb

Diferencialne enačbe in sisteme diferencialnih enačb rešujemo s funkcijo `sympy.dsolve()` (dokumentacija[38]):

```
sympy.dsolve(eq, func=None, hint='default', simplify=True,
             ics=None, xi=None, eta=None, x0=0, n=6, **kwargs)
```

kjer so izbrani argumenti:

- `eq` differencialna enačba ali sistem diferencialnih enačb,
- `func` rešitev, ki jo iščemo.

Poglejmo si primer mase m, ki drsi po površini s koeficientom trenja μ; začetna hitrost je v_0, pomik $x_0 = 0$.
Definirajmo simbole:

```
In [117]: x = sym.symbols('x')
          t, m, mu, g, v0, x0 = sym.symbols('t m mu g v0 x0', real=True, positive=True)
```

Definirajmo diferencialno enačbo:

```
In [118]: eq = sym.Eq(m*x(t).diff(t,2), -mu*g*m)
          eq
```

Out[118]:

$$m\frac{d^2}{dt^2}x(t) = -gm\mu$$

Poglejmo lastnosti diferencialne enačbe:

```
In [119]: sym.ode_order(eq, x(t))
```

[38] http://docs.sympy.org/latest/modules/solvers/ode.html#dsolve

Out[119]:

$$2$$

```
In [120]: sym.classify_ode(eq)
```

```
Out[120]: ('nth_linear_constant_coeff_undetermined_coefficients',
           'nth_linear_constant_coeff_variation_of_parameters',
           'nth_linear_constant_coeff_variation_of_parameters_Integral')
```

Rešimo jo:

```
In [121]: rešitev = sym.dsolve(eq, x(t))
          rešitev
```

Out[121]:

$$x(t) = C_1 + C_2 t - \frac{g\mu}{2} t^2$$

Pripravimo funkcijo pomika:

```
In [122]: x_r = rešitev.args[1]
          x_r
```

Out[122]:

$$C_1 + C_2 t - \frac{g\mu}{2} t^2$$

Ker je pri času $t = 0$ pomik nič, velja:

```
In [123]: x_r.subs(t,0).subs(x(0),0)
```

Out[123]:

$$C_1$$

torej določimo konstanto `C1`

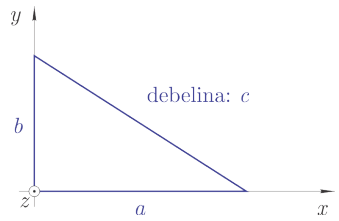
```
In [124]: C1 = sym.solve(x_r.subs(t,0).subs(x(0),0), 'C1')[0]
          C1
```

Out[124]:

$$0$$

Podobno je pri $t = 0$ hitrost enaka $v0$, torej določimo še konstanto `C2`:

```
In [125]: C2 = sym.solve(sym.Eq(x_r.diff(t).subs(t,0),v0), 'C2')[0]
          C2
```



Out[125]:

$$v_0$$

Ko sta določeni konstanti `C1` in `C2`, je enolično določena tudi funkcija lege mase:

In [126]:
```
x_r = x_r.subs('C1', C1).subs('C2', C2)
x_r
```

Out[126]:

$$-\frac{g\mu}{2}t^2 + tv_0$$

4.3 Nekaj vprašanj za razmislek!

1. Pojasnite na primeru *proceduralno* in *funkcijsko* programiranje.

- Definirajte preprost objekt, naredite nekaj funkcij temu objektu.
- Definirajte objekt, ki pri kreiranju instance zahteva zgolj celoštevilsko vrednost(npr.: dolžino seznama, ki jo bomo uporabili pri naslednji točki).
- Objektu iz prejšnje točke naj pri inicializaciji argumentu `data` priredi naključni seznam ustrezne dolžine (glejte funkcijo `np.random.rand`).
- Objektu iz prejšnje točke dodajte metodo za zapis vrednosti v datoteko s pomočjo funkcije `np.savetxt`.
- Enako kot pri prejšnji točki, vendar naj se podatki shranijo v binarni obliki s pomočjo modula `pickle`.
- Dodajte metodo za branje iz datoteke (s pomočjo `np.genfromtxt`).
- Uvozite ves naslovni prostor iz `SymPy`. Nastavite lep izpis rezultatov.
- Za trikotnik na sliki definirajte funkcijo za izračun površine in volumna.

- Izračunajte številčne vrednosti (podatki naj bodo definirani v slovarju in si jih izmislite).
- Izračunajte statični moment ploskve $S_{xx} = \int_A y\,dA = \int_0^b y\,x(y)\,dy$, kjer je $x(y) = a - a\,y/b$.
- Izračunajte vztrajnostni moment ploskve $I_{xx} = \int_A y^2\,dA$, $dA = x(y)\cdot dy$.
- Prikažite I_{xx} v odvisnosti od parametra b (a definirajte poljubno).
- Nedoločeno in določeno (v mejah od 0 do τ) integrirajte izraz: $\sin(5+t) + e^t$.
- Z odvajanjem pokažite pravilnost nedoločenega integrala iz predhodnega koraka.
- Za kotaleči valj (polmer r, masa m) povežite translatorno x prostost z rotacijsko φ. Pozneje boste vse izrazili s slednjo. Namig: Dolžina loka kroga ustreza zmnožku polmera r in kota φ [rad].
- Določite translatorno kinetično energijo težišča (definirajte s hitrostjo $\dot{x}$, zaradi predhodne povezave pa bi naj bil rezultat s $\dot{\varphi}$). $E_k = \frac{1}{2}\,m\,v^2$.

- Določite še masni vztrajnostni moment valja in rotacijsko kinetično energijo. Obe kinetični energiji seštejte in izraz poenostavite (če je potrebno). $J_v = \frac{1}{2}\, m\, r^2$ $E_{k,r} = \frac{1}{2}\, J_v \left[\frac{d}{dt}\varphi(t)\right]^2$
- Če na valj deluje moment $-M$, definirajte mehansko energijo: $E_m = -M\,\varphi$ in določite gibalno enačbo iz spremembe mehanske energije: $\frac{dE_m}{dt} = \frac{dE_k}{dt}$.
- Nadaljujete na predhodni enačbi: poiščite `sympy` funkcijo `replace` in ugotovite razliko s `subs`. Poskusite s pomočjo `replace` $\dot{\varphi}$ na obeh straneh enačbe spremeniti v 1.
- Najdite rešitev za predhodno pridobljeno diferencialno enačbo.
- Izmislite si začetne pogoje in jih uporabite na predhodno rešeni diferencialni enačbi. Izmislite si še preostale podatke ter prikažite rezultat.
- Določite čas, ko je zasuk φ spet enak začetnemu (če ste predpostavili začetni zasuk nič, potem torej iščete $\varphi = 0$. Določite tudi čas, ko je kotna hitrost $\dot{\varphi}$ enaka nič.

4.4 Dodatno

4.4.1 sympy.mechanics

`sympy` ima vgrajeno podporo za klasično mehaniko (dokumentacija[39]). Celovit tutorial je bil prikazan na znanstveni konferenci SciPy 2016[40].

[39] http://docs.sympy.org/latest/modules/physics/mechanics/index.html#classical-mechanics
[40] https://www.youtube.com/watch?v=r4piIKV4sDw

Poglavje 5

Uvod v numerične metode in sistemi linearnih enačb (1)

5.1 Uvod v numerične metode

Kadar želimo simulirati izbrani fizikalni proces, ponavadi postopamo takole:

1. postavimo matematični model*,

- izberemo numerično metodo in njene parametre,
- pripravimo program (pomagamo si z vgrajenimi funkcijami),
- izvedemo izračun, rezultate analiziramo in vrednotimo.

* Če lahko matematični model rešimo analitično, numerično reševanje ni potrebno.

Matematični model poskušamo rešiti analitično, saj taka rešitev ni obremenjena z napakami. Iz tega razloga se v okviru matematike učimo reševanja sistema enačb, integriranja, odvajanja in podobno. Bistvo **numeričnih metod** je, da matematične modele rešujemo **numerično**, torej na podlagi **diskretnih vrednosti**. Kakor bomo spoznali pozneje, nam numerični pristop v primerjavi z analitičnim omogoča reševanje bistveno obsežnejših in kompleksnejših problemov.

5.1.1 Zaokrožitvena napaka

V nadaljevanju si bomo pogledali nekatere omejitve in izzive numeričnega pristopa. Prva omejitev je, da so v računalniku realne vrednosti vedno zapisane s končno natančnostjo. V Pythonu se števila pogosto zapišejo v dvojni natančnosti s približno 15 signifikantnimi števkami.

Število z dvojno natančnostjo se v Pythonu imenuje **float64** in je zapisano v spomin v binarni obliki v 64 bitih (11 bitov eksponent in 53 bitov mantisa (1 bit za predznak)). Ker je mantisa definirana na podlagi 52 binarnih števk, se lahko pojavi pri njegovem zapisu *relativna napaka* največ $\epsilon \approx 2.2 \cdot 10^{-16}$. Ta napaka se imenuje **osnovna zaokrožitvena napaka** in se lahko pojavi pri vsakem vmesnem izračunu!

Če je korakov veliko, lahko napaka zelo naraste in zato je pomembno, da je njen vpliv na rezultat čim manjši!

Spodaj je primer podrobnejših informacij za tip podatkov z dvojno natančnostjo (`float`); pri tem si pomagamo z vgrajenim modulom `sys` za klic parametrov in funkcij python sistema (dokumentacija[1]):

[1] https://docs.python.org/3/library/sys.html

```
In [1]: import sys
        sys.float_info.epsilon
        #sys.float_info #preverite tudi širši izpis!

Out[1]: 2.220446049250313e-16
```

Poleg števila z dvojno natančnostjo se uporabljajo drugi tipi podatkov; dober pregled različnih tipov je prikazan v okviru numpy[2] in python[3] dokumentacije.

Tukaj si poglejmo primer tipa `int8`, kar pomeni celo število zapisano z 8 biti (8 bit = 1 byte). Z njim lahko v dvojiškem sistemu zapišemo cela števila od -128 do +127:

```
In [2]: import numpy as np
        število = np.int8(90) # poskušite še števila: -128 in nato 127, 128, 129. Kaj se dogaja?
        f'Število {število} tipa {type(število)} zapisano v binari obliki:{število:8b}'

Out[2]: "Število 90 tipa <class 'numpy.int8'> zapisano v binari obliki: 1011010"
```

5.1.2 Napaka metode

Poleg zaokrožitvene napake pa se pogosto srečamo tudi z **napako metode** ali **napako metode**, ki jo naredimo takrat, ko natančen analitični postopek reševanja matematičnega modela zamenjamo s približnim numeričnim.

Pomembna lastnost numeričnih algoritmov je **stabilnost**. To pomeni, da majhna sprememba vhodnih podatkov povzroči majhno spremembo rezultatov. Če se ob majhni spremembi na vhodu rezultati zelo spremenijo, pravimo, da je **algoritem nestabilen**. V praksi torej uporabljamo stabilne algoritme; bomo pa pozneje spoznali, da je stabilnost lahko pogojena tudi z vhodnimi podatki!

Poznamo pa tudi nestabilnost matematičnega modela/naloge/enačbe; v tem primeru govorimo o **slabi pogojenosti**.

Med izvajanjem numeričnega izračuna se napake lahko širijo. Posledično je rezultat operacije manj natančen (ima manj zanesljivih števk), kakor pa je zanesljivost podatkov izračuna.

Poglejmo si sedaj splošen pristop k oceni napake. Točno vrednost označimo z r, približek z a_1; velja $r = a_1 + e_1$, kjer je e_1 napaka. Če z numeričnim algoritmom izračunamo bistveno boljši približek a_2, velja $r = a_2 + e_2$.

Ker velja $a_1 + e_1 = a_2 + e_2$, lahko ob predpostavki $|e_1| >> |e_2|$ in $|e_2| \approx 0$ izpeljemo $a_2 - a_1 = e_1 - e_2 \approx e_1$.

$|a_1 - a_2|$ je torej pesimistična ocena absolutne napake,

$\left|\frac{a_1 - a_2}{a_2}\right|$ pa ocena relativne napake.

5.2 Uvod v sisteme linearnih enačb

Pod zgornjim naslovom razumemo sistem m linearnih enačb ($E_i, i = 0, 1, \ldots, m-1$) z n neznankami ($x_j, j = 0, 1, \ldots, n-1$):

$$\begin{array}{rccccccccc}
E_0: & A_{0,0}\,x_0 & + & A_{0,1}\,x_1 & + & \ldots & + & A_{0,n-1}\,x_{n-1} & = & b_0 \\
E_1: & A_{1,0}\,x_0 & + & A_{1,1}\,x_1 & + & \ldots & + & A_{2,n-1}\,x_{n-1} & = & b_1 \\
\vdots & & & & & \vdots & & & & \\
E_{m-1}: & A_{m-1,0}\,x_0 & + & A_{m-1,2}\,x_1 & + & \ldots & + & A_{m-1,n}\,x_{n-1} & = & b_{m-1}.
\end{array}$$

[2] https://docs.scipy.org/doc/numpy-1.10.1/user/basics.types.html
[3] https://docs.python.org/3/library/stdtypes.html

Koeficienti $A_{i,j}$ in b_i so znana števila.

V kolikor je desna stran enaka nič, torej $b_i = 0$, imenujemo sistem **homogenem**, sicer je **nehomogenem**.

Sistem enačb lahko zapišemo tudi v matrični obliki:

$$\mathbf{A}\,\mathbf{x} = \mathbf{b}$$

kjer sta **A** in **b** znana matrika in vektor, vektor **x** pa ni znan. Matriko **A** imenujemo **matrika koeficientov**, vektor **b** **vektor konstant** (tudi: vektor prostih členov ali vektor stolpec desnih strani) in **x** **vektor neznank**. Če matriki **A** dodamo kot stolpec vektor **b**, dobimo t. i. **razširjeno matriko** in jo označimo $[\mathbf{A}|\mathbf{b}]$.

Opomba glede zapisa:

- skalarne spremenljivke pišemo poševno, npr.: a, A,
- vektorske spremenljivke pišemo z majhno črko poudarjeno, npr.: **a**,
- matrične spremenljivke pišemo z veliko črko poudarjeno, npr.: **A**.

5.2.1 O rešitvi sistema linearnih enačb

Če nad sistemom linearnih enačb izvajamo **elementarne operacije**:

- množenje poljubne enačbe s konstanto (ki je različna od nič),
- spreminjanje vrstnega reda enačb,
- prištevanje ene enačbe (pomnožene s konstanto) drugi enačbi,

rešitve sistema ne spremenimo in dobimo ekvivalentni sistem enačb.

S pomočjo elementarnih operacij nad vrsticami matrike **A** jo lahko preoblikujemo v t. i. **vrstično kanonično obliko**:

1. če obstajajo ničelne vrstice, so te na dnu matrike,

- prvi neničelni element se nahaja desno od prvih neničelnih elementov predhodnih vrstic,
- prvi neničelni element v vrstici imenujemo **pivot** in je enak 1,
- pivot je edini neničelni element v vrstici.

Rang matrike predstavlja število neničelnih vrstic v vrstični kanonični obliki matrike; število neničelnih vrstic predstavlja število linearno neodvisnih enačb in je enako številu pivotnih elementov.

Primer preoblikovanja matrike **A**:

```
In [3]: import numpy as np #uvozimo numpy
        A = np.arange(9).reshape((3,3))+1
        A

Out[3]: array([[1, 2, 3],
               [4, 5, 6],
               [7, 8, 9]])
```

Očitno ima neničelni element `A[0,0]` vrednost 1 in je pivotni element. Prvo vrstico `A[0,:]` pomnožimo z -4 in produkt prištejemo drugi vrstici `A[1,:]-4A[0,:]`:

```
In [4]: A[1,:] -= A[1,0]*A[0,:]
        A

Out[4]: array([[ 1,  2,  3],
               [ 0, -3, -6],
               [ 7,  8,  9]])
```

Podobno naredimo za tretjo vrstico:

```
In [5]: A[2,:] -= A[2,0]*A[0,:]
        A

Out[5]: array([[  1,   2,   3],
               [  0,  -3,  -6],
               [  0,  -6, -12]])
```

Drugo vrstico sedaj delimo z `A[1,1]`, da dobimo pivot:

```
In [6]: A[1,:] = A[1,:]/A[1,1]
        A

Out[6]: array([[  1,   2,   3],
               [  0,   1,   2],
               [  0,  -6, -12]])
```

Odštejemo drugo vrstico od ostalih, da dobimo v drugem stolpcu ničle povsod, razen v drugi vrstici vrednost 1:

```
In [7]: A[0,:] -= A[0,1]*A[1,:] # odštevanje od prve vrstice
        A

Out[7]: array([[  1,   0,  -1],
               [  0,   1,   2],
               [  0,  -6, -12]])

In [8]: A[2,:] -= A[2,1]*A[1,:] # odštevanje od zadnje vrstice
        A

Out[8]: array([[ 1,  0, -1],
               [ 0,  1,  2],
               [ 0,  0,  0]])
```

Imamo dve neničelni vrstici; Matrika `A` ima dva pivota in predstavlja dve linearno neodvisni enačbi. Rang matrike je 2.

```
In [9]: A

Out[9]: array([[ 1,  0, -1],
               [ 0,  1,  2],
               [ 0,  0,  0]])
```

Rang matrike pa lahko določimo tudi s pomočjo `numpy` funkcije `numpy.linalg.matrix_rank` (dokumentacija[4]):

```
matrix_rank(M, tol=None)
```

kjer je `M` matrika, katere rang iščemo, `tol` opcijski parameter, ki določa mejo, pod katero se vrednosti v algoritmu smatrajo enake nič.

```
In [10]: np.linalg.matrix_rank(A)
```

```
Out[10]: 2
```

Če velja $r = \mathbf{rang}(\mathbf{A}) = \mathbf{rang}([\mathbf{A}|\mathbf{b}])$, potem rešitev **obstaja** (rečemo tudi, da je sistem **konsistenten**).

Konsistenten sistem ima:

- natanko eno rešitev, ko je število neznank n enako rangu r in
- neskončno mnogo rešitev, ko je rang r manjši od števila neznank n (rešitev je odvisna od $n - r$ parametrov).

Najprej se bomo omejili na sistem $m = n$ linearnih enačb z n neznankami ter velja $n = r$:

$$\mathbf{A}\,\mathbf{x} = \mathbf{b}.$$

Pod zgornjimi pogoji je matrika koeficientov $\mathbf{A}$ nesingularna ($|\mathbf{A}| \neq 0$) in sistem ima rešitev:

$$\mathbf{x} = \mathbf{A}^{-1}\,\mathbf{b}.$$

Poglejmo si primer sistema, ko so **enačbe linearno odvisne** ($r < n$):

```
In [11]: A = np.array([[1 , 2],
                       [2, 4]])
         b = np.array([1, 2])
         Ab = np.column_stack((A,b))
         Ab
```

```
Out[11]: array([[1, 2, 1],
                [2, 4, 2]])
```

S pomočjo `numpy` knjižnice poglejmo sedaj rang matrike koeficientov in razširjene matrike ter determinanto z uporabo `numpy.linalg.det` (dokumentacija[5]):

```
det(a)
```

kjer je `a` matrika (ali seznam matrik), katere determinanto iščemo; funkcija `det` vrne determinanto (ali seznam determinant).

```
In [12]: f'rang(A)={np.linalg.matrix_rank(A)}, rang(Ab)={np.linalg.matrix_rank(Ab)}, \
         število neznak: {len(A[:,0])}, det(A)={np.linalg.det(A)}'
```

[4] https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.matrix_rank.html
[5] https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.det.html

```
Out[12]: 'rang(A)=1, rang(Ab)=1, število neznak: 2, det(A)=0.0'
```

Poglejmo še primer, ko **rešitve sploh ni** (nekonsistenten sistem):

```
In [13]: A = np.array([[1 , 2],
                       [2, 4]])
         b = np.array([1, 1])
         Ab = np.column_stack((A,b))
         Ab

Out[13]: array([[1, 2, 1],
                [2, 4, 1]])

In [14]: f'rang(A)={np.linalg.matrix_rank(A)}, rang(Ab)={np.linalg.matrix_rank(Ab)}, \
         število neznak: {len(A[:,0])}, det(A)={np.linalg.det(A)}'

Out[14]: 'rang(A)=1, rang(Ab)=2, število neznak: 2, det(A)=0.0'
```

5.2.2 Norma in pogojenost sistemov enačb

Numerična naloga je slabo pogojena, če majhna sprememba podatkov povzroči veliko spremembo rezultata. V primeru *majhne spremembe podatkov*, ki povzročijo *majhno spremembo rezultatov*, pa je naloga **dobro pogojena**.

Sistem enačb je ponavadi dobro pogojen, če so absolutne vrednosti diagonalnih elementov matrike koeficientov velike v primerjavi z absolutnimi vrednostmi izven diagonalnih elementov.

Za sistem linearnih enačb $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ lahko računamo **število pogojenosti** (*angl.* condition number):

$$\text{cond}(\mathbf{A}) = ||\mathbf{A}||\,||\mathbf{A}^{-1}||.$$

Z $||\mathbf{A}||$ je označena **norma** matrike.

Obstaja več načinov računanja norme; navedimo dve:

- Evklidska norma (tudi Frobeniusova):

$$||\mathbf{A}||_e = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} A_{ij}^2}$$

- Norma vsote vrstic ali tudi neskončna norma:

$$||\mathbf{A}||_\infty = \max_{1\le i\le n} \sum_{j=1}^{n} |A_{ij}|$$

Pogojenost računamo z vgrajeno funkcijo `numpy.linalg.cond` (dokumentacija[6]):

```
cond(x, p=None)
```

ki sprejme dva parametra: matriko `x` in opcijski tip norme `p` (privzeti tip je `None`; v tem primeru se uporabi Evklidska/Frobeniusova norma).

Če je število pogojenosti majhno, potem je matrika dobro pogojena in obratno - pri slabi pogojenosti se število pogojenosti zelo poveča.

Žal je izračun pogojenosti matrike numerično relativno zahteven.

[6] https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.cond.html

Primer slabo pogojene matrike

Pogledali si bomo slabo pogojen sistem, kjer bomo z malenkostno spremembo na matriki koeficientov povzročili veliko spremembo rešitve.

Matrika koeficientov:

```
In [15]: A = np.array([[1 , 1],
                       [1, 1.00001]])
         np.linalg.cond(A)
```

```
Out[15]: 400002.00000320596
```

Vektor konstant:

```
In [16]: b = np.array([3, -3])
         Ab = np.column_stack((A,b))
         Ab
```

```
Out[16]: array([[ 1.     ,  1.     ,  3.     ],
                [ 1.     ,  1.00001, -3.     ]])
```

Preverimo rang in determinanto:

```
In [17]: f'rang(A)={np.linalg.matrix_rank(A)}, rang(Ab)={np.linalg.matrix_rank(Ab)}, \
         število neznak: {len(A[:,0])}, det(A)={np.linalg.det(A)}'
```

```
Out[17]: 'rang(A)=2, rang(Ab)=2, število neznak: 2, det(A)=1.000000000006551e-05'
```

Od druge enačbe odšejemo prvo:

```
In [18]: Ab[1,:] -= Ab[0,:]
         Ab
```

```
Out[18]: array([[  1.00000000e+00,   1.00000000e+00,   3.00000000e+00],
                [  0.00000000e+00,   1.00000000e-05,  -6.00000000e+00]])
```

Določimo `x1`:

```
In [19]: x1 = Ab[1,2]/Ab[1,1]
         x1
```

```
Out[19]: -599999.99999606924
```

Preostane še določitev `x0`:

```
In [20]: x0 = (Ab[0,2] - Ab[0,1]*x1)/Ab[0,0]
         x0
```

```
Out[20]: 600002.99999606924
```

Malenkostno spremenimo matriko koeficientov in ponovimo reševanje:

```
In [21]: A = np.array([[1 , 1],
                       [1, 1.0001 + 1e-5]]) # <= tukaj smo spremenili 1.0001 na 1.00011
         np.linalg.cond(A)

Out[21]: 36365.636446095115
```

Ponovimo izračun:

```
In [22]: Ab = np.column_stack((A,b))
         f'rang(A)={np.linalg.matrix_rank(A)}, rang(Ab)={np.linalg.matrix_rank(Ab)}, \
         število neznak: {len(A[:,0])}, det(A)={np.linalg.det(A)}'
         Ab[1,:] -= Ab[0,:]
         x1_ = Ab[1,2]/Ab[1,1]
         x0_ = (Ab[0,2] - Ab[0,1]*x1)/Ab[0,0]
```

Primerjamo obe rešitvi:

```
In [23]: [x0, x1] # prva rešitev

Out[23]: [600002.99999606924, -599999.99999606924]

In [24]: [x0_, x1_] # druga rešitev

Out[24]: [600002.99999606924, -54545.454545427521]
```

Ugotovimo, da je malenkostna sprememba enega koeficienta v matriki koeficientov povzročila veliko spremembo v rezultatu. Majhni spremembi podatkov se ne moremo izogniti, zaradi zapisa podatkov v računalniku.

5.2.3 Numerično reševanje sistemov linearnih enačb

Pogledali si bomo dva, v principu različna pristopa k reševanju sistemov linearnih enačb:

A) **Direktni pristop**: nad sistemom enačb izvajamo elementarne operacije, s katerimi predelamo sistem enačb v lažje rešljivega,

B) **Iterativni pristop**: izberemo začetni približek, nato pa približek iterativno izboljšujemo.

5.3 Gaussova eliminacija

Predpostavimo, da rešujemo sistem n enačb za n neznank, ki ima rang n. Tak sistem je enolično rešljiv.

Gaussova eliminacija spada med direktne metode, saj s pomočjo elementarnih vrstičnih operacij sistem enačb prevedemo v zgornje poravnani trikotni sistem (pod glavno diagonalo v razširjeni matriki so vrednosti nič).

Najprej pripravimo razširjeno matriko koeficientov:

$$[\mathbf{A}|\mathbf{b}] = \left[\begin{array}{cccc|c} A_{0,0} & A_{0,1} & \cdots & A_{0,n-1} & b_0 \\ A_{1,1} & A_{0,1} & \cdots & A_{1,n-1} & b_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ A_{n-1,1} & A_{n-1,1} & \cdots & A_{n-1,n-1} & b_{n-1} \end{array}\right]$$

Gaussovo eliminacijo si bomo pogledali na zgledu:

```
In [25]: A = np.array([[ 8., -6, 3],
                       [-6, 6,-6],
                       [ 3, -6, 6]])
         b = np.array([-14, 36, 6])
         Ab = np.column_stack((A,b))

In [26]: Ab

Out[26]: array([[  8.,  -6.,   3., -14.],
                [ -6.,   6.,  -6.,  36.],
                [  3.,  -6.,   6.,   6.]])
```

Korak 0: prvo vrstico pomnožimo z `Ab[1,0]/Ab[0,0]=-6/8` in odštejemo od druge:

```
In [27]: Ab[1,:] -= Ab[1,0]/Ab[0,0] * Ab[0,:]
         Ab

Out[27]: array([[  8.  ,  -6.  ,   3.  , -14.  ],
                [  0.  ,   1.5 ,  -3.75,  25.5 ],
                [  3.  ,  -6.  ,   6.  ,   6.  ]])
```

Nato prvo vrstico pomnožimo z `Ab[2,0]/Ab[0,0]=3/8` in odštejemo od tretje:

```
In [28]: Ab[2,:] -= Ab[2,0]/Ab[0,0] * Ab[0,:]
         Ab

Out[28]: array([[  8.   ,  -6.   ,   3.   , -14.   ],
                [  0.   ,   1.5  ,  -3.75 ,  25.5  ],
                [  0.   ,  -3.75 ,   4.875,  11.25 ]])
```

Korak 1: drugo vrstico pomnožimo z `Ab[2,1]/Ab[1,1]=-3.75/1.5` in odštejemo od tretje:

```
In [29]: Ab[2,:] -= Ab[2,1]/Ab[1,1] * Ab[1,:]
         Ab

Out[29]: array([[  8.  ,  -6.  ,   3.  , -14.  ],
                [  0.  ,   1.5 ,  -3.75,  25.5 ],
                [  0.  ,   0.  ,  -4.5 ,  75.  ]])
```

Dobili smo zgornje trikotno matriko in Gaussova eliminacija je končana. Lahko izračunamo rešitev, torej določimo vektor neznak x z **obratnim vstavljanjem**.

Iz zadnje vrstice zgornje trikotne matrike izračunamo x_2:

```
In [30]: Ab

Out[30]: array([[  8.  ,  -6.  ,   3.  , -14.  ],
                [  0.  ,   1.5 ,  -3.75,  25.5 ],
                [  0.  ,   0.  ,  -4.5 ,  75.  ]])
```

```
In [31]: x = np.zeros(3) #pripravimo prazen seznam
         x[2] = Ab[2,-1]/Ab[2,2]
         x

Out[31]: array([  0.        ,   0.        , -16.66666667])
```

S pomočjo predzadnje vrstice izračunamo x_1:

```
In [32]: Ab

Out[32]: array([[  8.  ,  -6.  ,   3.  , -14.  ],
                [  0.  ,   1.5 ,  -3.75,  25.5 ],
                [  0.  ,   0.  ,  -4.5 ,  75.  ]])

In [33]: x[1] = (Ab[1,-1] - Ab[1,2]*x[2]) / Ab[1,1]
         x

Out[33]: array([  0.        , -24.66666667, -16.66666667])
```

S pomočjo prve vrstice nato izračunamo x_0:

```
In [34]: Ab

Out[34]: array([[  8.  ,  -6.  ,   3.  , -14.  ],
                [  0.  ,   1.5 ,  -3.75,  25.5 ],
                [  0.  ,   0.  ,  -4.5 ,  75.  ]])

In [35]: x[0] = (Ab[0,3] - Ab[0,1:3]@x[1:]) / Ab[0,0]
         x

Out[35]: array([-14.        , -24.66666667, -16.66666667])
```

Preverimo rešitev:

```
In [36]: A @ x - b

Out[36]: array([  0.00000000e+00,   0.00000000e+00,   1.42108547e-14])
```

Povzetek Gaussove eliminacije

V modul `orodja.py` shranimo funkciji:

```
In [37]: def gaussova_eliminacija(A, b, prikazi_korake = False):
             Ab = np.column_stack((A, b))
             for p, pivot_vrsta in enumerate(Ab[:-1]):
                 for vrsta in Ab[p+1:]:
                     if pivot_vrsta[p]:
                         vrsta[p:] -= pivot_vrsta[p:]*vrsta[p]/pivot_vrsta[p]
                     else:
                         raise Exception('Deljenje z 0.')
                 if prikazi_korake:
```

```
                print('Korak: {:g}'.format(p))
                print(Ab)
        return Ab

    def gaussova_el_resitev(Ub):
        v = len(Ub)
        x = np.zeros(v)
        for p, pivot_vrsta in enumerate(Ub[::-1]):
            x[v-p-1] = (pivot_vrsta[-1] - pivot_vrsta[v-p:-1] @ x[v-p:] ) / (pivot_vrsta[v-p-1])
        return x
```

Algoritem, s katerim iz zgornje trikotnega sistema enačb $\mathbf{U}\,\mathbf{x} = \mathbf{b}$ izračunamo rešitev, imenujemo **obratno vstavljanje** (angl. *back substitution*); $\mathbf{U}$ je zgornje trikotna matrika.

V kolikor bi reševali sistem $\mathbf{L}\,\mathbf{x} = \mathbf{b}$ in je $\mathbf{L}$ spodnje trikotna matrika, bi to metodo imenovali **direktno vstavljanje** (angl. *forward substitution*).

```
In [38]: Ub = gaussova_eliminacija(A, b, prikazi_korake=False)
         Ub

Out[38]: array([[  8.  ,  -6.  ,   3.  , -14.  ],
                [  0.  ,   1.5 ,  -3.75,  25.5 ],
                [  0.  ,   0.  ,  -4.5 ,  75.  ]])

In [39]: gaussova_el_resitev(Ub)

Out[39]: array([-14.        , -24.66666667, -16.66666667])
```

5.3.1 Numerična zahtevnost

Numerično zahtevnost ocenjujemo po številu matematičnih operacij, ki so potrebne za izračun. Za rešitev n linearnih enačb tako z Gaussovo eliminacijo potrebujemo približno $n^3/3$ matematičnih operacij. Za določitev neznank $\mathbf{x}$ potrebujemo še dodatnih približno n^2 operacij.

Pri Gaussovi eliminaciji smo eliminacijo izvedli samo za člene pod diagonalo; če bi z eliminacijo nadaljevali in jo izvedli tudi za člene nad diagonalo, bi izvedli t. i. *Gauss-Jordanovo* eliminacijo, za katero pa potrebujemo dodatnih približno $n^3/3$ operacij* (kar se šteje kot glavna slabost te metode).

* Nekaj komentarjev na temo števila numeričnih operacij najdete tukaj: pinm.ladisk.si[7].

5.3.2 Uporaba knjižnjice `numpy`

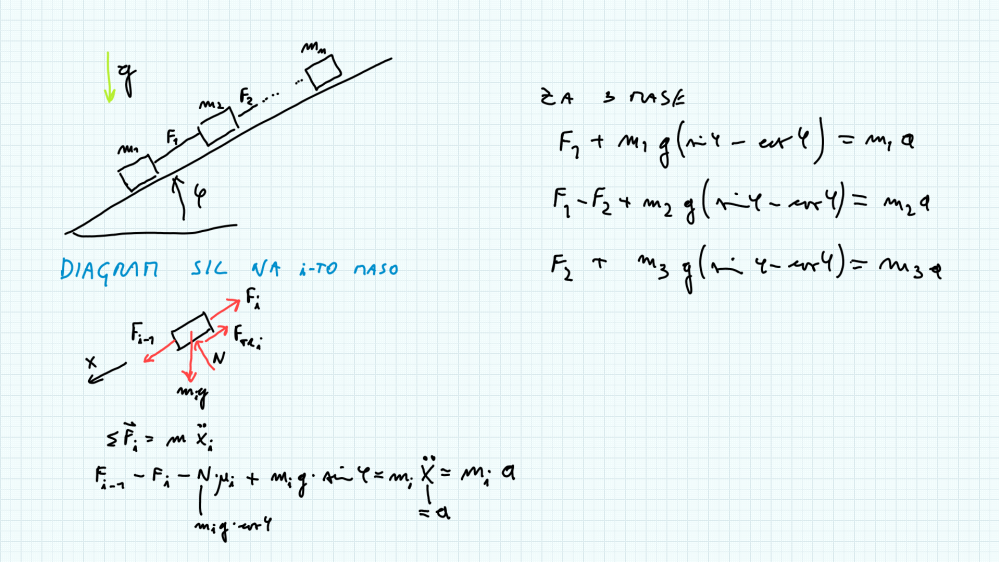
Reševanje sistema linearnih enačb z `numpy.linalg.solve` (dokumentacija[8]):

```
solve(a, b)
```

kjer je `a` matrika koeficientov (ali seznam matrik) in je `b` vektor konstant (ali seznam vektorjev). Funkcija vrne vektor (ali seznam vektorjev) rešitev.

```
In [40]: np.linalg.solve(A, b)

Out[40]: array([-14.        , -24.66666667, -16.66666667])
```

[7] http://pinm.ladisk.si/303/kako-dolo%C4%8Damo-numeri%C4%8Dno-zahtevnost-algoritmov
[8] https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.solve.html



5.4 Nekaj vprašanj za razmislek!

1. Za sistem enačb:

$$\mathbf{A} = \begin{bmatrix} 1 & -4 & 1 \\ 1 & 6 & -1 \\ 2 & -1 & 2 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 7 \\ 13 \\ 5 \end{bmatrix}$$

   najdite rešitev s pomočjo `np.linalg.solve()`.

- Za zgoraj definirano matriko **A** določite Evklidsko normo (lastni program).
- Za zgoraj definirano matriko **A** določite neskončno normo (lastni program).
- Za zgoraj definirano matriko **A** določite pogojenost (`numpy` funkcija).
- Definirajte funkcijo `gauss_elim`, ki za poljubno matriko **A** in vektor **b** izvede Gaussovo eliminacijo (posebej za matriko in posebej za vektor tako, da ne sestavite razširjene matrike $[\mathbf{A}|\mathbf{b}]$).
- Definirajte funkcijo `gauss_elim_x`, ki za razultat funkcije `gauss_elim` najde ustrezne vrednosti vektorja **x**.
- Zgornji funkciji dopolnite s štetjem matematičnih operacij.
- Na sliki je prikazan sistem mas:

  Predpostavite, da se sistem zaradi teže giblje po klancu navzdol z neznanim pospeškom a in da so vrvi napete z neznanimi silami F_i. Znane veličine so (sami jih določite): posamično telo ima maso m_i, koeficient trenja s podlago $\mu_i = 1$, $g = 9{,}81\text{m/s}^2$, $\varphi = 55°$. Določite sistem enačb v primeru dveh teles. Določite matriko koeficientov **A** in vektorja **b** ter **x**.
- Za zgoraj definiran sistem mas predpostavite, da je mas 4 (ali več) ter določite matriko koeficientov **A**, in vektorja **x** ter **b**. Rešite sistem s pomočjo Gaussove elimnacije/LU razcepa ali `np.linalg.solve`. Preverite pogojenost!
- V sistemu mas dobimo fizikalno nekonsistentno rešitev, če imamo v kateri od vrvi tlačno silo (vrv ne prenese tlačne sile). Preverite, ali je to v vašem primeru res. Ustrezno spremenite koeficient(e) trenja, da se bo to zgodilo.

5.5 Dodatno

Poglejte si strani: * micropython.org[9] * kivy.org[10] * openmodal.com[11]

5.5.1 Primer simbolnega reševanja sistema linearnih enačb v okviru `sympy`

```
In [41]: import sympy as sym
         sym.init_printing()
```

```
In [42]: A11, A12, A21, A22 = sym.symbols('A11, A12, A21, A22')
         x1, x2 = sym.symbols('x1, x2')
         b1, b2 = sym.symbols('b1, b2')
         A = sym.Matrix([[A11, A12],
                         [A21, A22]])
         x = sym.Matrix([[x1],
                         [x2]])
         b = sym.Matrix([[b1],
                         [b2]])
```

```
In [43]: eq = sym.Eq(A*x,b)
         eq
```

Out[43]:

$$\begin{bmatrix} A_{11}x_1 + A_{12}x_2 \\ A_{21}x_1 + A_{22}x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

```
In [44]: #%%timeit
         resitev = sym.solve(eq,[x1, x2])
         resitev
```

Out[44]:

$$\left\{ x_1 : \frac{-A_{12}b_2 + A_{22}b_1}{A_{11}A_{22} - A_{12}A_{21}}, \quad x_2 : \frac{A_{11}b_2 - A_{21}b_1}{A_{11}A_{22} - A_{12}A_{21}} \right\}$$

```
In [45]: A.det()
```

Out[45]:

$$A_{11}A_{22} - A_{12}A_{21}$$

[9] http://www.micropython.org
[10] http://www.kivy.org
[11] http://www.openmodal.com

Poglavje 6

Sistemi linearnih enačb (2)

6.1 Razcep LU

Za rešitev sistema linearnih enačb zahteva Gaussov eliminacijski postopek najmanjše število računskih operacij.

V primeru, ko se matrika koeficientov **A** ne spreminja in se spreminja zgolj vektor konstant **b**, se je mogoče izogniti ponovni Gaussovi eliminaciji matrike koeficientov. Z razcepom matrike **A** lahko pridemo do rešitve z manj računskimi operacijami. V ta namen si bomo pogledali razcep LU!

Poljubno matriko lahko zapišemo kot produkt dveh matrik:

$$\mathbf{A} = \mathbf{B}\,\mathbf{C}.$$

Pri tem je možnosti za zapis matrik **B** in **C** neskončno veliko.

Pri razcepu LU zahtevamo, da je matrika **B** spodnje trikotna in matrika **C** zgornje trikotna:

$$\mathbf{A} = \mathbf{L}\,\mathbf{U}.$$

Vsaka od matrik **L** in **U** ima $(n+1)\,n/2$ neničelnih elementov; skupaj torej n^2+n neznank. Znana matrika **A** definira n^2 vrednosti. Za enolično določitev matrik **L** in **U** torej manjka n enačb. Tukaj bomo uporabili razcep LU, ki dodatne enačbe pridobi s pogojem $L_{ii} = 1$, $i = 0, 1, \ldots, n-1$.

Sistem linearnih enačb:

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

torej zapišemo z razcepom matrike **A**:

$$\mathbf{L}\,\underbrace{\mathbf{U}\,\mathbf{x}}_{\mathbf{y}} = \mathbf{b}.$$

Do rešitve sistema $\mathbf{A}\mathbf{x} = \mathbf{b}$ sedaj pridemo tako, da rešimo dva trikotna sistema enačb.

Najprej izračunamo vektor **y**:

$$\mathbf{L}\,\mathbf{y} = \mathbf{b}. \qquad \text{(direktno vstavljanje)}$$

Ko je **y** izračunan, lahko iz:

$$\mathbf{U}\,\mathbf{x} = \mathbf{y} \qquad \text{(obratno vstavljanje)}$$

določimo **x**.

6.1.1 Razcep LU matrike koeficientov A

V nadaljevanju bomo pokazali, da Gaussova eliminacija dejanjsko predstavlja razcep LU matrike koeficientov **A**. Pri tem te si bomo pomagali s simbolnim izračunom, zato uvozimo paket `sympy`:

```
In [1]: import sympy as sym # uvozimo sympy
        sym.init_printing() # za lep prikaz izrazov
```

Prikaz začnimo na primeru simbolno zapisanih matrik **L** in **U** dimenzije 3×3:

```
In [2]: L21, L31, L32 = sym.symbols('L21, L31, L32')
        U11, U12, U13, U22, U23, U33 = sym.symbols('U11, U12, U13, U22, U23, U33')
        L = sym.Matrix([[  1,   0,  0],
                        [L21,   1,  0],
                        [L31, L32,  1]])
        U = sym.Matrix([[U11, U12, U13],
                        [  0, U22, U23],
                        [  0,   0, U33]])
```

Matrika koeficientov **A** zapisana z elementi matrik **L** in **U** torej je:

```
In [3]: A = L*U
        A
```

```
Out[3]:
```

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ L_{21}U_{11} & L_{21}U_{12} + U_{22} & L_{21}U_{13} + U_{23} \\ L_{31}U_{11} & L_{31}U_{12} + L_{32}U_{22} & L_{31}U_{13} + L_{32}U_{23} + U_{33} \end{bmatrix}$$

Izvedimo sedaj Gaussovo eliminacijo nad matriko koeficientov **A**.

S pomočjo prve vrstice izvedemo Gaussovo eliminacijo v prvem stolpcu:

```
In [4]: A[1,:] -= L21 * A[0,:]
        A[2,:] -= L31 * A[0,:]
        A
```

```
Out[4]:
```

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & L_{32}U_{22} & L_{32}U_{23} + U_{33} \end{bmatrix}$$

Nadaljujemo v drugem stolpcu:

```
In [5]: A[2,:] -= L32 * A[1,:]
        A
```

```
Out[5]:
```

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}$$

Iz zgornje eliminacije ugotovimo: 1. matrika **U** je enaka matriki, ki jo dobimo, če izvedemo Gaussovo eliminacijo nad matriko koeficientov **A**. * izven diagonalni členi **L** so faktorji, ki smo jih uporabili pri Gaussovi eliminaciji.

6.1.2 Numerična implementacija razcepa LU

Numerično implementacijo si bomo pogledali na sistemu, ki je definiran kot:

```
In [6]: import numpy as np

        A = np.array([[8, -6, 3],
                      [-6, 6, -6],
                      [3, -6, 6]], dtype=float)
        b = np.array([-14, 36, 6], dtype=float)
```

Izvedimo Gaussovo eliminacijo in koeficiente, s katerim množimo pivotno vrsto `m`, shranjuje v matriko `L` na mesto z indeksi, kot jih ima v matriki **A** eliminirani element.

```
In [7]: (v, s) = A.shape # v=število vrstic, s=število stolpcev
        U = A.copy()     # pripravimo matriko U kot kopijo A
        L = np.zeros_like(A) # pripravimo matriko L dimenzije enake A (vrednosti 0)
        # eliminacija
        for p, pivot_vrsta in enumerate(U[:-1]):
            for i, vrsta in enumerate(U[p+1:]):
                if pivot_vrsta[p]:
                    m = vrsta[p]/pivot_vrsta[p]
                    vrsta[p:] = vrsta[p:]-pivot_vrsta[p:]*m
                    L[p+1+i, p] = m
            print('Korak: {:g}'.format(p))
            print(U)
```

```
Korak: 0
[[ 8.    -6.     3.   ]
 [ 0.     1.5   -3.75 ]
 [ 0.    -3.75   4.875]]
Korak: 1
[[ 8.   -6.    3.  ]
 [ 0.    1.5  -3.75]
 [ 0.    0.   -4.5 ]]
```

```
In [8]: L
```

```
Out[8]: array([[ 0.   ,  0.   ,  0.   ],
               [-0.75 ,  0.   ,  0.   ],
               [ 0.375, -2.5  ,  0.   ]])
```

Dopolnimo diagonalo **L**:

```
In [9]: for i in range(v):
            L[i, i] = 1.
```

```
In [10]: L

Out[10]: array([[ 1.   ,  0.   ,  0.   ],
                [-0.75 ,  1.   ,  0.   ],
                [ 0.375, -2.5  ,  1.   ]])
```

Sedaj rešimo spodnje trikotni sistem enačb $\mathbf{L}\,\mathbf{y} = \mathbf{b}$:

```
In [11]: # direktno vstavljanje
         y = np.zeros_like(b)
         for i, b_ in enumerate(b):
             y[i] = (b_ - np.dot(L[i, :i], y[:i]))

In [12]: y

Out[12]: array([-14. ,  25.5,  75. ])
```

Nadaljujemo z reševanjem zgornje trikotnega sistema $\mathbf{U}\,\mathbf{x} = \mathbf{y}$:

```
In [13]: U

Out[13]: array([[ 8.  , -6.  ,  3.  ],
                [ 0.  ,  1.5 , -3.75],
                [ 0.  ,  0.  , -4.5 ]])

In [14]: y

Out[14]: array([-14. ,  25.5,  75. ])

In [15]: # obratno vstavljanje
         x = np.zeros_like(b)
         for i in range(v-1, -1,-1):
             x[i] = (y[i] - np.dot(U[i, i+1:], x[i+1:])) / U[i, i]
         x

Out[15]: array([-14.        , -24.66666667, -16.66666667])
```

Kakor smo navedli zgoraj, ob spremembi vektorja konstant **b** ponovna Gaussova eliminacija ni potrebna. Izvesti je treba samo direktno in nato obratno vstavljanje. Poglejmo primer:

```
In [16]: b = np.array([-1., 6., 7.])
         y = np.zeros_like(b)
         for i, b_ in enumerate(b):#direktno vstavljanje
             y[i] = (b_ - np.dot(L[i, :i], y[:i]))
         x = np.zeros_like(b)
         for i in range(v-1, -1,-1):# obratno vstavljanje
             x[i] = (y[i] - np.dot(U[i, i+1:], x[i+1:])) / U[i, i]
         x

Out[16]: array([-4.33333333, -7.88888889, -4.55555556])

In [17]: A.dot(x)

Out[17]: array([-1.,  6.,  7.])
```

6.2 Pivotiranje

Poglejmo si spodnji sistem enačb:

```
In [18]: A = np.array([[0, -6, 6],
                       [-6, 6, -6],
                       [8, -6, 3]], dtype=float) # poskusite tukaj izpustiti dtype=float in preveriet r
         b = np.array([6, 36, -14], dtype=float)
```

Če bi izvedli Gaussovo eliminacijo v prvem stolpcu matrike **A**:

```
In [19]: A[1,:] - A[1,0]/A[0,0] * A[0,:]
```

```
C:\Users\Janko\Anaconda3\lib\site-packages\ipykernel_launcher.py:1: RuntimeWarning: divide by zero enco
  """Entry point for launching an IPython kernel.
C:\Users\Janko\Anaconda3\lib\site-packages\ipykernel_launcher.py:1: RuntimeWarning: invalid value encou
  """Entry point for launching an IPython kernel.
```

```
Out[19]: array([ nan, -inf,  inf])
```

Opazimo, da imamo težavo z deljenjem z 0 v prvi vrstici. Elementarne operacije, ki jih nad sistemom lahko izvajamo, dovoljujejo zamenjavo poljubnih vrstic. Sistem lahko preuredimo tako, da pivotni element ni enak 0. Vseeno se lahko zgodi, da ima pivotni element, s katerim delimo, zelo majhno vrednost ε. Ker bi to povečevalo zaokrožitveno napako, izmed vseh vrstic za pivotno vrstico izberemo tisto, katere pivot ima največjo absolutno vrednost.

Če med Gaussovo eliminacijo zamenjamo vrstice tako, da je pivotni element največji, to imenujemo **pivotiranje vrstic** ali tudi **delno pivotiranje**. Tako dosežemo, da je Gaussova eliminacija numerično stabilna.

Pokazati je mogoče, da pri reševanju sistema enačb $\mathbf{A}\,\mathbf{x} = \mathbf{b}$, pri katerem je matrika **A** diagonalno dominantna, pivotiranje po vrsticah ni potrebno. Reševanje je brez pivotiranja numerično stabilno.

Pravokotna matrika **A** dimenzije n je **diagonalno dominantna**, če je absolutna vrednost diagonalnega elementa vsake vrstice večja od vsote absolutnih vrednosti ostalih elementov v vrstici:

$$|A_{ii}| > \sum_{j=1, j\neq i}^{n} |A_{ij}|$$

6.2.1 Gaussova eliminacija z delnim pivotiranjem

Pogledali si bomo Gaussovo eliminacijo z **delnim pivotiranjem**. Brez delnega pivotiranja v i-tem koraku eliminacije izberemo vrstico i za pivotiranje. Pri delnem pivotiranju pa najprej preverimo, ali je i-ti diagonalni element po absolutni vrednosti največji element v stolpcu i na ali pod diagonalo; če ni, zamenjamo vrstico i s tisto vrstico pod njo, v kateri je v stolpcu i po absolutni vrednosti največji element. Z delnim pivotiranjem zmanjšamo vpliv zaokrožitvene napake na rezultat.

V koliko bi izvedli **polno pivotiranje**, bi poleg zamenjave vrstic uporabili tudi zamenjavo vrstnega reda spremenljivk (zamenjava stolpcev). Polno pivotiranje izboljša stabilnost, se pa redko uporablja in ga tukaj ne bomo obravnavali.

Algoritem za Gaussovo eliminacijo z delnim pivotiranjem torej je:

```
In [20]: def gaussova_eliminacija_pivotiranje(A, b, prikazi_korake=False):
             """ Vrne Gaussovo eliminacijo razširjene matrike koeficientov,
```

```
            uporabi delno pivotiranje.

            :param A: matrika koeficientov
            :param b: vektor konstant
            :param prikazi_korake: ali izpišem posamezne korake
            :return Ab: trapezna razširjena matrika koeficientov
            """
            Ab = np.column_stack((A, b))
            for p in range(len(Ab)-1):
                p_max = np.argmax(np.abs(Ab[p:,p]))+p
                if p != p_max:
                    Ab[[p], :], Ab[[p_max], :] = Ab[[p_max], :], Ab[[p], :]
                pivot_vrsta = Ab[p, :]
                for vrsta in Ab[p + 1:]:
                    if pivot_vrsta[p]:
                        vrsta[p:] -= pivot_vrsta[p:] * vrsta[p] / pivot_vrsta[p]
                if prikazi_korake:
                    print('Korak: {:g}'.format(p))
                    print('Pivot vrsta:', pivot_vrsta)
                    print(Ab)
            return Ab
```

```
In [21]: A
```

```
Out[21]: array([[ 0., -6.,  6.],
                [-6.,  6., -6.],
                [ 8., -6.,  3.]])
```

```
In [22]: Ab = gaussova_eliminacija_pivotiranje(A, b, prikazi_korake=True)
```

```
Korak: 0
Pivot vrsta: [  8.  -6.   3. -14.]
[[  8.    -6.     3.   -14.  ]
 [  0.     1.5   -3.75  25.5 ]
 [  0.    -6.     6.     6.  ]]
Korak: 1
Pivot vrsta: [ 0. -6.  6.  6.]
[[  8.    -6.     3.   -14.  ]
 [  0.    -6.     6.     6.  ]
 [  0.     0.    -2.25  27.  ]]
```

6.2.2 Razcep LU z delnim pivotiranjem

Podobno kakor pri Gaussovi eliminaciji lahko tudi razcep LU razširimo z delnim pivotiranjem. Reševanje tako postane numerično **stabilno**. Pri tem moramo shraniti informacijo o zamenjavi vrstic, ki jo potem posredujemo v funkcijo za rešitev ustreznih trikotnih sistemov.

```
In [23]: def LU_razcep_pivotiranje(A, prikazi_korake=False):
            """ Vrne razcep LU matriko in vektor zamenjanih vrstic pivotiranje,
            uporabi delno pivotiranje.
```

```
        :param A: matrika koeficientov
        :param prikazi_korake: izpišem posamezne korake
        :return LU: LU matrika
        :return pivotiranje: vektor zamenjave vrstic (pomembno pri iskanju rešitve)
        """
        LU = A.copy()
        pivotiranje = np.arange(len(A))
        for p in range(len(LU)-1):
            p_max = np.argmax(np.abs(LU[p:,p]))+p
            if p != p_max:
                LU[[p], :], LU[[p_max], :] = LU[[p_max], :], LU[[p], :]
                pivotiranje[p], pivotiranje[p_max] = pivotiranje[p_max], pivotiranje[p]
            pivot_vrsta = LU[p, :]
            for vrsta in LU[p + 1:]:
                if pivot_vrsta[p]:
                    m = vrsta[p] / pivot_vrsta[p]
                    vrsta[p:] -= pivot_vrsta[p:] * m
                    vrsta[p] = m
                else:
                    raise Exception('Deljenje z 0.')
            if prikazi_korake:
                print('Korak: {:g}'.format(p))
                print('Pivot vrsta:', pivot_vrsta)
                print(LU)
        return LU, pivotiranje
```

Poglejmo si primer:

```
In [24]: A = np.array([[0, -6, 6],
                       [-6, 6, -6],
                       [8, -6, 3]], dtype=float)
         b = np.array([-14, 36, 6], dtype=float)
```

```
In [25]: lu, piv = LU_razcep_pivotiranje(A, prikazi_korake=True)
```

```
Korak: 0
Pivot vrsta: [ 8. -6.  3.]
[[ 8.   -6.    3.  ]
 [-0.75  1.5  -3.75]
 [ 0.   -6.    6.  ]]
Korak: 1
Pivot vrsta: [ 0. -6.  6.]
[[ 8.   -6.    3.  ]
 [ 0.   -6.    6.  ]
 [-0.75 -0.25 -2.25]]
```

V zgornjem primeru smo uporabili kompaketen način zapisa trikotnih matrik $\mathbf{L}$ in $\mathbf{U}$; vsaka je namreč definirana s 6 elementi, pri matriki $\mathbf{L}$ pa vemo, da so diagonalni elementi enaki 1. Matrika `lu` tako vsebuje $3 \times 3 = 9$ elementov:

```
In [26]: lu
```

```
Out[26]: array([[ 8.  , -6.  ,  3.  ],
                [ 0.  , -6.  ,  6.  ],
                [-0.75, -0.25, -2.25]])
```

Na diagonali in nad diagonalo so vrednosti zgornje trikotne matrike **U**, pod diagonalo pa so poddiagonalni elementi matrike **L**. Pri izračunu rešitve bomo upoštevali, da so diagonalne vrednosti **L** enake 1.

Numerični seznam `piv` nam pove, kako so bile zamenjane vrstice, kar je treba upoštevati pri izračunu rešitve:

```
In [27]: piv

Out[27]: array([2, 0, 1])
```

Določimo sedaj še rešitev (koda za računanje rešitve je v modulu `orodja.py`)

```
In [28]: from moduli import orodja

In [29]: r = orodja.LU_resitev_pivotiranje(lu, b, piv)
         r

Out[29]: array([ -3.66666667, -14.11111111, -16.44444444])
```

Preverjanje rešitve:

```
In [30]: A@r

Out[30]: array([-14.,  36.,   6.])
```

6.3 Modul `SciPy`

Modul `SciPy` temelji na `numpy` modulu in vsebuje veliko različnih visokonivojskih programov/modulov/funkcij. Teoretično ozadje modulov so seveda različni numerični algoritmi; nekatere spoznamo tudi v okviru tega učbenika. Dober vir teh numeričnih algoritmov v povezavi s `SciPy` predstavlja dokumentacija[1].

Kratek pregled hierarhije modula:

- Linearna algebra (scipy.linalg[2])
- Integracija (scipy.integrate[3])
- Optimizacija (scipy.optimize[4])
- Interpolacija (scipy.interpolate[5])
- Fourierjeva transformacija (scipy.fftpack[6])

- Problem lastnih vrednosti (redke matrike) (scipy.sparse[7])

[1] https://docs.scipy.org/doc/scipy/reference/tutorial/
[2] http://docs.scipy.org/doc/scipy/reference/linalg.html
[3] http://docs.scipy.org/doc/scipy/reference/integrate.html
[4] http://docs.scipy.org/doc/scipy/reference/optimize.html
[5] http://docs.scipy.org/doc/scipy/reference/interpolate.html
[6] http://docs.scipy.org/doc/scipy/reference/fftpack.html
[7] http://docs.scipy.org/doc/scipy/reference/sparse.html

- Statistika (scipy.stats[8])
- Procesiranje signalov (scipy.signal[9])
- Posebne funkcije (scipy.special[10])
- Večdimenzijsko procesiranje slik (scipy.ndimage[11])
- Delo z datotekami (scipy.io[12])

V sledečih predavanjih si bomo nekatere podmodule podrobneje pogledali.

Poglejmo si, kako je znotraj `SciPy` implementiran *razcep LU*:

```
In [31]: from scipy.linalg import lu_factor, lu_solve
```

Funkcija `scipy.linalg.lu_factor` (dokumentacija[13]):

```
lu_factor(a, overwrite_a=False, check_finite=True)
```

zahteva vnos matrike koeficientov (ali seznama matrik koeficientov) `a`, `overwrite_a` v primeru `True` z rezultatom razcepa prepiše vrednost `a` (to je lahko pomembno, da se prihrani spomin in poveča hitrost). Funkcija `lu_factor` vrne terko `(lu, piv)`:

- `lu` - **L** **U** matrika (matrika, ki je enake dimenzije kot `a`, vendar pod diagonalo vsebuje elemente **L**, preostali elementi pa definirajo **U**; diagonalni elementi **L** imajo vrednosti 1.
- `piv` - pivotni indeksi, predstavljajo permutacijsko matriko `P`: vrsta `i` matrike `a` je bila zamenjana z vrsto `piv[i]` (v vsakem koraku se upošteva predhodno stanje, malo drugačna logika kot v naši funkciji `LU_razcep_pivotiranje`).

`lu_factor` uporabljamo v paru s funkcijo `scipy.linalg.lu_solve` (dokumentacija[14]), ki nam poda rešitev sistema:

```
lu_solve(lu_and_piv, b, trans=0, overwrite_b=False, check_finite=True)
```

`lu_and_piv` je terka rezultata `(lu, piv)` iz `lu_factor`, `b` je vektor (ali seznam vektorjev) konstant. Ostali parametri so opcijski.

Poglejmo si uporabo:

```
In [32]: A = np.array([[0, -6, 6],
                       [-6, 6, -6],
                       [8, -6, 3]], dtype=float)
         b = np.array([-14, 36, 6], dtype=float)

In [33]: lu, piv = lu_factor(A)

In [34]: lu

Out[34]: array([[ 8.  , -6.  ,  3.  ],
                [ 0.  , -6.  ,  6.  ],
                [-0.75, -0.25, -2.25]])
```

[8] http://docs.scipy.org/doc/scipy/reference/stats.html
[9] http://docs.scipy.org/doc/scipy/reference/signal.html
[10] http://docs.scipy.org/doc/scipy/reference/special.html
[11] http://docs.scipy.org/doc/scipy/reference/ndimage.html
[12] http://docs.scipy.org/doc/scipy/reference/io.html
[13] https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lu_factor.html
[14] https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.lu_solve.html

```
In [35]: piv

Out[35]: array([2, 2, 2], dtype=int32)
```

Pridobimo rešitev:

```
In [36]: r = lu_solve((lu, piv), b)
         r

Out[36]: array([ -3.66666667, -14.11111111, -16.44444444])
```

Preverimo ustreznost rešitve:

```
In [37]: A@r

Out[37]: array([-14.,  36.,   6.])
```

6.4 Računanje inverzne matrike

Inverzno matriko h kvadratni matriki $\mathbf{A}$ reda $n \times n$ označimo z $\mathbf{A}^{-1}$. Je matrika reda $n \times n$, takšna, da velja

$$\mathbf{A}\,\mathbf{A}^{-1} = \mathbf{A}^{-1}\,\mathbf{A} = \mathbf{I},$$

kjer je $\mathbf{I}$ enotska matrika.

Najbolj učinkovit način za izračun inverzne matrike od matrike $\mathbf{A}$ je rešitev matrične enačbe:

$$\mathbf{A}\,\mathbf{X} = \mathbf{I}.$$

Matrika $\mathbf{X}$ je inverzna matriki $\mathbf{A}$: $\mathbf{A}^{-1} = \mathbf{X}$.

Izračun inverzne matrike je torej enak reševanju n sistemov n linarnih enačb:

$$\mathbf{A}\,\mathbf{x}_i = \mathbf{b}_i, \qquad i = 0, 1, 2, \ldots, n-1,$$

kjer je $\mathbf{b}_i$ i-ti stolpec matrike $\mathbf{B} = \mathbf{I}$.

Numerična zahtevnost: izvedemo razcep LU nad matriko $\mathbf{A}$ (računski obseg reda n^3) in nato poiščemo rešitev za vsak $\mathbf{x}_i$ ($2\,n^2$ računskih operacij za vsak i). Skupni računski obseg je torej $3\,n^3$.

```
In [38]: lu_piv = lu_factor(A)
         lu_piv

Out[38]: (array([[ 8.  , -6.  ,  3.  ],
                 [ 0.  , -6.  ,  6.  ],
                 [-0.75, -0.25, -2.25]]), array([2, 2, 2], dtype=int32))
```

Tukaj se sedaj pokaže smisel razcepa LU; razcep namreč izračunamo samo enkrat in potem za vsak vektor konstant poiščemo rešitev tako, da rešimo dva trikotna sistema. Če bi sisteme reševali po Gaussovi metodi, bi rabili $(n^3 + n^2)\,n$ računskih operacij.

Izračunajmo inverzno matriko od matrike $\mathbf{A}$. Najprej z uporabo `np.identity()` pripravimo enotsko matriko:

```
In [39]: np.identity(len(A))
```

```
Out[39]: array([[ 1.,  0.,  0.],
                [ 0.,  1.,  0.],
                [ 0.,  0.,  1.]])
```

Nato rešimo sistem enačb za vsak stolpec enotske matrike:

```
In [40]: A_inv = np.zeros_like(A) #zeros_like vrne matriko oblike in tipa matrike A z vrednostmi 0
         for b, a_inv in zip(np.identity(len(A)), A_inv):
             a_inv[:] = lu_solve(lu_piv, b)
         A_inv = A_inv.T
```

Rešitev torej je:

```
In [41]: A_inv
```

```
Out[41]: array([[ -1.66666667e-01,  -1.66666667e-01,   1.38777878e-17],
                [ -2.77777778e-01,  -4.44444444e-01,  -3.33333333e-01],
                [ -1.11111111e-01,  -4.44444444e-01,  -3.33333333e-01]])
```

Preverimo rešitev:

```
In [42]: A@A_inv
```

```
Out[42]: array([[  1.00000000e+00,   0.00000000e+00,   1.11022302e-16],
                [ -1.11022302e-16,   1.00000000e+00,  -1.11022302e-16],
                [  1.66533454e-16,   0.00000000e+00,   1.00000000e+00]])
```

Rešitev z uporabo Numpy:

```
In [43]: #%%timeit
         np.linalg.inv(A)
```

```
Out[43]: array([[ -1.66666667e-01,  -1.66666667e-01,   1.38777878e-17],
                [ -2.77777778e-01,  -4.44444444e-01,  -3.33333333e-01],
                [ -1.11111111e-01,  -4.44444444e-01,  -3.33333333e-01]])
```

Z ustrezno uporabo funkcij iz modula `SciPy` lahko do rešitve pridemo še hitreje. V funkcijo `lu_solve` lahko vstavimo vektor **b** ali matriko **B**, katere posamezni stolpec i predstavlja nov vektor konstant $\mathbf{b}_i$.

```
In [44]: #%%timeit
         lu_solve(lu_piv, np.identity(len(A)))
```

```
Out[44]: array([[ -1.66666667e-01,  -1.66666667e-01,   1.38777878e-17],
                [ -2.77777778e-01,  -4.44444444e-01,  -3.33333333e-01],
                [ -1.11111111e-01,  -4.44444444e-01,  -3.33333333e-01]])
```

6.5 Reševanje predoločenih sistemov

Kadar rešujemo sistem m linearnih enačbami z n neznankami ter velja $m > n$ in je rang $n+1$, imamo predoločeni sistem.

Predoločeni (tudi **nekonsistenten sistem**):

$$\mathbf{A}\,\mathbf{x} = \mathbf{b},$$

nima rešitve. Lahko pa poiščemo najboljši približek rešitve z metodo **najmanjših kvadratov** .

Vsota kvadratov preostankov je definirana s skalarnim produktom:

$$\|r\|^2 = (\mathbf{A}\,\mathbf{x} - \mathbf{b})^T\,(\mathbf{A}\,\mathbf{x} - \mathbf{b})$$

kar preoblikujemo v:

$$\|r\|^2 = \mathbf{x}^T\mathbf{A}^T\mathbf{A}\mathbf{x} - 2\mathbf{b}^T\mathbf{A}\mathbf{x} + \mathbf{b}^T\mathbf{b},$$

kjer smo upoštevali, da zaradi skalarne vrednosti velja: $\mathbf{b}^T\mathbf{A}\mathbf{x} = (\mathbf{b}^T(\mathbf{A}\mathbf{x}))^T = (\mathbf{A}\mathbf{x})^T\mathbf{b}$.

Rešitev enačbe, gradient vsote kvadratov, določa njen minimum:

$$\nabla_x\ \|r\|^2 = 2\mathbf{A}^T\mathbf{A}\,\mathbf{x} - 2\mathbf{A}^T\,\mathbf{b} = 0$$

Tako iz normalne enačbe:

$$\mathbf{A}^T\mathbf{A}\,\mathbf{x} = \mathbf{A}^T\,\mathbf{b}$$

določimo najboljši približek rešitve:

$$\mathbf{x} = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T\,\mathbf{b}.$$

Z vpeljavo **psevdo inverzne matrike**:

$$\mathbf{A}^+ = \left(\mathbf{A}^T\mathbf{A}\right)^{-1}\mathbf{A}^T$$

k matriki $\mathbf{A}$ je rešitev predoločenega sistema zapisana:

$$\mathbf{x} = \mathbf{A}^+\,\mathbf{b}.$$

Psevdo inverzno matriko lahko izračunamo z uporabo funkcij:

- `numpy.linalg.pinv` iz modula `numpy` (dokumentacija[15]),
- Funkcij `pinv`, `pinv2` ali `pinvh` iz modula `scipy.linalg` (izbira je odvisna od obravnavanega problema; glejte dokumentacijo[16]).

Primer sistema z enolično rešitvijo:

```
In [45]: # število enačb enako številu neznank
         A = np.array([[1., 2],
                       [2, 3]])
         b = np.array([5., 8])
         np.linalg.solve(A, b)
```

[15] https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.pinv.html
[16] https://docs.scipy.org/doc/scipy/reference/linalg.html

```
Out[45]: array([ 1.,  2.])
```

Naredimo sedaj predoločeni sistem (funkcija `numpy.vstack` (dokumentacija[17]) sestavi sezname po stolpcih, `numpy.random.seed` (dokumentacija[18]) ponastavi generator naključnih števil na vrednost semena `seed`, `numpy.random.normal` (dokumentacija[19]) pa generira normalno porazdeljeni seznam dolžine `size` in standardne deviacije `scale`):

```
In [46]: vA = np.vstack([A,A,A])
         np.random.seed(seed=0)
         vA += np.random.normal(scale=0.01, size=vA.shape) # pokvarimo rešitev -> sistem je predoločen
         vA # matrika koeficientov

Out[46]: array([[ 1.01764052,  2.00400157],
                [ 2.00978738,  3.02240893],
                [ 1.01867558,  1.99022722],
                [ 2.00950088,  2.99848643],
                [ 0.99896781,  2.00410599],
                [ 2.00144044,  3.01454274]])

In [47]: vb = np.hstack([b,b,b])
         vb += np.random.normal(scale=0.01, size=vb.shape)
         vb # vektor konstant

Out[47]: array([ 5.00761038,  8.00121675,  5.00443863,  8.00333674,  5.01494079,
                 7.99794842])
```

Rešimo sedaj predoločen sistem:

```
In [48]: Ap = np.linalg.pinv(vA)
         Ap.dot(vb)

Out[48]: array([ 0.94512271,  2.02675939])
```

Vidimo, da predoločeni sistem z naključnimi vrednostmi (simulacija šuma pri meritvi) poda podoben rezultat kakor rešitev brez šuma. V kolikor bi nivo šuma povečevali, bi se odstopanje od enolične rešitve povečevalo.

Psevdo inverzno matriko lahko določimo tudi sami in preverimo razliko z vgrajeno funkcijo:

```
In [49]: #%%timeit # preverite hitrost!
         Ap2 = np.linalg.inv(vA.T@vA) @ vA.T

         Ap2 - Ap

Out[49]: array([[  8.88178420e-15,  -8.21565038e-15,   9.65894031e-15,
                  -8.65973959e-15,   1.04360964e-14,  -7.10542736e-15],
                [ -5.66213743e-15,   4.71844785e-15,  -5.66213743e-15,
                   4.49640325e-15,  -6.21724894e-15,   3.55271368e-15]])
```

[17] https://docs.scipy.org/doc/numpy/reference/generated/numpy.vstack.html
[18] https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.seed.html
[19] https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.normal.html

6.6 Iterativne metode

Pogosto se srečamo z velikimi sistemi linearnih enačb, katerih matrika koeficientov ima malo od nič različnih elementov (take matrike imenujemo **redke** ali tudi **razpršene**, angl. *sparse*).

Pri reševanju takih sistemov linearnih enačb se zelo dobro izkažejo iterativne metode; prednosti v primerjavi z direktnimi metodami so:

- računske operacije se izvajajo samo nad neničelnimi elementi (kljub iterativnemu reševanju jih je lahko manj)
- zahtevani spominski prostor je lahko neprimerno manjši.

6.6.1 Gauss-Seidlova metoda

V nadaljevanju si bomo pogledali idejo *Gauss-Seidelove* iterativne metode. Najprej sistem enačb $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ zapišemo kot:

$$\sum_{j=0}^{n-1} A_{ij} x_j = b_i \qquad i = 0, 1, \ldots, n-1.$$

Predpostavim, da smo v $k-1$ koraku iterativne metode in so znani približki $x_j^{(k-1)}$ ($j = 0, 1, \ldots, n-1$). Iz zgornje vsote izpostavimo člen i:

$$A_{ii}\, x_i^{(k-1)} + \sum_{j=0, j\neq i}^{n-1} A_{ij} x_j^{(k-1)} = b_i \qquad i = 0, 1, \ldots, n-1,$$

Ker približki $x_j^{(k-1)}$ ne izpolnjujejo natančno linearnega problema, lahko iz zgornje enačbe določimo nov približek $x_i^{(k)}$:

$$x_i^{(k)} = \frac{1}{A_{ii}} \left(b_i - \sum_{j=0}^{i-1} A_{ij} x_j^{(k)} - \sum_{j=i+1}^{n-1} A_{ij} x_j^{(k-1)} \right)$$

Vsoto smo razdelili na dva dela in za izračun i-tega člena upoštevali v k-ti iteraciji že določene člene z indeksom manjšim od i.

Iterativni pristop prekinemo, ko dosežemo želeno natančnost rešitve ϵ:

$$||x_i^{(k)} - x_i^{(k-1)}|| < \epsilon$$

6.6.2 Zgled

```
In [50]: A = np.array([[8, -1, 1],
                       [-1, 6, -1],
                       [0, -1, 6]], dtype=float)
         b = np.array([-14, 36, 6], dtype=float)
```

Začetni približek:

```
In [51]: x = np.zeros(len(A))
         x

Out[51]: array([ 0.,  0.,  0.])
```

Pripravimo matriko **A** brez diagonalnih elementov (Zakaj? Poskusite odgovoriti spodaj, ko bomo izvedli iteracije.)

```
In [52]: K = A.copy() #naredimo kopijo, da ne povozimo podatkov
         np.fill_diagonal(K, np.zeros(3)) #spremenimo samo diagonalne elemente

In [53]: K

Out[53]: array([[ 0., -1.,  1.],
                [-1.,  0., -1.],
                [ 0., -1.,  0.]])
```

Izvedemo iteracije:

$$x_i^{(k)} = \frac{1}{A_{ii}} \left(b_i - \sum_{j=0}^{i-1} A_{ij} x_j^{(k)} - \sum_{j=i+1}^{n-1} A_{ij} x_j^{(k-1)} \right)$$

(Ker bomo vrednosti takoj zapisali v x, ni treba razbiti vsote na dva dela)

```
In [54]: for k in range(3):
             xk_1 = x.copy()
             print(5*'-' + f'iteracja {k}' + 5*'-')
             for i in range(len(A)): #opazujte kaj se dogaja, ko to celico poženete večkrat!
                 x[i] = (b[i]-K[i,:].dot(x))/A[i,i]
                 print(f'Približek za element {i}', x)
             e = np.linalg.norm(x-xk_1)
             print(f'Norma {e}')

-----iteracja 0-----
Približek za element 0 [-1.75  0.    0.  ]
Približek za element 1 [-1.75        5.70833333  0.        ]
Približek za element 2 [-1.75        5.70833333  1.95138889]
Norma 6.281360365408393
-----iteracja 1-----
Približek za element 0 [-1.28038194  5.70833333  1.95138889]
Približek za element 1 [-1.28038194  6.11183449  1.95138889]
Približek za element 2 [-1.28038194  6.11183449  2.01863908]
Norma 0.6227976321231091
-----iteracja 2-----
Približek za element 0 [-1.23835057  6.11183449  2.01863908]
Približek za element 1 [-1.23835057  6.13004808  2.01863908]
Približek za element 2 [-1.23835057  6.13004808  2.02167468]
Norma 0.04590845211691733
```

Preverimo rešitev

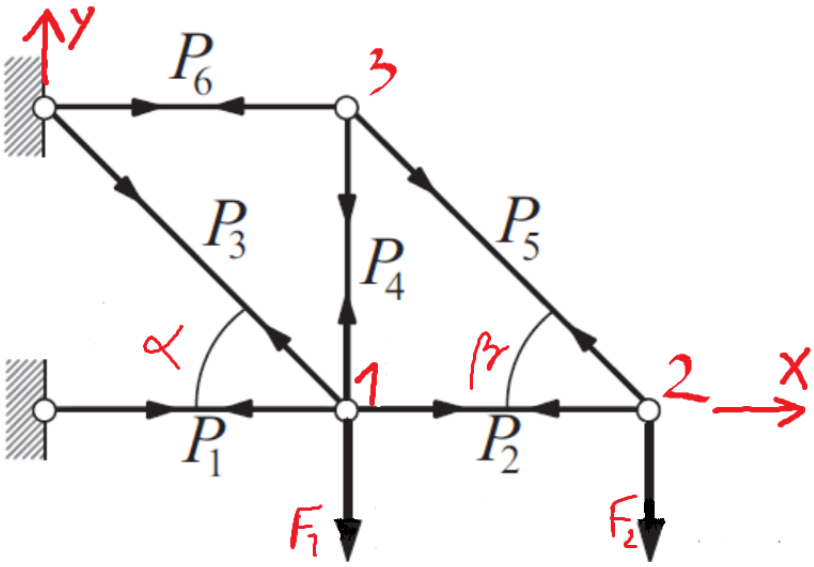
```
In [55]: A@x
```

```
Out[55]: array([-14.01517799,  35.9969644 ,   6.        ])
```

Metoda deluje dobro, če je matrika diagonalno dominantna (obstajajo pa metode, ki delujejo tudi, ko matrika ni diagonalno dominantna, glejte npr.: J. Petrišič, Reševanje enačb, 1996, str 149: Metoda konjugiranih gradientov).

6.7 Nekaj vprašanj za razmislek!

Na sliki je prikazano paličje. Ob delovanju sil F_1 in F_2 se v palicah razvijejo notranje sile P_i. Dimenzije paličja zagotavljata kota α in β.



Sile v palicah izračunamo s pomočjo sistema linearnih enačb.

1. V simbolni obliki zapišite ravnotežje sil za točko 1 v x in y smeri (namig: naloga je posplošitev naloge 15 na strani 81 v knjigi Numerical methods in Eng with Py 3 z nastavkom za rešitev):

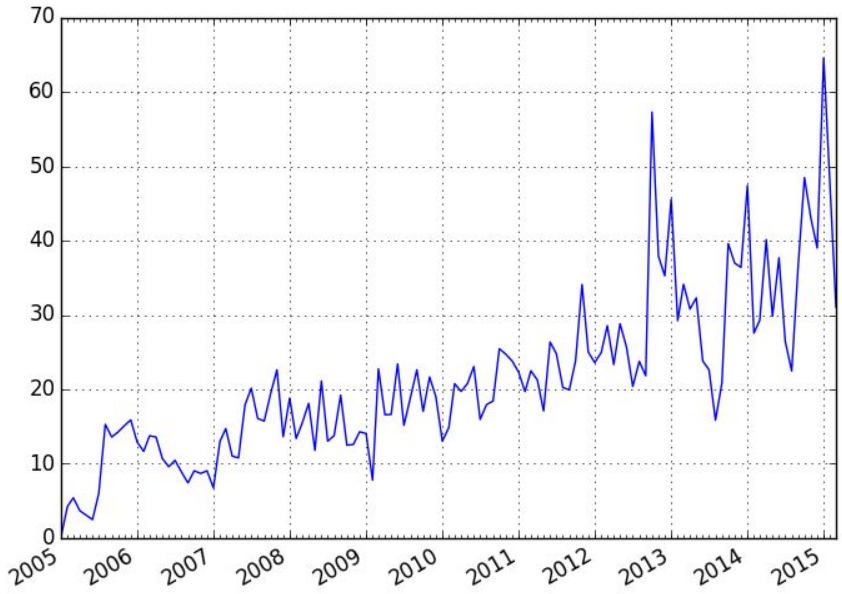
$$\begin{bmatrix} -1 & 1 & -1/\sqrt{2} & 0 & 0 & 0 \\ 0 & 0 & 1/\sqrt{2} & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1/\sqrt{2} & 0 \\ 0 & 0 & 0 & 0 & 1/\sqrt{2} & 0 \\ 0 & 0 & 0 & 0 & 1/\sqrt{2} & 1 \\ 0 & 0 & 0 & -1 & -1/\sqrt{2} & 0 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \\ P_5 \\ P_6 \end{bmatrix} = \begin{bmatrix} 0 \\ 18 \\ 0 \\ 12 \\ 0 \\ 0 \end{bmatrix}$$

   Zgornji nastavek ima napako v predzadnji vrstici. Rešitev sistema v knjigi je: $P_1 = -42000, P_2 = -12000, P_3 = 42426, P_4 = -12000, P_5 = 16971, P_6 = 12000$).
2. V simbolni obliki zapišite ravnotežje sil za točko 2 v x in y smeri.
3. Najdite simbolno rešitev za sile P_i.
4. Uporabite podatke: $\alpha = \beta = \pi/4$, $F_1 = 18$ kN in $F_2 = 12$ kN ter najdite številčno rešitev.

5. Pripravite si funkcijo, ki bo za poljubne podatke (npr: `podatki = {a: pi/4, b: pi/4, F1: 18000, F2: 12000}`) vrnila numerično matriko koeficientov **A** in vektor konstant **b**. Če ne uspete tega narediti avtomatizirano, delajte "na roke"(splača se vam potruditi, saj bomo to večkrat rabili).
6. Razširite zgornjo funkcijo, da vam vrne rešitev linearnega sistema (uporabite kar `numpy` knjižnico)
7. Predpostavite $F_1 = F_2 = 10$ kN. V vsaj petih vrednostih kota $\alpha = \beta$ od 10° do 80° izračunajte sile v palicah.
8. Za primer iz predhodne naloge narišite sile v palicah.
9. S pomočjo funkcije `np.linalg.solve` izračunajte inverz poljubne matrike `A` (nato izračunajte še inverz s pomočjo funkcije `np.linalg.inv`).
10. Na primeru poljubnih podatkov (npr: `podatki = {a: pi/4, b: pi/4, F1: 18000, F2: 12000}`) pokažite Gaussovo eliminacijo z delnim pivotiranjem.
11. Na primeru poljubnih podatkov (npr: `podatki = {a: pi/4, b: pi/4, F1: 18000, F2: 12000}`) pokažite Gauss-Seidlov iterativni pristop k iskanju rešitve.

6.7.1 Dodatno

Analizirajte, koliko e-mailov dobite na dan:



https://plot.ly/ipython-notebooks/graph-gmail-inbox-data/ (Nasvet: sledite kodi in uporabite `matplotlib` za prikaz).

Poglavje 7

Interpolacija

7.1 Uvod

Pri **interpolaciji** izhajamo iz tabele (različnih) vrednosti x_i, y_i:

x	y
x_0	y_0
x_1	y_1
...	...
x_{n-1}	y_{n-1}

določiti pa želimo vmesne vrednosti. Če želimo določiti vrednosti zunaj območja x v tabeli, govorimo o **ekstrapolaciji**.

V okviru **interpolacije** (angl. *interpolation*) točke povežemo tako, da predpostavimo neko funkcijo in dodamo pogoj, da funkcija *mora* potekati skozi podane točke.

Pri **aproksimaciji** (angl. *approximation* ali tudi *curve fitting*) pa predpostavimo funkcijo, ki se čimbolj (glede na izbrani kriterij) prilega podatkom.

Poglejmo si primer:

x	y
1.0	0.54030231
2.5	-0.80114362
4.0	-0.65364362

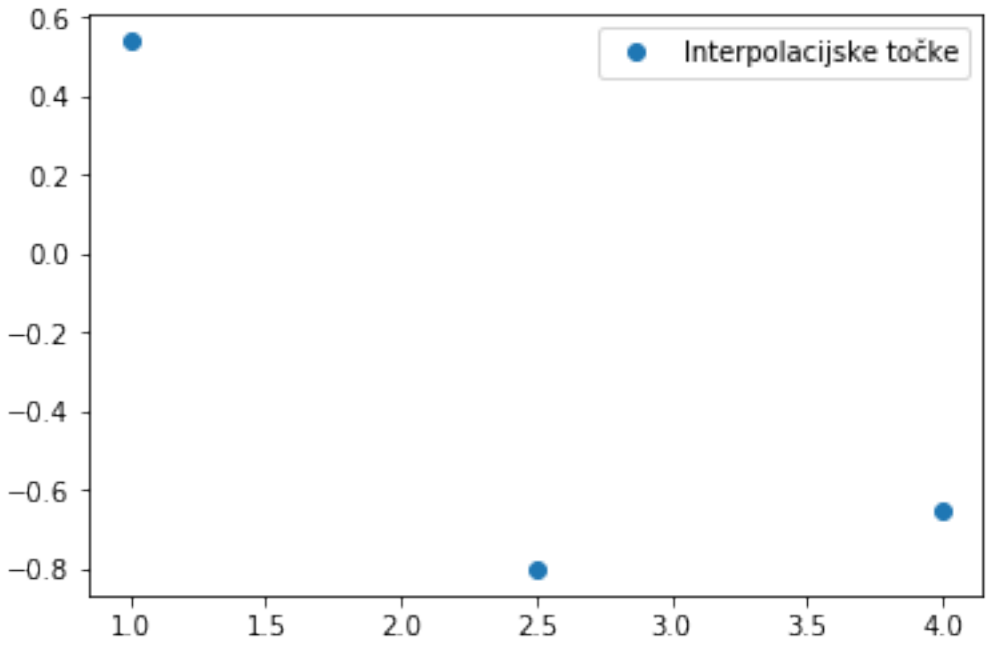
Pri interpolaciji izhajamo iz tabele vrednosti. Da bomo pozneje lahko enostavno prikazali napako, smo zgornjo tabelo generirali s pomočjo izraza $y = \cos(x)$!

Pripravimo numerični zgled; najprej uvozimo pakete:

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

Nato pripravimo tabelo ter prikaz:

```
In [2]: n = 3
        x = np.linspace(1, 4, n)
        f = np.cos # posplošimo interpolirano funkcijo (lahko spremenite v drugo funkcijo)
        f_ime = f.__str__().split('\'')[1] # avtomatsko vzamemo ime funkcije
        y = f(x)
        plt.plot(x, y, 'o', label='Interpolacijske točke');
        plt.legend();
```



7.2 Interpolacija s polinomom

Interpolacijo s polinomom se zdi najbolj primerna, saj je enostaven!

Polinom stopnje $n-1$:

$$y = a_0\, x^{n-1} + a_1\, x^{n-2} + \cdots + a_{n-2}\, x + a_{n-1}.$$

je definiran z n konstantami a_i. Da določimo n konstant, potrebujemo n (različnih) enačb. Za vsak par x_i, y_i lahko torej zapišemo:

$$y_i = a_0\, x_i^{n-1} + a_1\, x_i^{n-2} + \cdots + a_{n-2}\, x_i + a_{n-1}.$$

Ker imamo podanih n parov, lahko določimo n neznanih konstant a_i, ki definirajo polinom stopnje $n-1$. Sistem n linearnih enačb lahko zapišemo:

$$\begin{bmatrix} x_0^{n-1} & x_0^{n-2} & \dots & x_0^0 \\ x_1^{n-1} & x_1^{n-2} & \dots & x_1^0 \\ & & \vdots & \\ x_{n-1}^{n-1} & x_{n-1}^{n-2} & \dots & x_{n-1}^0 \end{bmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Sistem linearnim enačb zapišemo v obliki:

$$\mathbf{M}\,\mathbf{a} = \mathbf{b}$$

Definirajmo matriko koeficientov **M**:

```
In [3]: M = np.asarray([[_**p for p in reversed(range(len(x)))] for _ in x])
        M

Out[3]: array([[  1.  ,   1.  ,   1.  ],
               [  6.25,   2.5 ,   1.  ],
               [ 16.  ,   4.  ,   1.  ]])
```

Izračunamo koeficiente $a_0, a_1, \ldots$:

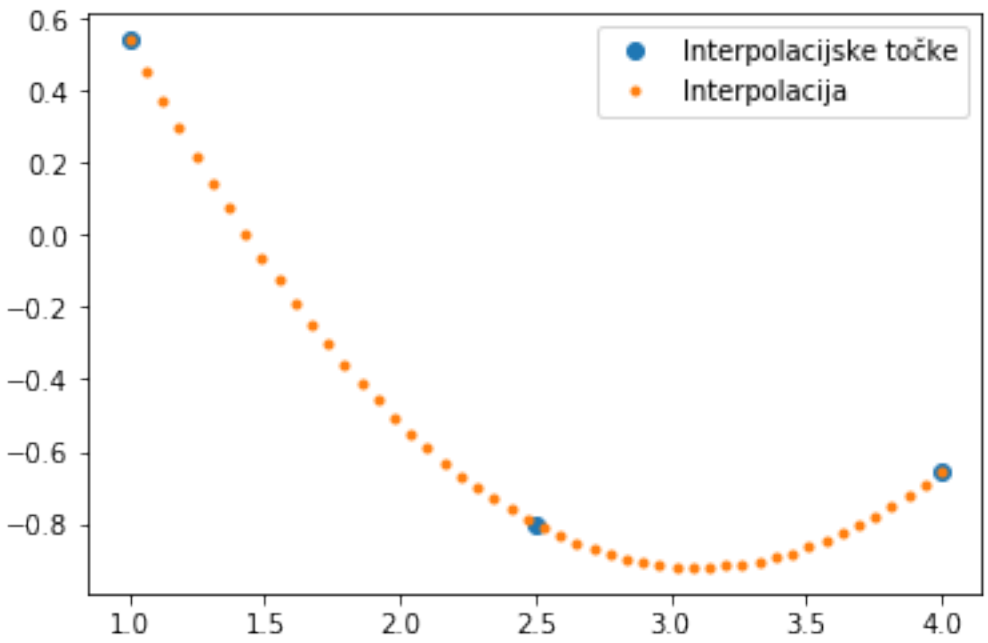
```
In [4]: resitev = np.linalg.solve(M, y)
        resitev

Out[4]: array([ 0.33087687, -2.05236633,  2.26179176])
```

Pripravimo interpolacijski polinom kot Pythonovo funkcijo:

```
In [5]: def y_function(x, resitev):
            A = np.asarray([[_**p for p in reversed(range(len(resitev)))] for _ in x])
            return A.dot(resitev)
```

Izris interpolacijskega polinoma pri bolj gosti mreži točk:

```
In [6]: xint = np.linspace(np.min(x), np.max(x), 50)
        yint = y_function(xint, resitev)
        plt.plot(x, y, 'o', label='Interpolacijske točke')
        plt.plot(xint, yint, '.',label='Interpolacija')
        plt.legend();
```



Slabosti zgornjega postopka so:

- število numeričnih operacij raste sorazmerno z n^3,
- problem je lahko slabo pogojen (z večanjem stopnje polinoma slaba pogojenost naglo narašča):

```
In [7]: np.linalg.cond(M)
```

```
Out[7]: 71.302278703110773
```

Navodilo: vrnite se par vrstic nazaj in spremenite število interpolacijskih točk n na višjo vrednost (npr. 10).

7.3 Lagrangeva metoda

Lagrangeva metoda ne zahteva reševanja sistema enačb in je s stališča števila računskih operacij (narašča sorazmerno z n^2 (vir[1])) boljša od predhodno predstavljene polinomske interpolacije (število operacij narašča sorazmerno z n^3), kjer smo reševali sistem linearnih enačb. Rešitev pa je seveda popolnoma enaka!

Lagrangev interpolacijski polinom stopnje $n-1$ je definiran kot:

$$P_{n-1}(x) = \sum_{i=0}^{n-1} y_i \, l_i(x),$$

kjer je l_i Lagrangev polinom:

$$l_i(x) = \prod_{j=0, j\neq i}^{n-1} \frac{x - x_j}{x_i - x_j}.$$

Poglejmo si interpolacijo za zgoraj prikazane x in y podatke.

Definirajmo najprej Lagrangeve polinome $l_i(x) = \prod_{j=0,j\neq i}^{n-1} \frac{x-x_j}{x_i-x_j}$:

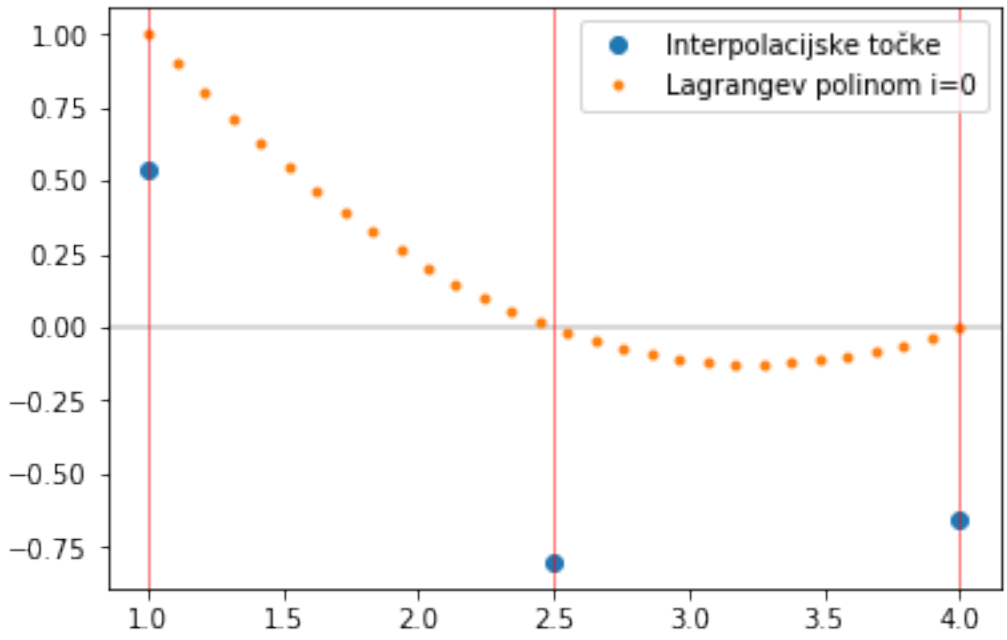
```
In [8]: def lagrange(x, x_int, i):
            """ Vrne vrednosti i-tega Lagrangevega polinoma

            x: neodvisna spremenljivka (skalar ali numerično polje)
            x_int: seznam interpolacijskih točk
            i: indeks polinoma
            """
            Lx = 1.0
            for j in range(len(x_int)):
                if j != i:
                    Lx *= (x-x_int[j]) / (x_int[i]-x_int[j])
            return Lx
```

```
In [9]: def slika(i=0):
            xint = np.linspace(np.min(x), np.max(x), 30)
            plt.plot(x, y, 'o', label='Interpolacijske točke')
```

[1] http://www.ams.org/journals/mcom/1970-24-109/S0025-5718-1970-0258240-X/S0025-5718-1970-0258240-X.pdf

```
            plt.axhline(0, color='k', linewidth=0.3);
            plt.plot(xint, lagrange(xint, x_int=x, i=i), '.',label=f'Lagrangev polinom i={i}');
            for _ in x:
                plt.axvline(_, color='r', linewidth=0.5);
            plt.legend()
            plt.show()
```

```
In [10]: slika(i=0)
```



Opazimo, da ima i-ti Lagrangev polinom v x_i vrednost 1, v ostalih podanih točkah pa nič!

Če torej Lagrangev polinom za $i = 0$ pomnožimo z y_0, bomo pri $x = x_0$ dobili pravo vrednost, v ostalih interpolacijskih točkah pa nič; implentirajmo torej Lagrangev interpolacijski polinom:

$$P_{n-1}(x) = \sum_{i=0}^{n-1} y_i \, l_i(x),$$

```
In [11]: def lagrange_interpolacija(x, x_int, y_int):
             """ Vrne vrednosti Lagrangeve interpolacije

             x: neodvisna spremenljivka (skalar ali numerično polje)
             x_int: abscisa interpolacijskih točk
             y_int: ordinata interpolacijskih točk
             """
             y = 0.
             for i in range(len(x_int)):
                 Lx = 1.0
                 for j in range(len(x_int)):
                     if j != i:
```

```
                Lx *= (x-x_int[j]) / (x_int[i]-x_int[j])
            y += y_int[i] * Lx
        return y
```

Pripravimo sliko:

```
In [12]: def slika(i=0):
             xint = np.linspace(np.min(x), np.max(x), 30)
             plt.plot(x, y, 'o', label='Interpolacijske točke')
             plt.plot(xint, lagrange(xint, x_int=x, i=i), '.',label=f'Lagrangev polinom i={i}');
             plt.plot(xint, lagrange_interpolacija(xint, x_int=x, y_int=y), '.',label=f'Lagrangev int po
             plt.axhline(0, color='k', linewidth=0.3);
             for _ in x:
                 plt.axvline(_, color='r', linewidth=0.5);
             plt.legend()
             plt.show()
```

Iz `ipywidgets` uvozimo `interact`, ki je močno orodje za avtomatsko generiranje (preprostega) uporabniškega vmesnika znotraj `jupyter` okolja. Tukaj bomo uporabili relativno preprosto interakcijo s sliko; za pregled vseh zmožnosti pa radovednega bralca naslavljamo na dokumentacijo[2].

Uvoz funkcije `interact`

```
In [13]: from ipywidgets import interact
```

```
In [14]: interact(slika, i=(0,len(x)-1,1));
```

```
A Jupyter Widget
```

Iz slike vidimo, da ima Lagrangev polinom i samo pri x_i vrednost 1 v ostalih točkah $\neq i$ pa ima vrednosti nič; ko Lagrangev polinom $l_i(x)$ pomnožimo z y_i zadostimo i-ti točki iz tabele. Posledično Lagrangeva interpolacija z vsoto lagrangevih polinomov interpolira tabelo.

Polinomska interpolacija pri velikem številu točk je lahko slabo pogojena naloga in zato jo odsvetujemo.

7.3.1 Ocena napake

Če je $f(x)$ funkcija, ki jo interpoliramo in je $P_{n-1}(x)$ interpolacijski polinom stopnje $n-1$, potem se lahko pokaže (glejte npr.: Burden, Faires, Burden: Numerical Analysis), da je napaka interpolacije s polinomom:

$$e = f(x) - P_{n-1}(x) = \frac{f^{(n)}(\xi)}{n!}(x-x_0)(x-x_1)\cdots(x-x_{n-1}),$$

kjer je $f^{(n)}$ odvod funkcije, $n-1$ stopnja interpolacijskega polinoma in ξ vrednost na interpoliranem intervalu $[x_0, x_{n-1}]$.

7.3.2 Zgled

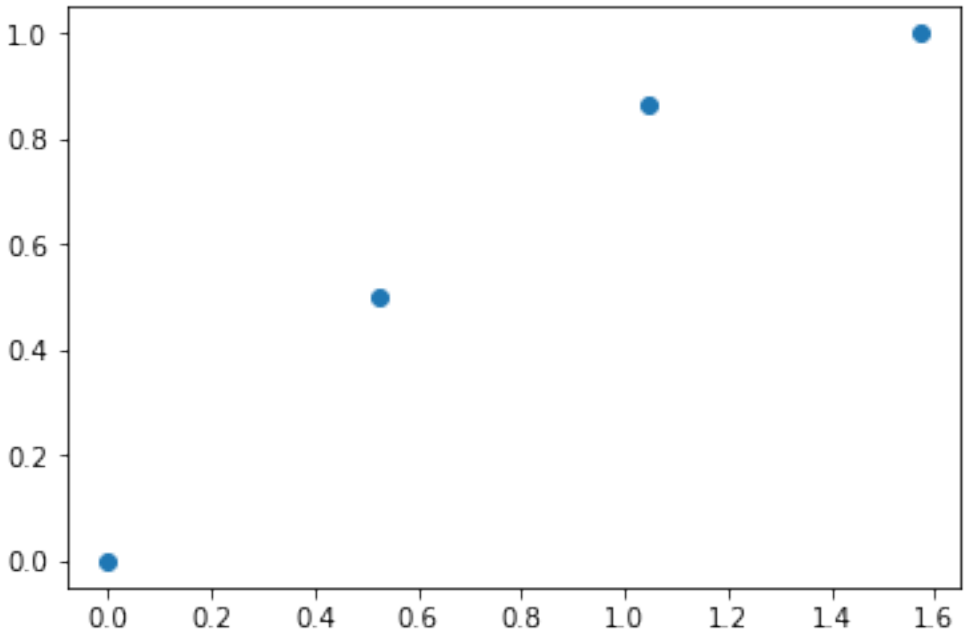
Tukaj si bomo ogledali interpolacijo točk:

[2] http://ipywidgets.readthedocs.io/en/latest/examples/Using%20Interact.html

```
In [15]: x = np.array([0.        ,  0.52359878,  1.04719755,  1.57079633])
         y = np.array([ 0.       ,  0.5       ,  0.8660254,  1.        ])
```

Točke prikažimo:

```
In [16]: plt.plot(x, y, 'o');
```



Linearna interpolacija za vrednost pri x=1.57079633/2:

```
In [17]: y_linearna = lagrange_interpolacija(x=x[-1]/2, x_int=x[1:3], y_int=y[1:3])
         y_linearna
```

```
Out[17]: 0.68301270000000003
```

Kvadratna:

```
In [18]: y_kvadratna = lagrange_interpolacija(x=x[-1]/2, x_int=x[0:3], y_int=y[0:3])
         y_kvadratna
```

```
Out[18]: 0.69975952364641758
```

Kubična

```
In [19]: y_kubična = lagrange_interpolacija(x=x[-1]/2, x_int=x, y_int=y)
         y_kubična
```

```
Out[19]: 0.70588928684463403
```

7.3.3 Zgled ocene napake

Pri interpolaciji ponavadi funkcije $f(x)$ ne poznamo; zgoraj interpolirane točke pa pripadajo funkciji $y = f(x) = \sin(x)$, vendar pa ne poznamo ξ in zato napako ocenimo s pomočjo formule:

$$e = \frac{f^{(n)}(\xi)}{n!}(x - x_0)(x - x_1)\cdots(x - x_n)$$

Ker je v primeru linearne aproksimacije ($n = 2$) drugi odvod sinusne funkcije ($f^{(n)}$) med -1 in +1, velja:

$$|e| \leq \left|\frac{-1}{2!}(\pi/4 - \pi/6)(\pi/4 - \pi/3)\right| = \frac{1}{2}\frac{\pi}{12}\frac{\pi}{12} = \frac{\pi^2}{288} = 0,033$$

Poleg **Lagrangeve metode** bi si tukaj lahko pogledali še **Newtonovo metodo** interpolacije.

7.3.4 Interpolacija z uporabo `scipy`

Poglejmo si interpolacijo v okviru modula `scipy.interpolate` (dokumentacija[3]).

Uporabili bomo funkcijo za interpoliranje tabele z zlepki, `scipy.interpolate.interp1d` (dokumentacija[4]):

```
interp1d(x, y, kind='linear', axis=-1, copy=True, bounds_error=None, fill_value=nan, assume_sorted=Fals
```

Podati moramo vsaj dva parametra: seznama interpolacijskih točk `x` in `y`. Privzeti parameter `kind='linear'` pomeni, da inteproliramo z odsekoma linearno funkcijo. `interp1d` vrne funkcijo `f`, ki jo kličemo (npr. `y = f(x)`) za izračun interpolirane vrednosti.

Parameter `kind` je lahko npr. tudi: `'zero'`, `'slinear'`, `'quadratic'` in `'cubic'`; takrat se uporabi interpolacijski zlepek (ang. *spline*) reda 0, 1, 2 oz. 3. Zlepke si bomo pogledali v naslednjem poglavju.

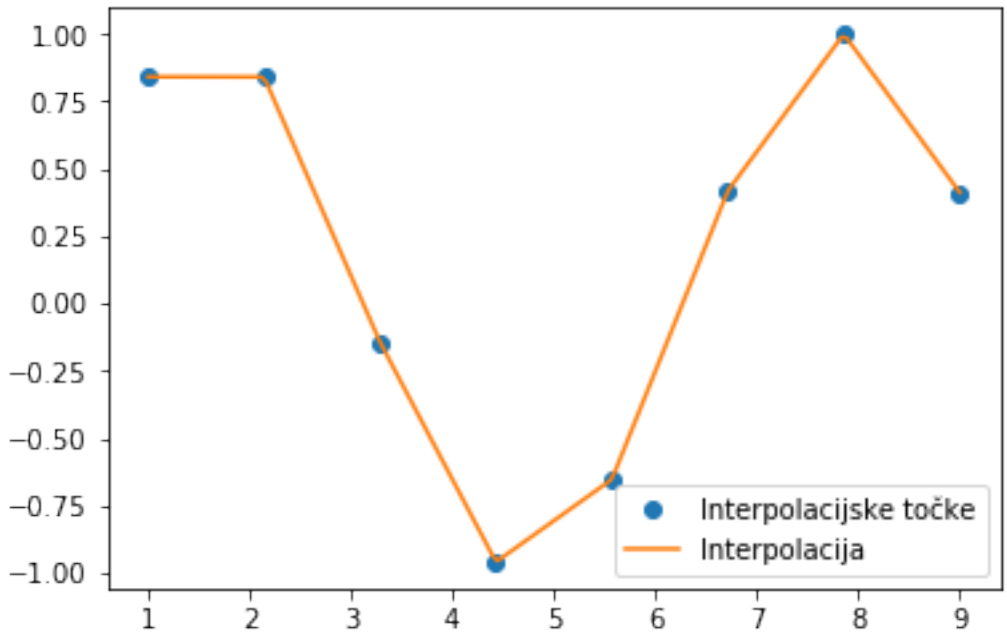
```
In [20]: from scipy.interpolate import interp1d
```

Definirajmo tabelo podatkov:

```
In [21]: x = np.array([ 1.        ,  2.14285714,  3.28571429,  4.42857143,  5.57142857,
                 6.71428571,  7.85714286,  9.        ])

         y = np.array([ 0.84147098,  0.84078711, -0.14362322, -0.95999344, -0.65316501,
                 0.41787078,  0.999995  ,  0.41211849])

In [22]: len(x)

Out[22]: 8

In [23]: f = interp1d(x, y, kind='linear')
         x_g = np.linspace(x[0], x[-1], 20*len(x)-1)
         plt.plot(x, y, 'o', label='Interpolacijske točke')
         plt.plot(x_g, f(x_g), '-', label='Interpolacija')
         plt.legend();
```

[3] https://docs.scipy.org/doc/scipy/reference/interpolate.html

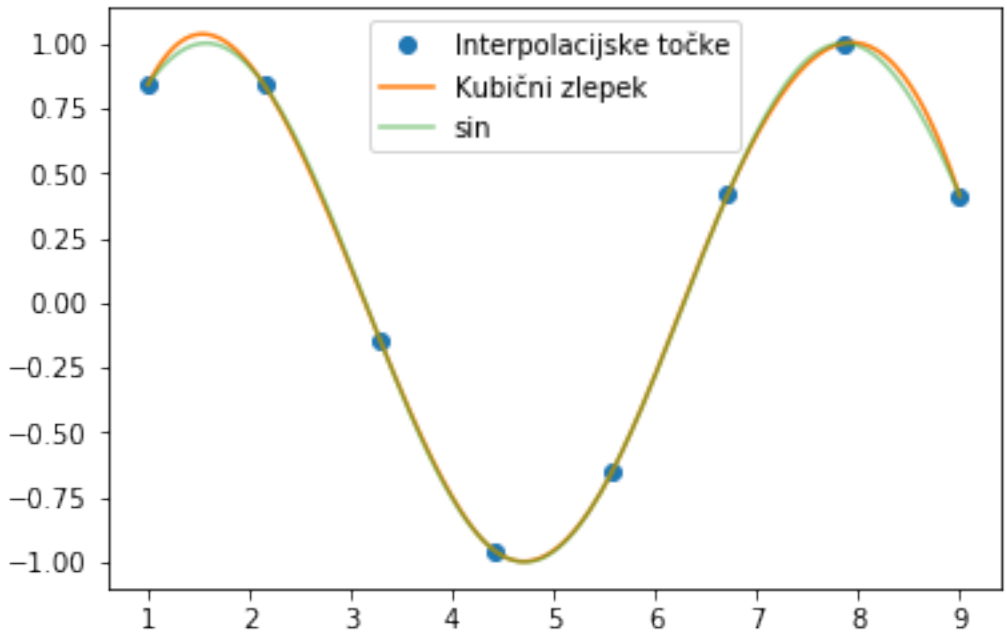
[4] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.interp1d.html#scipy.interpolate.interp1d



7.4 Kubični zlepki

Preden gremo v teorijo zlepkov, si poglejmo rezultat, ki ga dobimo s klicanjem funkcije `interp1d` s parametrom `kind='cubic'` (rezultat je kubični zlepek).

```
In [24]: f = interp1d(x, y, kind='cubic')
         plt.plot(x, y, 'o', label='Interpolacijske točke')
         plt.plot(x_g, f(x_g), '-', label='Kubični zlepek')
         plt.plot(x_g, np.sin(x_g), label='sin', alpha=0.5)
         plt.legend();
```



Kubični zlepki so pogost način interpolacije.

Zahtevamo, da je: $x_0 < x_1 < \cdots < x_n$.

Od točke x_i do x_{i+1} naj bo zlepek polinom:

$$f_{i,i+1}(x) = a_{i,3}\,x^3 + a_{i,2}\,x^2 + a_{i,1}\,x + a_{i,0},$$

pri čemer so neznane vrednosti konstant $a_{i,j}$.

Če imamo na primer $n+1$ točk, potem je treba določiti n polinomov.

Celotni zlepek čez $n+1$ točk je definiran z:

$$f(x) = \begin{cases} f_{0,1}(x); & x \in [x_0, x_1) \\ f_{1,2}(x); & x \in [x_1, x_2) \\ \vdots & \\ f_{n-1,n}(x); & x \in [x_{n-1}, x_n] \end{cases}$$

Vsak polinom $f_{i,i+1}$ je definiran s 4 konstantami $a_{i,j}$; skupaj torej moramo izračunati $4n$ konstant $a_{i,j}$.

Kako določimo konstante $a_{i,j}$?

Za določitev $4n$ neznak potrebujemo $4n$ enačb. Poglejmo si, kako jih dobimo:

- n enačb dobimo iz interpolacijskega pogoja:

  $$y_i = f_{i,i+1}(x_i), \quad i = 0, 1, 2, \ldots, n-1$$

- 1 enačbo iz zadnje točke:

  $$y_n = f_{n-1,n}(x_n)$$

- $3(n-1)$ enačb dobimo iz pogoja C^2zveznosti:

$$\lim_{x\to x_i^-} f(x) = \lim_{x\to x_i^+} f(x),$$

$$\lim_{x\to x_i^-} f'(x) = \lim_{x\to x_i^+} f'(x)$$

in

$$\lim_{x\to x_i^-} f''(x) = \lim_{x\to x_i^+} f''(x).$$

Skupaj imamo definiranih $4n-2$ enačbi, manjkata torej še dve!

Različni tipi zlepkov se ločijo po tem, kako ti dve enačbi določimo. V nadaljevanju si bomo pogledali *naravne kubične zlepke*.

7.4.1 Naravni kubični zlepki

Naravni kubični zlepki temeljijo na ideji Eulerjevega nosilca:

$$E\,I\,\frac{\mathrm{d}^4 y}{\mathrm{d}x^4} = q(x),$$

kjer je E elastični modul, I drugi moment preseka in $q(x)$ zunanja porazdeljena sila. Ker zunanje porazdeljene sile ni ($q(x)=0$), velja:

$$E\,I\,\frac{\mathrm{d}^4 y}{\mathrm{d}x^4} = 0.$$

Sledi, da lahko v vsaki točki tanek nosilec popišemo s polinomom tretje stopnje.

C^2 zveznost je zagotovljena v kolikor so vmesne podpore nosilca členki (moment zato nima nezvezne spremembe).

Manjkajoči 2 neznanki pri naravnih kubičnih zlepkih določimo iz pogoja, da je moment na koncih enak nič (členkasto vpetje):

$$f''(x_0) = 0 \qquad \text{in} \qquad f''(x_n) = 0$$

Izpeljava je natančneje prikazana v knjigi Kiusalaas J: Numerical Methods in Engineering with Python 3, 2013, stran 120 (glejte tudi J. Petrišič: Interpolacija, Fakulteta za strojništvo, 1999); podrobna izpeljava presega obseg te knjige.

Tukaj si bomo pogledali samo končni rezultat, ki ga lahko izpeljemo ob zgornjih pogojih. V primeru ekvidistantne delitve $h = x_{i+1} - x_i$ tako izpeljemo sistem enačb ($i = 1, \ldots, n-1$):

$$k_{i-1} + 4k_i + k_{i+1} = \frac{6}{h^2}\left(y_{i-1} - 2y_i + y_{i+1}\right).$$

kjer je neznanka k_i drugi odvod odsekovne funkcije $k_i = f''_{i,i+1}(x_i)$.

Rešljiv sistem enačb dobimo, če dodamo še robna pogoja za naravne kubične zlepke:

$$k_0 = k_n = 0.$$

Ko določimo neznake k_i, jih uporabimo v odsekoma definirani funkciji:

$$f_{i,i+1}(x) = \frac{k_i}{6}\left(\frac{(x-x_{i+1})^3}{h} - (x-x_{i+1})\,h\right) - \frac{k_{i+1}}{6}\left(\frac{(x-x_i)^3}{h} - (x-x_i)\,h\right) + \frac{y_i\,(x-x_{i+1}) - y_{i+1}\,(x-x_i)}{h}.$$

7.4.2 Numerična implementacija

Najprej pripravimo funkcijo, katera za podane interpolacijske točke reši sistem linearnih enačb in vrne koeficiente k_i:

```
In [25]: def kubicni_zlepki_koeficient(x, y):
             """ Vrne koeficiente kubičnih zlepkov `k`, matriko koeficientov `A` in konstant.

             x in y predstavljata seznam znanih vrednosti; x mora biti ekvidistanten.
             """
             n = len(x)
             A = np.zeros((n, n)) # pripravimo matriko koeficientov
             h = x[1]-x[0] # korak h
             for i in range(n):
                 if i==0 or i==n-1:
                     A[i,i] = 1. # k_0 in k_n sta nič zato tukaj damo 1
                                 # pri vektorju konstant pa bomo dali 0, k_0 in k_n bosta torej 0
                 else:
                     A[i, i-1:i+2] = np.asarray([1., 4., 1.])
             b = np.zeros(n)
             b[1:-1] = (6/h**2)*(y[:-2] - 2*y[1:-1] + y[2:]) # desna stran zgornje enačbe
             k = np.linalg.solve(A,b)
             return k, A, b
```

Opomba: pri zgornjem linarnem problemu, lahko izračun zelo pohitrimo, če upoštevamo tridiagonalnost matrike koeficientov!

Poglejmo si primer izračuna koeficientov:

```
In [26]: x = np.asarray([1, 2, 3, 4, 5])
         y = np.asarray([0, 1, 0, 1, 0])

         k, A, b = kubicni_zlepki_koeficient(x, y)
         print('Matrika koeficientov A lin. sistema:\n', A)
         print('Vektor konstant b lin. sistema:     ', b)
         print('Koeficienti k so:', k)
```

```
Matrika koeficientov A lin. sistema:
 [[ 1.  0.  0.  0.  0.]
 [ 1.  4.  1.  0.  0.]
 [ 0.  1.  4.  1.  0.]
 [ 0.  0.  1.  4.  1.]
 [ 0.  0.  0.  0.  1.]]
Vektor konstant b lin. sistema:      [  0. -12.  12. -12.   0.]
Koeficienti k so: [ 0.         -4.28571429  5.14285714 -4.28571429  0.        ]
```

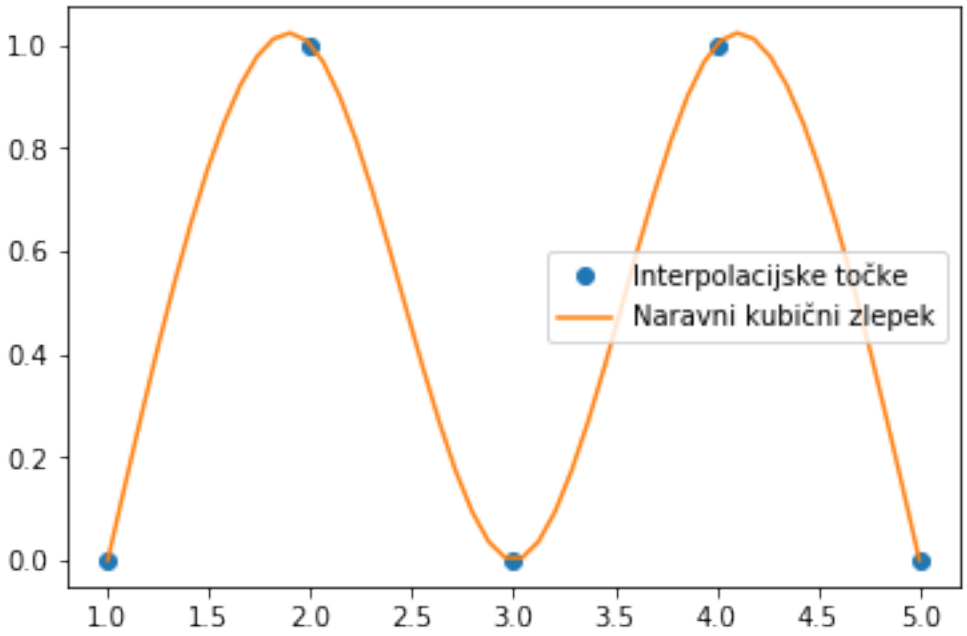
Nato potrebujemo še kubični polinom v določenem intervalu; implementirajmo izraz:

$$f_{i,i+1}(x) = \frac{k_i}{6}\left(\frac{(x-x_{i+1})^3}{h} - (x-x_{i+1})\,h\right) - \frac{k_{i+1}}{6}\left(\frac{(x-x_i)^3}{h} - (x-x_i)\,h\right) + \frac{y_i\,(x-x_{i+1}) - y_{i+1}\,(x-x_i)}{h}$$

```
In [27]: def kubicni_zlepki(k, x, y, x_najdi):
             """ Vrne kubicni zlepek pri delitvi `xint`.

             :param k: koeficienti kubičnih zlepkov
             :param x in y: znane vrednosti, x mora biti ekvidistanten
             :param x_najdi: vrednosti kjer želimo izračunati kubični zlepek
             """
             h = x[0] - x[1]
             i = int((x_najdi-x[0])//(-h))
             if i >= len(k)-1:
                 i = len(k)-2
             out = ((x_najdi - x[i+1])**3/h - (x_najdi - x[i+1])*h)*k[i]/6.0 \
                 - ((x_najdi - x[i])**3/h - (x_najdi - x[i])*h)*k[i+1]/6.0 \
                 + (y[i]*(x_najdi - x[i+1]) \
                 - y[i+1]*(x_najdi - x[i]))/h
             return out
```

Izračunamo interpolirane vrednosti:

```
In [28]: xint = np.linspace(np.min(x), np.max(x), 50)
         yint = np.asarray([kubicni_zlepki(k, x, y, _) for _ in xint])

In [29]: plt.plot(x, y, 'o', label='Interpolacijske točke')
         plt.plot(xint, yint, label='Naravni kubični zlepek')
         plt.legend();
```



7.5 Nekaj vprašanj za razmislek!

1. Preštudirajte Lagrangevo polinomsko interpolacijo in pripravite funkcijo za Lagrangeve polinome. Pojasnite (z grafičnim prikazom) Lagrangeve polinome.
2. Definirajte funkcijo za Lagrangevo polinomsko interpolacijo. Na primeru pojasnite, kako deluje.
3. Pojasnite teoretično ozadje naravnih kubičnih zlepkov.
4. Naravne kubične zlepke smo izpeljali pod pogojem, da momenta na koncu ni; včasih želimo drugačne pogoje na koncih (npr. znani naklon ali znani moment). Modificirajte na predavanjih predstavljeno kodo za primer, da je na koncih moment $\neq 0$ (predpostavite neko numerično vrednost).
5. Podatke:

   ```
   x = np.linspace(0, 10, 10)
   y = np.random.rand(10)-0.5
   ```

   interpolirajte z uporabo `scipy.InterpolatedUnivariateSpline`. Podatke prikažite.
6. Za zgoraj definirane podatke preučite pomoč in najdite vse ničle. Prikažite jih na predhodni sliki.
7. Za zgoraj definirani zlepek izračunajte prvi odvod in ga prikažite.
8. Za zgoraj definirani zlepek izračunajte določeni integral od začetka do konca.
9. Za zgoraj definirane podatke z uporabo vgrajenih funkcij prikažite izračun linearnega in kvadratnega zlepka. Prikažite na sliki.
10. Preučite pomoč za funkcijo `scipy.interpolate.lagrange` in k predhodni sliki dodajte Lagrangev interpolacijski polinom. Komentirajte rezultate.
11. Preučite pomoč za funkcijo `scipy.interpolate.interp1d` in k predhodni sliki dodajte kvadratni zlepek.
12. Preučite pomoč za funkcijo `scipy.interpolate.BarycentricInterpolator` in pojasnite ter prikažite njeno prednost.
13. Preučite pomoč za funkcijo `scipy.interpolate.KroghInterpolator` in pojasnite njeno prednost.

7.5.1 Dodatno

- 2D interpolacija: https://www.youtube.com/watch?v=_cJLVhdj0j4
- Strojno prevajanje: https://pypi.python.org/pypi/goslate)

7.5.2 Nekaj komentarjev modula `scipy.interpolate`

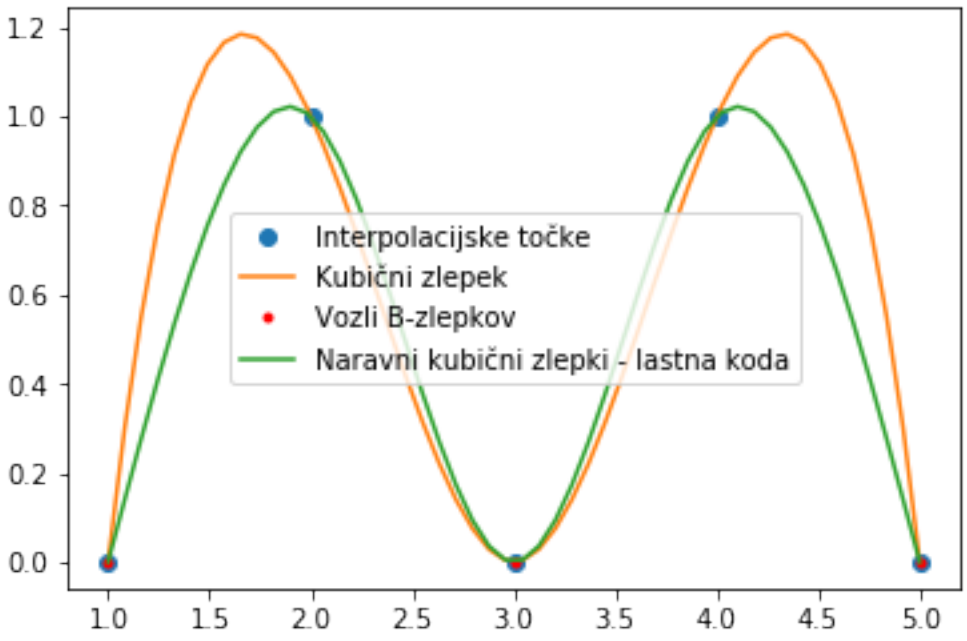
SciPy ima implementiranih večje število različnih interpolacij (glejte dokumentacijo[5]). S stališča uporabe se bomo tukaj dotaknili objektne implementacije `scipy.interpolate.InterpolatedUnivariateSpline` (dokumentacija[6]) (starejši pristop temelji na funkcijskem programiranju, glejte dokumentacijo[7] `scipy.interpolate.splrep`):

[5] https://docs.scipy.org/doc/scipy/reference/interpolate.html
[6] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.InterpolatedUnivariateSpline.html
[7] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.splrep.html

```
InterpolatedUnivariateSpline(x, y, w=None, bbox=[None, None], k=3, ext=0, check_finite=False)
```

Pri inicializaciji objekta `InterpolatedUnivariateSpline` moramo posredovati interpolacijske točke `x` in `y`. Argument `k` s privzeto vrednostjo `k=3` definira red intepolacijskega zlepka (`1<=k<=5`). Pomemben opcijski parameter je tudi `w`, ki definira uteži posameznim interpolacijskim točkam (uporabimo ga, če želimo določenim področjem dati večji poudarek).

```
In [30]: from scipy.interpolate import InterpolatedUnivariateSpline

In [31]: spl = InterpolatedUnivariateSpline(x, y, k=3) # poglejte opcije!
         plt.plot(x, y, 'o', label='Interpolacijske točke')
         plt.plot(xint, spl(xint), label='Kubični zlepek');
         plt.plot(spl.get_knots(), spl(spl.get_knots()), 'r.', label='Vozli B-zlepkov')
         plt.plot(xint, yint, label='Naravni kubični zlepki - lastna koda');
         plt.legend();
```



Ker gre za B-zlepke, je rezultat drugačen kot tisti, ki smo ga izpeljali z naravnimi kubičnimi zlepki. V nasprotju z naravnimi kubičnimi zlepki, ki imajo vozle (angl. *knots*) v interpolacijskih točkah, se vozli B-zlepkov prilagodijo podatkom. V konkretnem primeru so vozli v točkah:

```
In [32]: spl.get_knots()

Out[32]: array([ 1.,  3.,  5.])
```

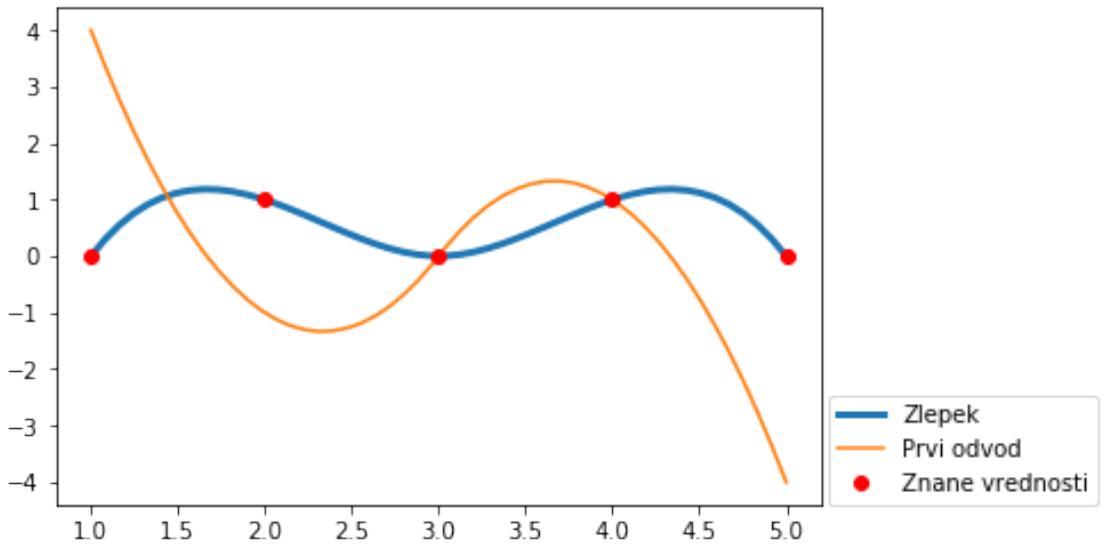
7.5.3 Odvajanje, integriranje ... zlepkov

Zlepke lahko odvajamo in integriramo, saj so polinomi. Objekt `InterpolatedUnivariateSpline` je tako že pripravljen za odvajanje, integriranje, iskanje korenov (ničel), vozlov ... (glejte dokumentacijo[8]).

[8] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.InterpolatedUnivariateSpline.html

Za prvi odvod zlepka v objektu `spl` na primer uporabimo metodo `spl.derivative(1)`, ki vrne nov objekt zlepka (njen red je sedaj za 1 nižji):

```
In [33]: spl1 = spl.derivative(1)
         #spl2 = spl.derivative(2)
         #spl3 = spl.derivative(3)
         plt.plot(xint, spl(xint), lw=3, label='Zlepek')
         plt.plot(xint, spl1(xint), label='Prvi odvod')
         #plt.plot(xint, spl2(xint), label='Drugi odvod')
         #plt.plot(xint, spl3(xint), label='Tretji odvod')
         plt.plot(x, y, 'ro', label='Znane vrednosti')
         #plt.plot(spl.get_knots(), spl(spl.get_knots()), 'k.', label='Vozli B-zlepka')
         plt.legend(loc=(1.01, 0));
```



Poglavje 8

Aproksimacija

8.1 Uvod

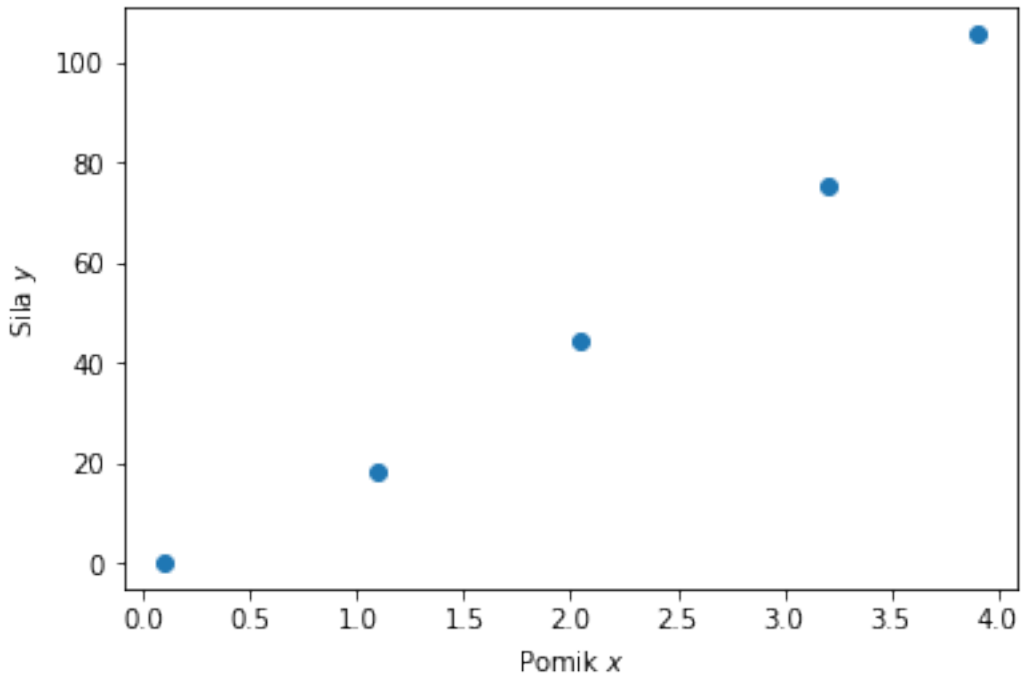
V strojniški praksi se pogosto srečamo s tabelo podatkov, ki so lahko obremenjeni z merilnimi ali numeričnimi napakami.

Oglejmo si primer meritve (linearne) vzmeti (x je raztezek, y je pomerjena sila):

```
In [1]: import numpy as np

        x = np.array([0.1,  1.1, 2.05, 3.2, 3.9  ])
        y = np.array([0.2, 18.1, 44.3, 75.5, 105.6])
```

Poglejmo si podatke na sliki, najprej uvozimo potrebne pakete:

```
In [2]: import matplotlib.pyplot as plt
        %matplotlib inline
        plt.plot(x, y, 'o');
        plt.xlabel('Pomik $x$')
        plt.ylabel('Sila $y$');
```



Za konkreten primer bi bilo, glede na poznavanje fizikalnega ozadja linearne vzmeti, primerno, da bi meritve poskušali popisati z linearno funkcijo:

$$f(x) = a_0\, x + a_1$$

Poznamo tabelo n podatkov x_i, y_i za $i = 0, 1, \ldots, n-1$; teh je več, kot jih potrebujemo za določitev dveh konstant a_0 in a_1, zato imamo torej predoločen sistem linearnih enačb:

$$y_i = a_0\, x_i + a_1 \quad \text{za} \quad i = 0, 1, \ldots, n-1.$$

Iščemo taki vrednosti konstanti a_0 in a_1, da se bo funkcija $f(x)$ v znanih točkah x_i *najbolje* ujemala z y_i.

Najprej torej potrebujemo kriterij za *najboljše* ujemanje.

Za vrednosti iz tabele x_i, y_i bi lahko iskali vrednosti a_0 in a_1, pri katerih bi bila vsota absolutne vrednosti odstopkov S najmanjša:

$$P(a_0, a_1) = \sum_{i=0}^{n-1} |y_i - (a_0\, x_i + a_1)|.$$

Ker pa taka funkcija $P(a_0, a_1)$ ni zvezno odvedljiva,raje uporabimo **metodo najmanjših kvadratov**:

$$S(a_0, a_1) = \sum_{i=0}^{n-1} (y_i - (a_0\, x_i + a_1))^2\,.$$

Takšna funkcija $S(a_0, a_1)$ je zvezna in zvezno odvedljiva. S parcialnim odvajanjem po parametrih a_0 in a_1 lahko najdemo stacionarno točko (parcialna odvoda sta enaka 0). Postopek si bomo za linearno funkcijo pogledali v naslednjem poglavju.

8.2 Metoda najmanjših kvadratov za linearno funkcijo

Poiskati moramo konstanti a_0, a_1, da bo vsota kvadratov razlik med funkcijo in tabelirano vrednostjo (x_i, y_i, kjer $i = 0, 1, \ldots, n-1$ in je n število tabeliranih podatkov):

$$S(a_0, a_1) = \sum_{i=0}^{n-1} (y_i - (a_0\, x_i + a_1))^2$$

najmanjša. Vrednost bo najmanjša v stacionarni točki, ki jo določimo s parcialnim odvodom po parametrih a_0 in a_1.

Najprej izvedemo parcialni odvod po parametru a_0:

$$\frac{\partial S(a_0, a_1)}{\partial a_0} = 2 \sum_{i=0}^{n-1} (y_i - a_0\, x_i - a_1)\ (-x_i)$$

Izraz uredimo:

$$\frac{\partial S(a_0, a_1)}{\partial a_0} = -2 \left(\sum_{i=0}^{n-1} y_i\, x_i - a_0 \sum_{i=0}^{n-1} x_i^2 - a_1 \sum_{i=0}^{n-1} x_i \right)$$

Podobno postopamo še za a_1:

$$\frac{\partial S(a_0, a_1)}{\partial a_1} = 2 \sum_{i=0}^{n-1} (y_i - a_0\, x_i - a_1)\ (-1)$$

$$\frac{\partial S(a_0, a_1)}{\partial a_1} = -2 \left(\sum_{i=0}^{n-1} y_i - a_0 \sum_{i=0}^{n-1} x_i - a_1 \sum_{i=0}^{n-1} 1 \right)$$

Ker v stacionarni točki velja $\partial S(a_0, a_1)/\partial a_0 = 0$ in $\partial S(a_0, a_1)/\partial a_1 = 0$, iz zgornjih izrazov izpeljemo:

$$a_0 \sum_{i}^{n} x_i^2 + a_1 \sum_{i}^{n} x_i = \sum_{i}^{n} y_i\, x_i$$

in

$$a_0 \sum_{i}^{n} x_i + a_1\, n = \sum_{i}^{n} y_i.$$

Dobili smo sistem dveh linearnih enačb za neznanki a_0 in a_1, ki ga znamo rešiti. Imenujemo ga *normalni sistem* (število enačb je enako številu neznank).

Zapišimo normalni sistem v matrični obliki:

```
In [3]: A = [[np.sum(x**2), np.sum(x)], # matrika koeficientov
             [np.sum(x), len(x)]]
        b = [np.dot(y,x), np.sum(y)]    # vektor konstant
        A = np.asarray(A)
        b = np.asarray(b)
        print('A:', A)
        print('b:', b)
```

```
A: [[ 30.8725  10.35  ]
 [ 10.35     5.    ]]
b: [ 764.185  243.7  ]
```

Sedaj moramo rešiti linearni sistem:

$$\mathbf{A}\,\mathbf{a} = \mathbf{b},$$

Opomba, tukaj smo vektor neznank zapisali kot $\mathbf{a} = (a_0, a_1)$.

Sistem rešimo:

```
In [4]: a0, a1 = np.linalg.solve(A, b)
        a0, a1
```

```
Out[4]: (27.490050804403033, -8.1644051651142853)
```

Preverimo še število pogojenosti:

```
In [5]: np.linalg.cond(A)
```

```
Out[5]: 25.200713508702421
```

Sedaj si bomo pogledali še rezultat. Najprej pripravimo sliko, ki bo vsebovala tudi informacijo o vsoti kvadratov odstopanja $f(x_i)$ od tabeliranih vrednosti y_i.

```
In [6]: def slika(naklon=a0, premik=a1):
            d=1
            def linearna_f(x, a0, a1):
                return a0*x+a1
            def S(x, y, f):
                return np.sum(np.power(y-f,2))
            plt.plot(x,y,'.', label='Tabela podatkov')
            linearna_f1 = linearna_f(x, naklon, premik)
            linearna_f1_MNK = linearna_f(x, a0, a1)
            plt.plot(x, linearna_f1, '-', label='Izbrani parametri')
            plt.plot(x, linearna_f1_MNK, '-', label='Metoda najmanjših kvadratov')
            napaka = S(x, y, linearna_f(x, naklon, premik))
            sprememba_napake_v_smeri_a0 = (S(x, y, linearna_f(x, naklon+d, premik))-napaka)/d
            sprememba_napake_v_smeri_a1 = (S(x, y, linearna_f(x, naklon, premik+d))-napaka)/d
            title = f'S: {napaka:g}, \
                    $\Delta S/\Delta a_0$: {sprememba_napake_v_smeri_a0:g}, \
```

```
                $\Delta S/\Delta a_1$: {sprememba_napake_v_smeri_a1:g}'
        plt.title(title)
        plt.legend()
        plt.ylim(-10,110)
        plt.show()
```

```
In [7]: from ipywidgets import interact
        interact(slika, naklon=(0, 50, 2), premik=(-10, 10, 1));
```

```
A Jupyter Widget
```

8.2.1 Uporaba psevdo inverzne matrike

Do podobnega rezultata lahko pridemo z uporabo psevdo inverzne matrike. Iščemo $y(x) = a_0\,x + a_1$ in nastavimo predoločen sistem $\mathbf{A}\,\mathbf{a} = \mathbf{y}$, kjer je matrika koeficientov $\mathbf{A}$ definirana glede na vrednosti x_i ($i = 0, 1, \ldots, n-1$):

```
In [8]: A=np.array([x, np.ones_like(x)]).T
        A
```

```
Out[8]: array([[ 0.1 ,  1.  ],
               [ 1.1 ,  1.  ],
               [ 2.05,  1.  ],
               [ 3.2 ,  1.  ],
               [ 3.9 ,  1.  ]])
```

Vektor konstant smo označili z $\mathbf{b}$, v našem primeru pa je to kar vektor vrednosti $\mathbf{y}$ z elementi y_i ($i = 0, 1, 2, \ldots, n-1$):

```
In [9]: y
```

```
Out[9]: array([   0.2,   18.1,   44.3,   75.5,  105.6])
```

Vektor konstant $\mathbf{a}$ določimo z uporabo psevdo inverzne matrike:

$$\mathbf{a} = \mathbf{A}^{+}\,\mathbf{y}$$

```
In [10]: np.linalg.pinv(A).dot(y)
```

```
Out[10]: array([ 27.4900508 ,  -8.16440517])
```

8.3 Metoda najmanjših kvadratov za poljubni polinom

Linearno aproksimacijo, predstavljeno zgoraj, bomo posplošili za poljubni polinom stopnje m:

$$f(a_0, a_1, \ldots, a_m, x) = \sum_{v=0}^{m} a_s \underbrace{x^{m-s}}_{f_v(x)},$$

kjer $f_s(x) = x^{m-s}$ imenujemo bazna funkcija ($s = 0, 1, 2, \ldots, m$).

Tabela podatkov naj bo definirana z x_i, y_i, kjer je $i = 0, 1, 2, \ldots, n-1$.

Opomba: zaradi kompaktnosti zapisa bomo konstante a zapisali v vektorski obliki $\mathbf{a} = [a_0, a_1, \ldots, a_m]$.

Uporabimo metodo najmanjših kvadratov:

$$S(\mathbf{a}) = \sum_{i=0}^{n-1} (y_i - f(\mathbf{a}, x_i))^2 = \sum_{i=0}^{n-1} \left(y_i - \sum_{s=0}^{m} a_s \, x_i^{m-s} \right)^2 .$$

Potreben pogoj za nastop ekstrema funkcije $m+1$ neodvisnih spremenljivk je, da najdemo stacionarno točko za vsak a_v, iščemo torej $\partial S(\mathbf{a})/\partial a_v = 0$ (namesto s smo uporabili indeks v).

Najprej določimo parcialni odvod za izbrani a_v:

$$\frac{\partial S(\mathbf{a})}{\partial a_v} = \sum_{i=0}^{n-1} -2 \left(y_i - \sum_{s=0}^{m} a_s \, x_i^{m-s} \right) x_i^{m-v}$$

Opomba: $\frac{\partial}{\partial a_v} \left(\sum_{s=0}^{m} a_s \, x_i^{m-s} \right) = x_i^{m-v}$.

Ker je parcialni odvod v stacionarni točki enak 0, zgornji izraz preoblikujemo:

$$\sum_{i=0}^{n-1} \left(\sum_{s=0}^{m} a_s \, x^{m-s} \right) x_i^{m-v} = \sum_{i=0}^{n-1} y_i \, x_i^{m-v}$$

Izraz uredimo:

$$\sum_{i=0}^{n-1} \sum_{s=0}^{m} a_s \, x_i^{2m-s-v} = \sum_{i=0}^{n-1} y_i \, x_i^{m-v}$$

Zamenjamo vrstni red seštevanja ter izpeljemo:

$$\sum_{s=0}^{m} \left(a_s \sum_{i=0}^{n-1} x_i^{2m-s-v} \right) = \sum_{i=0}^{n-1} y_i \, x_i^{m-v} \quad \text{za:} \quad v = 0, 1, \ldots, m$$

Izpeljali smo enačbo v sistema $m+1$ linearnih enačb:

$$\mathbf{A}\,\mathbf{a} = \mathbf{b}$$

Element $A_{v,s}$ matrike koeficientov je:

$$A_{v,s} = \sum_{i=0}^{n-1} x_i^{2m-v-s},$$

Element vektorja konstant je:

$$b_v = \sum_{i=0}^{n-1} y_i \, x_i^{m-v}$$

8.3.1 Numerični zgled

Uporabimo podatke iz prve naloge in poskusimo aproksimirati s polinomom 2. stopnje ($m = 2$).
Tabela podatkov je:

```
In [11]: x

Out[11]: array([ 0.1 ,  1.1 ,  2.05,  3.2 ,  3.9 ])

In [12]: y

Out[12]: array([   0.2,   18.1,   44.3,   75.5,  105.6])
```

Izračunajmo matriko koeficientov:

$$A_{v,s} = \sum_{i=0}^{n-1} x_i^{2m-v-s}$$

```
In [13]: m = 2 #stopnja
         A = np.zeros((m+1,m+1))
         for v in range(m+1):
             for s in range(m+1):
                 A[v,s] = np.sum(x**(2*m-v-s))
         A

Out[13]: array([[ 355.32690625,  102.034125  ,   30.8725    ],
                [ 102.034125  ,   30.8725    ,   10.35      ],
                [  30.8725    ,   10.35      ,    5.        ]])
```

Izračunajmo še vektor konstant:

$$b_v = \sum_{i=0}^{n-1} y_i \, x_i^{m-v}$$

```
In [14]: b = np.zeros(m+1)
         for v in range(m+1):
             b[v] = np.dot(y,x**(m-v))
         b

Out[14]: array([ 2587.36975,   764.185  ,   243.7    ])
```

Preverimo število pogojenosti:

```
In [15]: np.linalg.cond(A)

Out[15]: 963.21258562906587
```

Rešimo sistem:

```
In [16]: a = np.linalg.solve(A, b)
         a

Out[16]: array([  3.18393375,  14.64106847,  -1.22621065])
```

Glede na definicijo aproksimacijskega polinoma:

$$f(a_0, a_1, \ldots, a_m, x) = \sum_{v=0}^{m} a_v \, x^{m-v}$$

Kar v konkretnem primeru je aproksimacijski polinom:

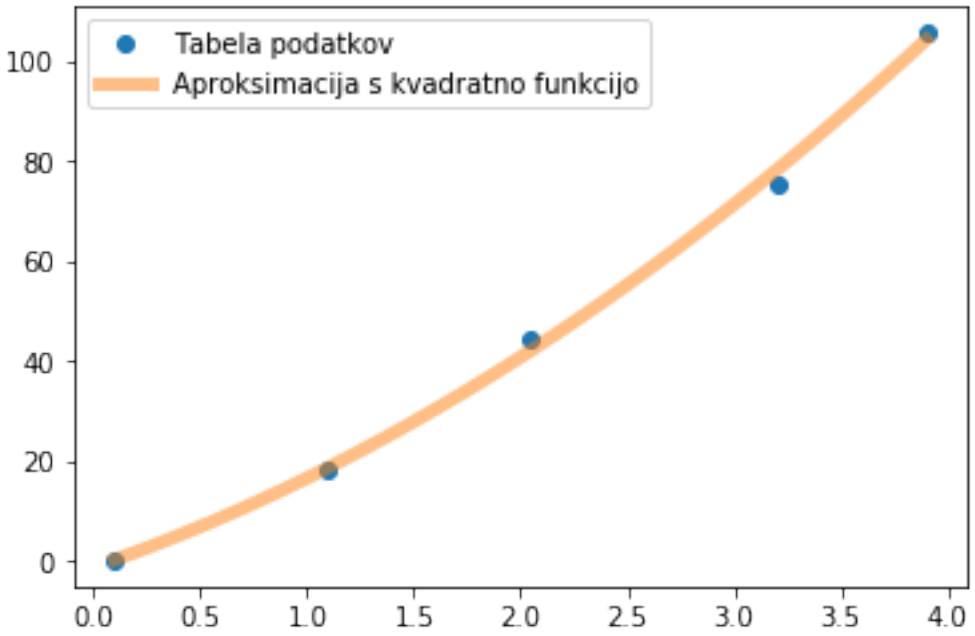
$$f(x) = 3.18393375\, x^2 + 14.64106847\, x - 1.22621065$$

Definirajmo numerično implementacijo:

```
In [17]: def apr_polinom(x, a):
             """Vrne vrednosti aproksimacijskega polinoma

             :param x: vrednosti kjer računamo aproksimirani rezultat
             :param a: koeficienti aproksimacijskega polinoma
             """
             m = len(a) - 1
             return np.sum(np.asarray([_*x**(m-v) for v,_ in enumerate(a)]), axis=0)
```

Prikažemo:

```
In [18]: x_g = np.linspace(np.min(x), np.max(x), 100) # več točk za prikaz
         plt.plot(x, y, 'o', label='Tabela podatkov')
         plt.plot(x_g, apr_polinom(x_g, a), lw=5, alpha=0.5, label='Aproksimacija s kvadratno funkcijo')
         plt.legend();
```



Poglejmo še napako aproksimacije:

$$e_i = y_i - f(x_i)$$

za $i = 0, 1, 2, \ldots, n-1$.

Pri pravilno izvedni aproksimaciji je nekaj e_i pozitivnih in nekaj negativnih. Poglejmo, če je to res v našem primeru:

```
In [19]: e = y - apr_polinom(x, a)
         e

Out[19]: array([-0.06973553, -0.6315245 ,  2.13153872, -2.72869002,  1.29841133])
```

Opomba: višje stopnje polinoma kot uporabimo, večja je verjetnost slabe pogojenosti. Iz tega razloga s stopnjo polinoma ne pretiravamo (v praksi uporabljamo predvsem nizke stopnje)!

8.3.2 Uporaba `numpy` za aproksimacijo s polinomom

Poglejmo si, kako uporabimo knjižnico `numpy` za polinomsko aproksimacijo.

Najprej uporabimo funkcijo `numpy.polyfit` (dokumentacija[1]):

```
polyfit(x, y, deg, rcond=None, full=False, w=None, cov=False)
```

ki zahteva tri parametre: `x` in `y` predstavljata tabelo podatkov (lahko tudi v obliki seznamov vektorjev), `deg` pa stopnjo polinoma. Ostali parametri so opcijski (npr. `w` za uporabo uteži pri aproksimaciji).

Funkcija `polyfit` vrne seznam koeficientov polinoma (najprej za najvišji red); rezultat je lahko tudi seznam seznamov (če so vhodni podatki seznam vektorjev).

Poglejmo si uporabo za predhodno obravnavani primer:

```
In [20]: koef = np.polyfit(x, y, deg=2)
         koef

Out[20]: array([  3.18393375,  14.64106847,  -1.22621065])

In [21]: a # rezultat lastne implementacije

Out[21]: array([  3.18393375,  14.64106847,  -1.22621065])
```

Ko imamo koeficiente, lahko ustvarimo objekt polinoma s klicem `numpy.poly1d` (dokumentacija[2]):

```
poly1d(c_or_r, r=False, variable=None)
```

kjer `c_or_r` predstavlja seznam koeficientov polinoma oz. ničle polinoma v primeru, da je `r=True`. Funkcija vrne instanco objekta, s klicem katere lahko izračunamo vrednosti aproksimacijskega polinoma pri `x`, lahko pa izračunamo tudi druge stvari, kot na primer ničle polinoma.

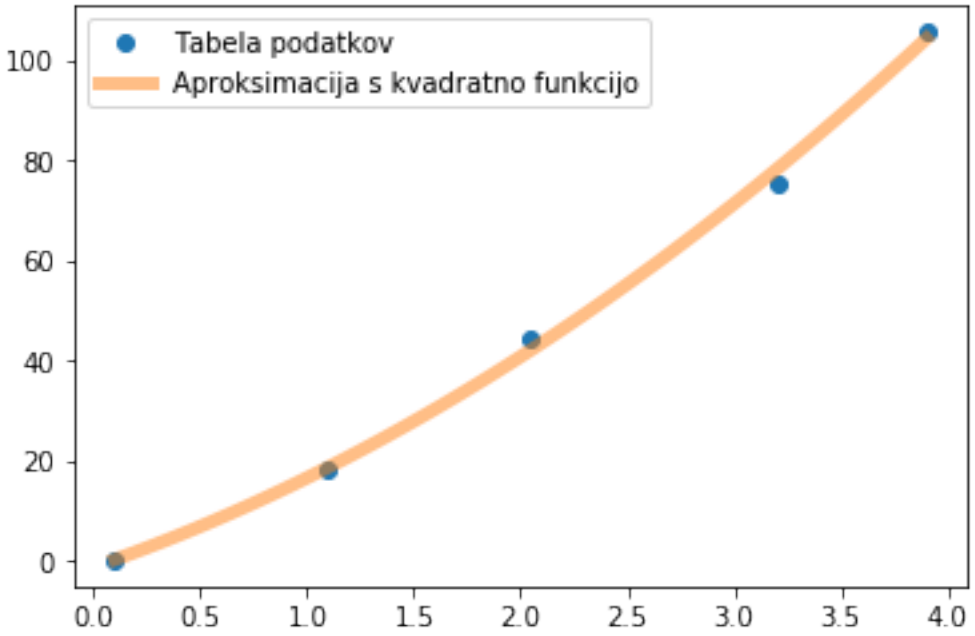
Poglejmo si primer:

[1] https://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html
[2] https://docs.scipy.org/doc/numpy/reference/generated/numpy.poly1d.html#numpy.poly1d

```
In [22]: p = np.poly1d(koef) # p je instanca objekta poly1d. Aproksimacijo sedaj dobimo z y = p(x).
```

Izrišimo vrednosti:

```
In [23]: plt.plot(x, y, 'o', label='Tabela podatkov')
         plt.plot(x_g, p(x_g), lw=5, alpha=0.5, label='Aproksimacija s kvadratno funkcijo')
         plt.legend();
```



Izračunajmo ničle polinoma:

```
In [24]: p.roots
```

```
Out[24]: array([-4.68070044,  0.08227923])
```

8.4 Aproksimacija s poljubno funkcijo

Pri aproksimaciji nismo omejeni zgolj na polinome. Tabele podatkov lahko aproksimiramo:

- z linearno kombinacijo linearno neodvisnih baznih funkcij ali
- s funkcijo, v kateri nastopajo parametri v nelinearni zvezi (npr. $a_0 \sin(a_1 x + a_2)$).

Za podrobnosti glejte vir J. Petrišič: Uvod v Matlab za inženirje, Fakulteta za strojništvo 2013, str 145.

Osredotočili se bomo na uporabo `scipy` paketa za aproksimacijo z nelinearno fukcijo, ki temelji na metodi najmanjših kvadratov.

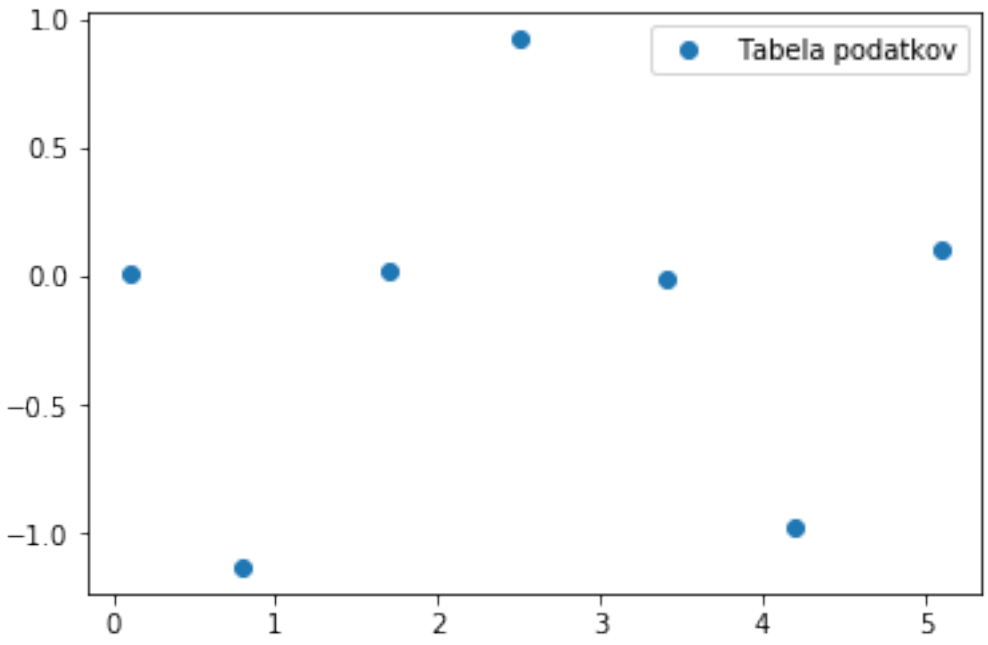
8.4.1 Aproksimacija s harmonsko funkcijo

Tabela podatkov je definira kot:

```
In [25]: x = np.array([ 0.1, 0.8, 1.7, 2.5, 3.4, 4.2,  5.1])
         y = np.array([ 0.01, -1.13, 0.02, 0.92, -0.01, -0.98, 0.1])
```

Prikažimo tabelo podatkov:

```
In [26]: plt.plot(x, y, 'o', label='Tabela podatkov')
         plt.legend();
```



Aproksimacijo z nelinearno funkcijo bomo izvedli s pomočjo `scipy.optimize.curve_fit` (dokumentacija[3]):

```
curve_fit(f, xdata, ydata, p0=None, sigma=None, absolute_sigma=False, check_finite=True, bounds=(-inf,
```

katera zahteva tri parametre: `f` predstavlja definicijo Python funkcije, s katero želimo aproksimirati, in katere parametre spreminjamo z uporabo metode najmanjših kvadratov. `xdata` in `ydata` predstavljata tabelo podatkov. Priporočeno je tudi, da definiramo približek iskanih parametrov `p0`. Ostali parametri so opcijski.

Funkcija vrne dve numerični polji: `popt`, ki predstavlja najdene parametre ter `pcov`, ki predstavljajo ocenjeno kovarianco `popt`.

Definirajmo najprej Python funkcijo, katere prvi parameter je neodvisna spremenljivka `x`, nato pa sledijo parametri, ki jih želimo določiti:

```
In [27]: def func(x, A, ω, ϕ):
             return A*np.sin(ω*x+ϕ)
```

[3] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html

kjer je A amplitua, ω krožna frekvenca in ϕ faza harmonske funkcije. S pomočjo slike lahko ugibamo prve približke: A=1, ω=1, ϕ=0

Sedaj uvozimo curve_fit in izvedemo optimizacijski postopek:

```
In [28]: from scipy.optimize import curve_fit
```

```
In [29]: popt, pcov = curve_fit(func, x, y, p0=[1, 1, 0])
         popt
```

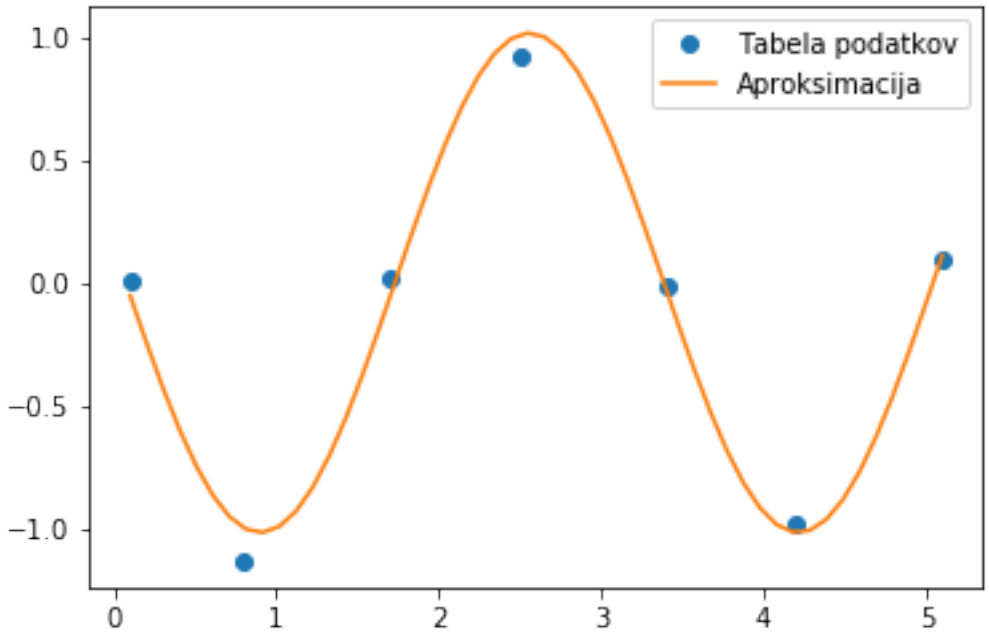
```
Out[29]: array([ 1.01461945,  1.89652046, -3.27943097])
```

Izračunali smo pričakovane vrednosti (glejte zgoraj).

```
In [30]: func(x, *popt)
```

```
Out[30]: array([-0.0525477 , -0.99608783, -0.05612665,  1.00860622, -0.02753942,
                -1.01426499,  0.1110182 ])
```

```
In [31]: x_g = np.linspace(np.min(x), np.max(x), 50)
         y_g = func(x_g, *popt) # bodi pozorni kako smo v funkcijo posredovali parametre
         plt.plot(x, y, 'o', label='Tabela podatkov')
         plt.plot(x_g, y_g, label='Aproksimacija')
         plt.legend();
```



8.5 Nekaj vprašanj za razmislek!

1. Podatki:

   $$x = [-1.00, -0.50, 0.00, 0.50, 1.00]$$

   $$y = [-1.00, -0.55, 0.00, 0.45, 1.00]$$

   uporabite linearne zlepke in določite prvi odvod.

2. Na zgornjih podatkih izračunajte linearno aproksimacijo ter določite parametra aproksimacije.

3. Na nateznem testu ste testirali aluminijeve vzorce; rezultati testa so podani spodaj.

   Napetost [MPa]:

   $$\sigma = [34.5, 69.0, 103.5, 138.0]$$

   Specifična deformacija [mm/m]

   $$vzorec_1 = [0.46, 0.95, 1.48, 1.93]$$

   $$vzorec_2 = [0.34, 1.02, 1.51, 2.09]$$

   $$vzorec_3 = [0.37, 1.00, 1.51, 2.05]$$

   S pomočjo linearne aproksimacije določite elastični modul (napetost/specifična deformacija) vsakega posameznega vzorca.

4. Za vzorce zgoraj linearno aproksimirajte elastični modul čez vse vzorce. Določite tudi standardno napako (glejte `np.std`).

5. Raziščite pomoč za funkcijo `np.polyfit` in utežite različne vzorce z različno utežjo (npr. da prvi meritvi zaupate manj). Izračunajte nato linearno aproksimiran elastični modul.

6. Pojasnite bistvo metode najmanjših kvadratov na primeru linearne aproksimacije.

7. Podatki:

   $$x = [1.0, 2.5, 3.5, 4.0, 1.1, 1.8, 2.2, 3.7]$$

   $$y = [6.008, 15.722, 27.130, 33.772, 5.257, 9.549, 11.098, 28.828$$

   Pripravite in pojasnite funkcijo za linearno aproksimacijo.

8. Nadaljujte zgornjo nalogo in z vgrajeno funkcijo `np.polyfit` izvedite linearno, kvadratno in kubično polinomsko aproksimacijo.

9. Nadaljujte zgornjo nalogo in aproksimacije narišite ter določite standarno napako. Katera aproksimacija najbolje popiše podatke?

10. Definirajte polinom 2. ali 3. stopnje. Dodajte šum (enakomeren `np.random.rand` ali normalen `np.random.randn`) ter nato aproksimirajte s polinomom 1., 2. in 3. stopnje. Vse rezultate narišite in jih vrednotite.

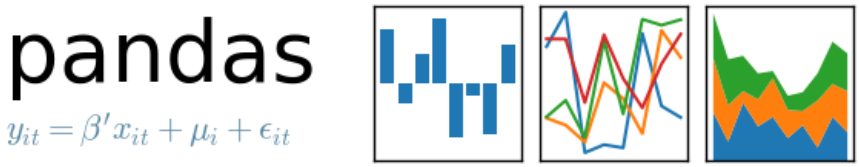
11. Podatke iz prejšnje točke aproksimirajte s pomočjo kubičnih zlepkov. Uporabite vgrajeno funkcijo in preučite vpliv parametra `s`.

8.6 Dodatno

Naredite *.exe* svojega programa:

- https://pypi.python.org/pypi/py2exe/
- http://www.pyinstaller.org/
- http://pinm.ladisk.si/323/kako-iz-python-kodo-prevedem-v-exe-datoteko

Poglejte pandas paket[4].



8.6.1 Aproksimacija z zlepki in uporabo `SciPy`

Tabela podatkov naj bo:

```
In [32]: x = np.linspace(-3, 3, 20)
         x

Out[32]: array([-3.        , -2.68421053, -2.36842105, -2.05263158, -1.73684211,
                -1.42105263, -1.10526316, -0.78947368, -0.47368421, -0.15789474,
                 0.15789474,  0.47368421,  0.78947368,  1.10526316,  1.42105263,
                 1.73684211,  2.05263158,  2.36842105,  2.68421053,  3.        ])

In [33]: np.random.seed(0) # seme generatorja naključnih števil
         y = np.exp(-x**2) + 0.1 * np.random.normal(scale=.5, size=len(x))
         y

Out[33]: array([ 0.08832603,  0.02075073,  0.0526001 ,  0.12684217,  0.14234432,
                 0.08387244,  0.34226064,  0.52862107,  0.79385312,  0.99590738,
                 0.98257964,  0.87172774,  0.57424082,  0.30083997,  0.15492949,
                 0.06565014,  0.08950146, -0.00659471,  0.01639626, -0.04258138])
```

Poglejmo si objekt `scipy.interpolate.UnivariateSpline` (dokumentacija[5]), ki omogoča tako interpolacijo kot aproksimacijo z zlepki:

```
UnivariateSpline(x, y, w=None, bbox=[None, None], k=3, s=None, ext=0, check_finite=False)
```

Parametra `x` in `y` predstavljata tabelo podatkov.

Opcijski parameter `s` določa vrednost, katere vsota kvadratov razlik aproksimacijskega zlepka in aproksimacijskih točk ne sme preseči:

```
sum((w[i] * (y[i]-spl(x[i])))**2, axis=0) <= s
```

[4] http://pandas.pydata.org/
[5] https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.UnivariateSpline.html

`w` so uteži posameznih točk.

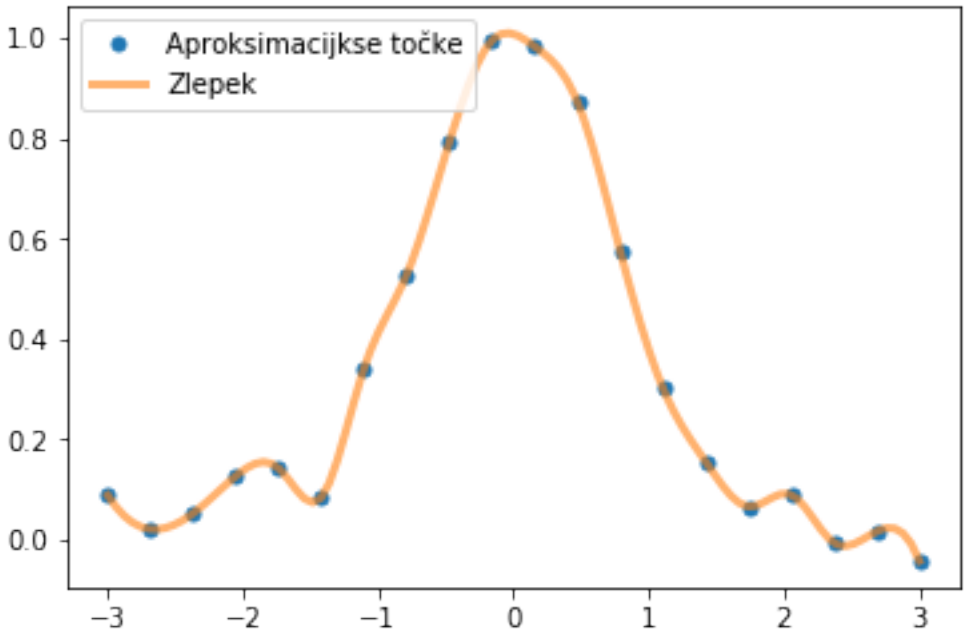
Če definiramo `s=0`, zahtevamo interpolacijo.

Parameter `k` definira stopnjo polinomskega zlepka (privzeto je `k=3`).

Aproksimacijo z zlepki izvedemo tako, da ob tabeli podatkov `x` in `y` definiramo še parameter `s`. Izvedimo interpolacijo:

```
In [34]: from scipy.interpolate import UnivariateSpline
         spl = UnivariateSpline(x, y, s=0.)
```

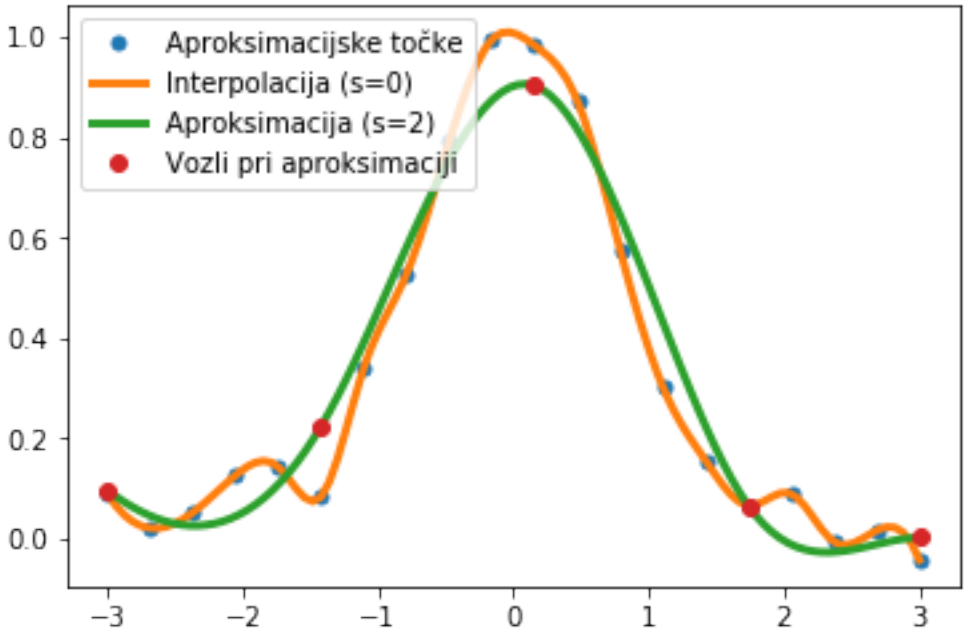
```
In [35]: x_g = np.linspace(-3, 3, 100)
         plt.plot(x, y, 'o', ms=5, label='Aproksimacijkse točke')
         plt.plot(x_g, spl(x_g), lw=3, label='Zlepek', alpha=0.6)
         plt.legend();
```



Izvedimo še aproksimacijo:

```
In [36]: spl_a = UnivariateSpline(x, y, s=.1)
```

```
In [37]: plt.plot(x, y, 'o', ms=5, label='Aproksimacijske točke')
         plt.plot(x_g, spl(x_g), lw=3, label='Interpolacija (s=0)');
         plt.plot(x_g, spl_a(x_g), lw=3, label='Aproksimacija (s=2)');
         plt.plot(spl_a.get_knots(), spl_a(spl_a.get_knots()), 'o', label='Vozli pri aproksimaciji');
         plt.legend();
```



Dejanski preostanek:

```
In [38]: spl_a.get_residual()
```

```
Out[38]: 0.0999987185545532
```

Poglavje 9

Reševanje enačb

9.1 Uvod

V okviru reševanja enačb obravnavamo poljubno enačbo, ki je odvisna od spremenljivke x in iščemo rešitev:

$$f(x) = 0.$$

Rešitvam enačbe rečemo tudi *koreni* (angl. *roots*). Koren enačbe $f(x) = 0$ je hkrati tudi ničla funkcije $y = f(x)$.

Funkcija $y = f(x)$ ima lahko ničle stopnje:

- ničla prve stopnje: funkcija seka abscisno os pod neničelnim kotom,
- ničle sode stopnje: funkcija se dotika abscisne osi, vendar je ne seka,
- ničle lihe stopnje: funkcija seka abscisno os, pri ničli stopnje 3 in več imamo prevoj (tangenta je vzporedna z abscisno osjo).

Tukaj je pomembno izpostaviti, da iščemo rešitev poljubne enačbe $f(x) = 0$. Če za linearne, kvadratne ali kubične enačbe, lahko določimo analitične rešitve; za večino nelinearnih enačb analitične rešitve ne moremo določiti. Iz tega razloga so numerični pristopi toliko bolj pomembni.

9.1.1 Omejitve funkcije $f(x)$

Za funkcijo $y = f(x)$ zahtevamo, da je na zaprtem intervalu $[x_0, x_1]$ zvezna. Pri računanju ničel, se bomo omejili samo na ničle prve stopnje.

9.1.2 Zgled

Poljubno funkcijo $y = f(x)$ lahko definiramo s *Pythonovo funkcijo*; za zgled tukaj definirajmo polinom:

```
In [1]: def f(x):
            x**3 - 10*x**2 + 5
```

Ker pa gre za polinom $x^3 - 10x^2 + 5$ s koeficienti `[1, -10, 0, 5]` pa je bolje, da ga definiramo s pomočjo `np.poly1d`[1]:

[1] https://docs.scipy.org/doc/numpy/reference/generated/numpy.poly1d.html

```
numpy.poly1d(c_or_r, r=False, variable=None)
```

kjer so parametri:

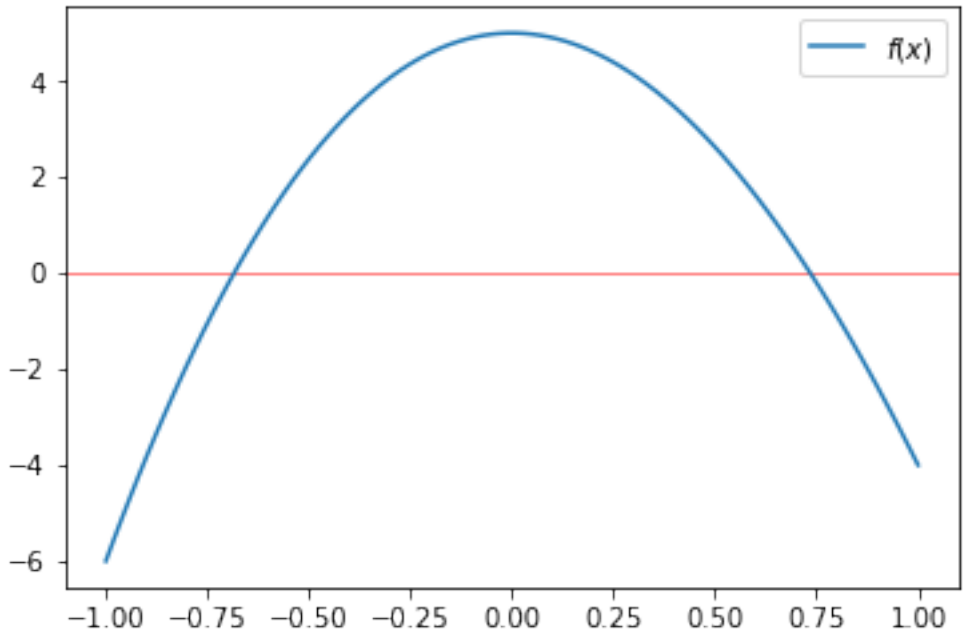
- `c_or_r` koeficienti polinoma s padajočo potenco ali če je `r=True` ničle polinoma,
- `r` je privzeto `False`, kar pomeni, da se podajo koeficienti polinoma,
- `variable` spremenljivka, ki se izpiše pri uporabi funkcije `print()`.

Uvozimo `numpy` in definirajmo polinom:

```
In [2]: import numpy as np # uvozimo numpy
        f = np.poly1d([1, -10, 0, 5]) # definiramo koeficiente polinoma
        print(f) # prikažemo polinom

   3      2
1 x - 10 x + 5
```

Prikažimo funkcijo $f(x)$:

```
In [3]: import matplotlib.pyplot as plt       # uvozimo matplotlib
        %matplotlib inline
        x_r = np.linspace(-1, 1, 100)
        plt.axhline(0, color='r', lw=0.5)     # horizontalna črta,
        plt.plot(x_r, f(x_r), label='$f(x)$') # da je ničla nekje blizu $x = 0.7$.
        plt.legend();
```



Opazimo, da so ničle funkcije $f(x)$ blizu -0,7 in +0,7. Objekt `poly1d` ima atribut `roots` ali tudi `r` (glejte dokumentacijo[2]), ki vrne te ničle:

[2] https://docs.scipy.org/doc/numpy/reference/generated/numpy.poly1d.html

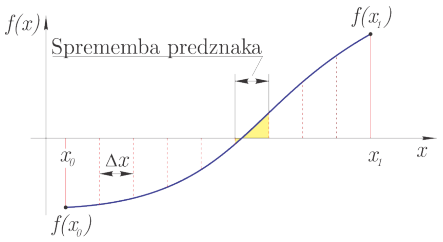
```
In [4]: f.r
```

```
Out[4]: array([ 9.94949106,  0.73460351, -0.68409457])
```

V nadaljevanju bo naš cilj numerično določiti ničlo za poljubno funkcijo `f`.

9.2 Inkrementalna metoda

Inkrementalno reševanje temelji na ideji, da v kolikor ima funkcija $f(x)$ pri x_0 in x_1 različna predznaka, potem je vmes vsaj ena ničla. Zaprti interval $[x_0, x_1]$ razdelimo torej na odseke širine Δx; na odseku, kjer opazimo spremembo predznaka, je vsaj ena ničla funkcije. Metoda je prikazana na sliki.



Za ničlo zahtevamo:

$$|x_{i+1} - x_i| < \varepsilon \quad \text{in} \quad |f(x_{i+1})| + |f(x_i)| < D,$$

kjer je ϵ zahtevana natančnost rešitve in D izbrana majhna vrednost, ki prepreči, da bi kot ničlo razpoznali pol (kar sicer zaradi pogoja zveznosti ni mogoče).

Inkrementalna metoda ima nekatere slabosti:

- je zelo počasna,
- lahko zgreši dve ničli, ki sta zelo blizu,
- večkratne sode ničle (lokalni ekstrem, ki se samo dotika abscise) ne zazna.

Inkrementalna metoda spada med t. i. *zaprte* (angl. *bracketed*) metode, saj išče ničle funkcije samo na intervalu $[x_0, x_1]$. Pozneje bomo spoznali tudi *odprte* metode, ki lahko konvergirajo k ničli zunaj podanega intervala.

Zaradi vseh zgoraj navedenih slabosti inkrementalno metodo pogosto uporabimo samo za izračun začetnega približka ničle.

9.2.1 Numerična implementacija

Poglejmo si sedaj inkrementalno iskanje ničel funkcije:

```
In [5]: def inkrementalna(fun, x0, x1, dx):
            """ Vrne prvi interval (x1, x2) kjer leži ničla

            :param fun: funkcija katere ničle iščemo
            :param x1:  spodnja meja iskanja
            :param x2:  zgornja meja iskanja
```

```
        :param dx:  inkrement iskanja
        """
        x_d = np.arange(x0, x1, dx)    # pripravimo x vrednosti
        f_d = np.sign(fun(x_d))        # pripravimo predznake funkcije
        f_d = f_d[1:]*f_d[:-1]         # pomnožimo sosednje elemente
        i = np.argmin(f_d)             # prvi prehod skozi ničlo
        # vsota abs funk vrednosti
        x0 = x_d[i]
        x1 = x_d[i+1]
        D = np.abs(fun(x0)) + np.abs(fun(x1))
        return np.asarray([x0, x1]), D
```

Poglejmo sedaj uporabo na zgoraj definiranem polinomu:

```
In [6]: rez_inkr, D = inkrementalna(f, 0., 1., 0.001)
        rez_inkr
```

```
Out[6]: array([ 0.734,  0.735])
```

Ničja je izolirana z natančnostjo 0,001, preverimo še vsoto absolutnih funkcijskih vrednosti:

```
In [7]: D
```

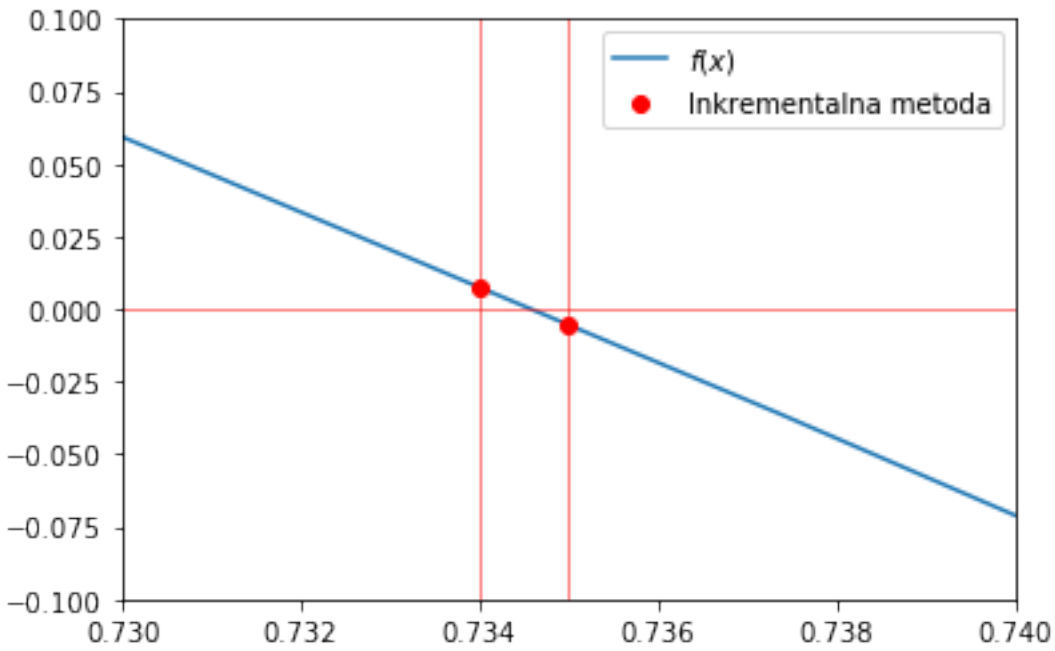
```
Out[7]: 0.013071529000000304
```

Ugotovimo, da je relativno majhna; bomo pa se s sledečimi metodami trudili rezultat bistveno izboljšati.
Pripravimo sliko:

```
In [8]: def fig():
            plt.plot(x_r, f(x_r), label='$f(x)$')
            plt.axhline(0, color='r', lw=0.5)     # horizontalna črta
            plt.axvline(rez_inkr[0], color='r', lw=0.5)     # vertikalna črta
            plt.axvline(rez_inkr[1], color='r', lw=0.5)     # vetrikalna črta
            plt.plot(rez_inkr, f(rez_inkr), 'ro', label='Inkrementalna metoda')
            plt.xlim(0.73,0.74)
            plt.ylim(-0.1, 0.1)
            plt.legend();
```

Prikažimo rezultat:

```
In [9]: fig()
```

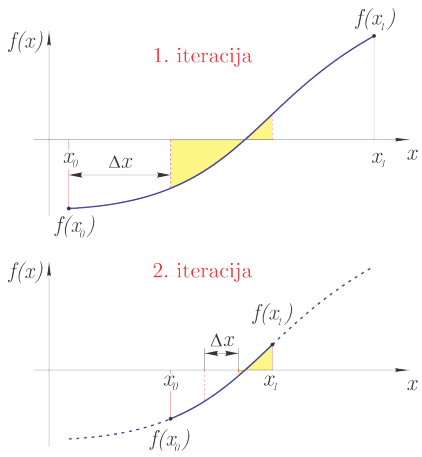


Da smo torej na intervalu $[0,1]$ izračunali rešitev z natančnostjo $\Delta x = 0,001$, smo morali 1000-krat klicati funkcijo $f(x)$. Gre za zelo neučinkovito metodo, zato bomo iskali boljše načine; najprej s preprostim iterativnim inkrementalnim pristopom.

9.3 Iterativna inkrementalna metoda

Iterativna inkrementalna metoda v prvi iteraciji z inkrementalno metodo omeji interval iskanja ničel pri relativno velikem koraku. Interval, najden v prvi iteraciji, se v drugi iteraciji razdeli na manjše intervale in ponovi se inkrementalno iskanje ničle. Tretja iteracije se nato omeji na interval določen v drugi in tako dalje. Z iteracijami zaključimo, ko smo dosegli predpisano natančnost rešitve ϵ.

Metoda je prikazana na sliki:



9.3.1 Numerična implementacija

```
In [10]: def inkrementalna_super(fun, x0, x1, iteracij=3):
             """ Vrne interval (x0, x1) kjer leži ničla

             :param fun: funkcija katere ničlo iščemo
             :param x0:  spodnja meja iskanja
             :param x1:  zgornja meja iskanja
             :iteraci:   število iteracij inkrementalne metode
             """
             for i in range(iteracij):
                 dx = (x1 - x0)/10
                 x0x1, _ = inkrementalna(fun, x0, x1, dx)
                 x0, x1 = x0x1
             # vsota abs funk vrednosti
             D = np.abs(fun(x0)) + np.abs(fun(x1))
             return np.asarray([x0, x1]), D
```

S 30 klici funkcije $f(x)$ tako dobimo podobno natančnost kot prej v 1000:

```
In [11]: rez30, D30 = inkrementalna_super(f, 0., 1., iteracij=3)
         rez30

Out[11]: array([ 0.734,  0.735])

In [12]: rez_inkr

Out[12]: array([ 0.734,  0.735])
```

Seveda pa lahko natančnost bistveno izboljšamo z večanjem števila iteracij:

```
In [13]: rez80, D80 = inkrementalna_super(f, 0., 1., iteracij=8)
         rez80

Out[13]: array([ 0.7346035 ,  0.73460351])
```

Preverimo še kriterij vsote absolutnih funkcijskih vrednosti, ki mora biti majhen:

```
In [14]: D80

Out[14]: 1.3073143190212022e-07
```

9.4 Bisekcijska metoda

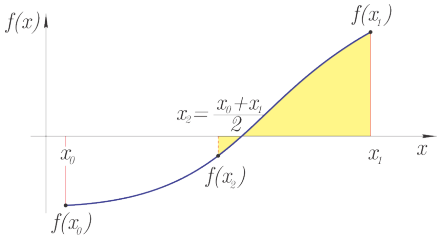
Na intervalu $[x_0, x_1]$, kjer vemo, da obstaja ničla funkcije (predznaka $f(x_0)$ in $f(x_1)$ se razlikujeta), lahko uporabimo *bisekcijsko metodo*.

Ideja metode je:

- interval $[x_0, x_1]$ razdelimo na pol (od tukaj ime: *bi-sekcija*): $x_2 = (x_0 + x_1)/2$,
- če imata $f(x_0)$ in $f(x_2)$ različne predznake, je nov interval iskanja ničle $[x_0, x_2]$, sicer pa: $[x_2, x_1]$,

- glede na predhodni korak definiramo nov zaprt interval $[x_0, x_1]$ in nadaljujemo z iterativnim postopkom, dokler ne dosežemo želene natančnosti $|x_1 - x_0| < \varepsilon$.

Slika metode:



Bisekcijska metoda spada med *zaprte* metode, ki vrne ničlo funkcije na podanem intervalu $[x_0, x_1]$.

9.4.1 Ocena napake

Če v začetku začnemo z intervalom $\Delta x = |x_1 - x_0|$, potem je natančnost bisekcijske metode po prvem koraku bisekcije:

$$\varepsilon_1 = \Delta x/2,$$

po drugem koraku:

$$\varepsilon_2 = \Delta x/2^2$$

in po n korakih:

$$\varepsilon_n = \Delta x/2^n.$$

Ponavadi zahtevamo, da je rešitev podana z natančnostjo ε in iz zgornje enačbe lahko izpeljemo število potrebnih korakov bisekcijske metode:

$$n = \frac{\log\left(\frac{\Delta x}{\varepsilon}\right)}{\log(2)}.$$

Seveda je število korakov celo število.

9.4.2 Numerična implementacija

```
In [15]: def bisekcija(fun, x0, x1, tol=1e-3, Dtol=1e-1, izpis=True):
             """ Vrne ničlo z natančnostjo tol

             :param fun: funkcija katere ničlo iščemo
             :param x0:  spodnja meja iskanja
             :param x1:  zgornja meja iskanja
             :param tol: zahtevana natančnost
             :param Dtol:največja vsota absolutnih vrednosti rešitve
             :izpis:     ali na koncu izpiše kratko poročilo
             """
             if np.sign(fun(x0))==np.sign(fun(x1)):
```

```
        raise Exception('Ničla ni izolirana. Root is not bracketed.')
    n = np.ceil( np.log(np.abs(x1-x0)/tol)/np.log(2) ).astype(int) # število iteracij
    for i in range(n):
        x2 = (x0 + x1) / 2
        f1 = fun(x0)
        f3 = fun(x2)
        f2 = fun(x1)
        if np.sign(fun(x2))!=np.sign(fun(x0)):
            x1 = x2
        else:
            x0 = x2
    D = np.abs(fun(x0)) + np.abs(fun(x1))
    if D > Dtol:
        raise Exception('Verjetnost pola ali več ničel.')
    r = (x0+x1)/2
    if izpis:
        decimalk = int(np.log10(1/tol)) # ne deluje vedno in za vse primere:)
        print(f'Rešitev: {r:5.{decimalk}f}, število iteracij: {n:g}, D: {D:5.5f}')
    return r
```

Sedaj poskusimo najti ničlo z natančnostjo `1e-3`:

```
In [16]: bisekcija(f, 0, 1, tol=1e-3);
```

```
Rešitev: 0.735, število iteracij: 10, D: 0.01277
```

V desetih iteracijah smo dobili isto natančen rezultat kakor zgoraj pri iterativni inkrementalni metodi `rez30`. Poglejmo še izračun ničle s še večjo natančnostjo:

```
In [17]: bisekcija(f, 0, 1, tol=1e-6);
```

```
Rešitev: 0.734603, število iteracij: 20, D: 0.00001
```

Hitrost izvajanja lahko preverimo s t. i. *magic funkcijo* `timeit` (dokumentacija[3]), ki večkrat požene funkcijo in analizira čas izvajanja. Če je pred magic funkcijo dvojni znak %%, se izvede in meri čas celotne celice, če pa le enojni %, pa samo ene vrstice.

```
In [18]: %%timeit
         bisekcija(f, 0., 1., izpis=False)
```

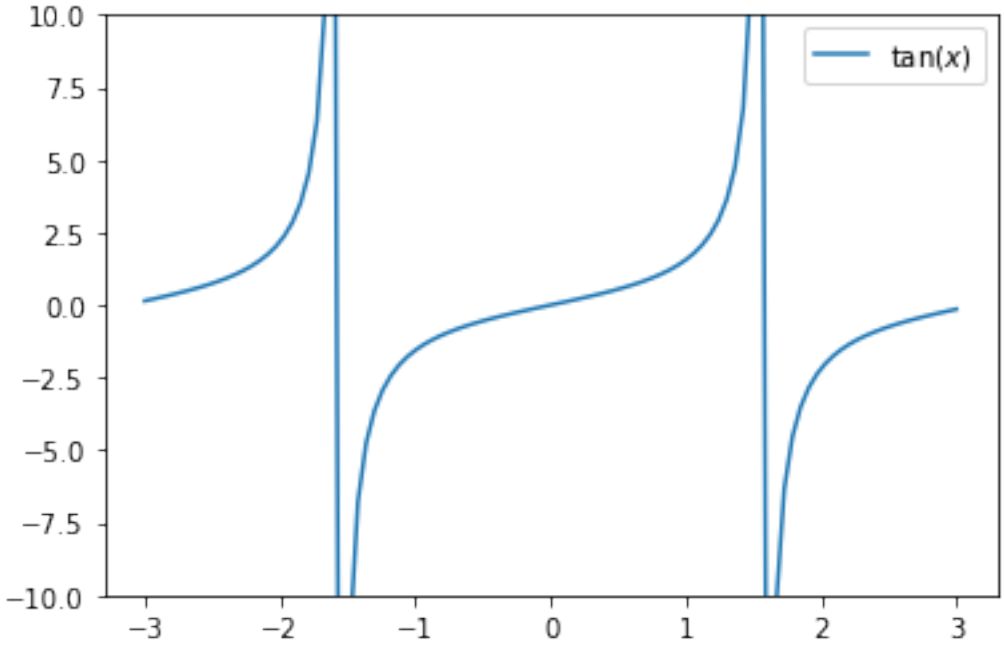
```
1.12 ms ± 222 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Iskanje ničle v okolici pola

Poglejmo sedaj iskanje ničle funkcije `tan` v okolici pola (ničla dejansko ne obstaja):

[3] http://ipython.readthedocs.io/en/stable/interactive/magics.html#magic-timeit

```
In [19]: x_t = np.linspace(-3, 3, 100)
         plt.plot(x_t, np.tan(x_t), label='$\\tan(x)$')
         plt.ylim(-10, 10)
         plt.legend();
```



V primeru iskanja na intervalu $[-1, 1]$ najdemo pravo ničlo:

```
In [20]: bisekcija(np.tan, -1, 1, tol=1e-3);
```

```
Rešitev: -0.000, število iteracij: 11, D: 0.00098
```

V primeru iskanja v okolici pola, pa nas program na to opozori (klic funkcije je tukaj zakomentiran, sicer se avtomatsko generiranje ne izvede pravilno):

```
In [21]: # bisekcija(np.tan, -3, 0, tol=1e-6)
```

V `scipy` vgrajena bisekcijska metoda takega preverjanja nima (zaradi hitrosti) in bo vrnila rezultat, ki bo pa napačen. Pri uporabi moramo torej biti previdni.

9.4.3 Uporaba `scipy.optimize.bisect`

Bisekcijska metoda je *počasna*, vendar zanesljiva metoda iskanja ničel in je implementirana znotraj `scipy`. Najprej jo uvozimo:

```
In [22]: from scipy.optimize import bisect
```

```
bisect(f, a, b, args=(), xtol=2e-12, rtol=8.8817841970012523e-16, maxiter=100,
    full_output=False, disp=True)
```

Funkcija `bisect` zahteva tri parametre: funkijo `f` ter parametra `a` in `b`, ki definirata zaprti interval `[a, b]`. Predznaka `f(a)` in `f(b)` morata biti različna. Ostali parametri, npr. absolutna `xtol` in relativna `rtol` napaka ter največje število iteracij `maxiter` so opcijski - imajo privzete vrednosti. Za več glejte dokumentacijo[4].

Poglejmo uporabo:

```
In [23]: bisect(f, a=0, b=1, xtol=1e-3)
```

```
Out[23]: 0.7353515625
```

in hitrost:

```
In [24]: %timeit bisect(f, 0, 1, xtol=1e-3)
```

```
232 µs ± 15.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Ugotovimo, da je *naša* implementacija bisekcijske metode približno 5-krat počasnejša.

Preverimo še lahko funkcijo `tan(x)`, najprej na intervalu, kjer je funkcija zvezna:

```
In [25]: bisect(np.tan, -1, 1)
```

```
Out[25]: 0.0
```

Potem še v okolici pola, kjer ni zvezna:

```
In [26]: bisect(np.tan, -3, -1)
```

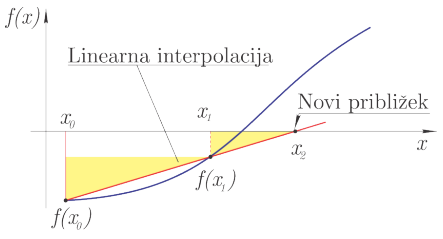
```
Out[26]: -1.5707963267941523
```

kar je napačna rešitev!

9.5 Sekantna metoda

Sekantna metoda zahteva dva začetna približka x_0 in x_1 in funkcijo $f(x)$. Ob predpostavki linearne interpolacije med točkama $x_0, f(x_0)$ in $x_1, f(x_1)$ (skozi točki potegnemo *sekanto*, od tukaj tudi ime), se določi x_2, kjer ima linearna interpolacijska funkcija ničlo. x_2 predstavlja nov približek ničle.

Glede na sliko:



[4] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.bisect.html

lahko zapišemo (podobna trikotnika sta na sliki označena z rumeno):

$$\frac{f(x_1)}{x_2 - x_1} = \frac{f(x_0) - f(x_1)}{x_1 - x_0}.$$

Sledi, da je nov približek ničle:

$$x_2 = x_1 - f(x_1)\,\frac{x_1 - x_0}{f(x_1) - f(x_0)}.$$

V naslednjem koraku pri sekantni metodi izvedemo sledeče zamenjave: $x_0 = x_1$ in $x_1 = x_2$.

Sekantna metoda spada med *odprte* metode, saj lahko najde ničlo funkcije, ki se nahaja zunaj območja $[x_0, x_1]$.

9.5.1 Ocena napake

Konzervativno lahko napako ocenimo iz razlike med dvema zaporednima približkoma:

$$\varepsilon = |x_{n-1} - x_n|$$

9.5.2 Konvergenca in red konvergence

Konvergenca pomeni, da zaporedje približkov konvergira k rešitvi enačbe α (α je rešitev enačbe).

Red konvergence označuje hitrost konvergiranja.

Z ε označimo napako približka in z vsakim korakom iteracije napako linearno zmanjšamo, bi to zapisali:

$$\varepsilon_n = C\,\varepsilon_{n-1}^1,$$

kjer bi bila C konstanta, napaka ε bi pa imela potenco 1; posledično je red konvergence 1!

Pri predhodno obravnavani bisekcijski metodi napako na vsakem koraku zmanjšamo za $1/2$ ($\varepsilon_n/\varepsilon_{n-1} = C = 1/2$). *Bisekcijska metoda ima red konvergence 1.*

Red konvergence *sekantne metode je višji* in jo je mogoče oceniti z:

$$\varepsilon_n = C\,\varepsilon_{n-1}^{1.618}.$$

Iz zgornje ocene sledi, da se na vsakem koraku iteracije število točnih cifer poveča za približno 60%. Ker je red konvegence višji od 1 in manjši od kvadratične, tako konvergenco imenujemo *superlinearna* konvergenca.

Še boljšo, kvadratično, konvergenco ima Newtonova metoda. Pri kvadratični konvergenci se na vsakem koraku iteracije število točnih cifer podvoji. Isti red konvergence ima tudi *Ridderjeva* metoda, ki je implementirana v paketu `scipy` (glejte dokumentacijo[5]) in je zaprtega tipa ter sorodna sekantni metodi. Kljub uporabnosti si je tukaj ne bomo podrobno ogledali.

[5] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.ridder.html

9.5.3 Numerična implementacija

```
In [27]: def sekantna(fun, x0, x1, tol=1e-3, Dtol=1e-1, max_iter=50, izpis=True):
             """ Vrne ničlo z natančnostjo tol

             :param fun: funkcija katere ničlo iščemo
             :param x0:  spodnja meja iskanja
             :param x1:  zgornja meja iskanja
             :param tol: zahtevana natančnost
             :max_iter:  maksimalno število iteracij preden se izvajanje prekine
             :param Dtol:največja vsota absolutnih vrednosti rešitve
             :izpis:     ali na koncu izpiše kratko poročilo
             """
             if np.sign(fun(x0))==np.sign(fun(x1)):
                 raise Exception('Ničla ni izolirana. Root is not bracketed.')
             for i in range(max_iter):
                 f0 = fun(x0)
                 f1 = fun(x1)
                 x2 = x1 - f1 * (x1 - x0)/(f1 - f0)
                 x0 = x1
                 x1 = x2
                 if izpis:
                     print('{:g}. korak: x0={:g}, x1={:g}.'.format(i+1, x0, x1))
                 if np.abs(x1-x0)<tol:
                     r = (x0+x1)/2
                     D = np.abs(fun(x0)) + np.abs(fun(x1))
                     if D > Dtol:
                         raise Exception('Verjetnost pola ali več ničel.')
                     r = (x0+x1)/2
                     if izpis:
                         decimalk = int(np.log10(1/tol)) # ne deluje vedno in za vse primere:)
                         print(f'Rešitev: {r:5.{decimalk}f}, D: {D:5.5f}')
                     return r
```

Poglejmo si uporabo:

```
In [28]: sekantna(f, 0, 1., tol=1.e-8, izpis=True);
```

```
1. korak: x0=1, x1=0.555556.
2. korak: x0=0.555556, x1=0.707845.
3. korak: x0=0.707845, x1=0.737957.
4. korak: x0=0.737957, x1=0.734549.
5. korak: x0=0.734549, x1=0.734603.
6. korak: x0=0.734603, x1=0.734604.
7. korak: x0=0.734604, x1=0.734604.
Rešitev: 0.73460351, D: 0.00000
```

in hitrost

```
In [29]: %timeit sekantna(f, 0, 1., tol=1.e-8, izpis=False)
```

```
361 μs ± 38.9 μs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Kakor smo zapisali zgoraj, je sekantna metoda odprtega tipa. Rešitev enačbe je lahko zunaj podanega intervala. Poglejmo si primer:

```
In [30]: sekantna(np.tan, 1, 2);

1. korak: x0=2, x1=1.41615.
2. korak: x0=1.41615, x1=1.85165.
3. korak: x0=1.85165, x1=1.69887.
4. korak: x0=1.69887, x1=1.97485.
5. korak: x0=1.97485, x1=2.09379.
6. korak: x0=2.09379, x1=2.43519.
7. korak: x0=2.43519, x1=2.76579.
8. korak: x0=2.76579, x1=3.05013.
9. korak: x0=3.05013, x1=3.13625.
10. korak: x0=3.13625, x1=3.14158.
11. korak: x0=3.14158, x1=3.14159.
Rešitev: 3.142, D: 0.00002
```

9.5.4 Uporaba `scipy.optimize.newton`

Znotraj `scipy` je sekantna metoda definirana v okviru `scipy.optimize.newton` funkcije. Če le-tej namreč ne podamo funkcije, ki definira prvi odvod, potem je uporabljena sekantna metoda (glejte dokumentacijo[6]).

Najprej jo uvozimo:

```
In [31]: from scipy.optimize import newton

newton(func, x0, fprime=None, args=(), tol=1.48e-08, maxiter=50, fprime2=None)
```

Funkcija `newton` zahteva dva parametra: funkcijo `func` ter začetno vrednost `x0`. Opcijska parametra sta še `fprime` in `fprime2` za prvi in drugi odvod funkcije `func`; če nista podana, se uporabi sekantna metoda (več o odvodih spodaj pri Newtonovi metodi). Največje število iteracij definira `maxiter`.

V primeru sekantne metode, se druga meja intervala izračuna glede na kodo:

```
if x0 >= 0:
    x1 = x0*(1 + 1e-4) + 1e-4
else:
    x1 = x0*(1 + 1e-4) - 1e-4
```

Poglejmo si uporabo:

```
In [32]: r = newton(f, x0=0.1, tol=1.e-8)
         f'Rezultat: {r:2.8f}'

Out[32]: 'Rezultat: 0.73460351'
```

in hitrost

```
In [33]: %timeit newton(f, x0=0.1, tol=1.e-8)
```

[6] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.newton.html

```
170 μs ± 16.2 μs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Ugotovimo, da je naša implementacija sekantne metode podobno hitra.

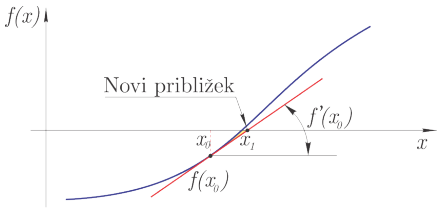
Uporabimo še Ridderjevo metodo:

```
In [34]: from scipy.optimize import ridder
         r = ridder(f, a=0, b=1, xtol=1.e-8)
         f'Rezultat: {r:2.8f}'

Out[34]: 'Rezultat: 0.73460350'
```

9.6 Newtonova metoda

V literaturi za **Newtonovo** metodo tudi najdemo izraza **tangentna** in **Newton-Raphsonova** metoda. Potrebuje en začetni približel x_0, poleg definicije funkcije $f(x)$ pa tudi njen odvod $f'(x)$.

Princip delovanja metode je prikazan na sliki:



Metodo bi lahko izpeljali grafično (s slike), tukaj pa si poglejmo izpeljavo s pomočjo Taylorjeve vrste:

$$f(x_{i+1}) = f(x_i) + f'(x_i)\ (x_{i+1} - x_i) + O^2\,(x_{i+1} - x_i)\,,$$

če naj bo pri x_{i+1} vrednost funkcije nič, potem velja:

$$0 = f(x_i) + f'(x_i)\ (x_{i+1} - x_i) + O^2\,(x_{i+1} - x_i)\,.$$

Naredimo napako metode in zanemarimo člene višjega reda v Taylorjevi vrsti. Lahko izpeljemo:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

x_{i+1} je tako nov približek iskane ničle.

Algoritem Newtonove metode je:

1. izračunamo nov približek x_{i+1},
2. računanje prekinemo, če je največje število iteracij doseženo (rešitve enačbe nismo našli),
3. če velja $|x_{i+1} - x_i| < \varepsilon$ računanje prekinemo (izračunali smo približek ničle), sicer povečamo indeks i in gremo v prvi korak.

Opombi:

- ε je zahtevana absolutna natančnost,

- *Newtonova* metoda lahko divergira, zato v algoritmu predpišemo največje število iteracij.

Zgoraj smo omenili, da je Newtonova metoda ena izmed boljših metod za iskanje ničel funkcij. Ima pa tudi nekaj slabosti/omejitev:

- spada med *odprte* metode,
- kvadratična konvergenca je zagotovljena le v dovolj majhni okolici rešitve enačbe,
- poznati moramo odvod funkcije.

9.6.1 Red konvegence

Red konvergence Newtonove metode je kvadraten:

$$\varepsilon_n = C\,\varepsilon_{n-1}^2,$$

kjer je C:

$$C = -\frac{f''(x)}{2\,f'(x)}.$$

Konvergenca je torej hitra, v vsaki novi iteraciji se število točnih števk v približku podvoji.

9.6.2 Numerična implementacija

```
In [35]: def newtonova(fun, dfun, x0, tol=1e-3, Dtol=1e-1, max_iter=50, izpis=True):
             # ime `newtonova` zato ker je `newton` vgrajena funkcija v `scipy`
             """ Vrne ničlo z natančnostjo tol

             :param fun:  funkcija katere ničlo iščemo
             :param dfun: f'
             :param x0:   začetni približek
             :param tol:  zahtevana natančnost
             :max_iter:  maksimalno število iteracij preden se izvajanje prekine
             :param Dtol:največja vsota absolutnih vrednosti rešitve
             :izpis:     ali na koncu izpiše kratko poročilo
             """
             for i in range(max_iter):
                 x1 = x0 - fun(x0)/dfun(x0)
                 if np.abs(x1-x0)<tol:
                     r = (x0+x1)/2
                     D = np.abs(fun(x0)) + np.abs(fun(x1))
                     if D > Dtol:
                         raise Exception('Verjetnost pola ali več ničel.')
                     if izpis:
                         decimalk = int(np.log10(1/tol)) # ne deluje vedno in za vse primere:)
                         print(f'Rešitev: {x1:5.{decimalk}f}, število iteracij: {i+1}, D: {D:5.8f}')
                     return x1
                 x0 = x1
             raise Exception('Metoda po {:g} iteracijah ne konvergira'.format(max_iter))
```

Definirajmo polinom `f` in njegov prvi odvod `df`:

```
In [36]: def f(x):
             return x**3 - 10*x**2 + 0*x + 5
         def df(x):
             return 3*x**2 - 20*x
```

Izračunajmo sedaj ničlo:

```
In [37]: newtonova(fun=f, dfun=df, x0=1, tol=1e-8);
```

```
Rešitev: 0.73460351, število iteracij: 5, D: 0.00000000
```

Preverimo hitrost izvajanja:

```
In [38]: %timeit newtonova(fun=f, dfun=df, x0=1, tol=1e-8, izpis=False)
```

```
23.3 µs ± 2.52 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

9.6.3 Uporaba `scipy.optimize.newton`

Znotraj `scipy` je Newtonova metoda definirana v okviru `scipy.optimize.newton` funkcije, ki smo jo že spoznali zgoraj pri sekantni metodi (glejte dokumentacijo[7]).

```
newton(func, x0, fprime=None, args=(), tol=1.48e-08, maxiter=50, fprime2=None)
```

Če v funkcijo `newton` poleg funkcije `func` in začetne vrednosti `x0` podamo tudi prvi odvod `fprime`, se dejansko uporabi Newtonova metoda; če podamo tudi drugi odvod `fprime2`, se uporabi Halleyeva metoda, ki temelji na izrazu:

$$x_{n+1} = x_n - \frac{2f(x_n)f'(x_n)}{2f'^2(x_n) - f(x_n)f''(x_n)}$$

Poglejmo si primer uporabe Newtonove metode v `scipy.optimize`:

```
In [39]: r = newton(func=f, x0=1., fprime=df)
         f'Rezultat: {r:2.8f}'
```

```
Out[39]: 'Rezultat: 0.73460351'
```

In izmerimo hitrost:

```
In [40]: %timeit newton(func=f, x0=1., fprime=df)
```

```
9.83 µs ± 1.41 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

[7] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.newton.html

9.7 Reševanje sistemov nelinarnih enačb

Rešujemo sistem enačb, ki ga v vektorski obliki zapišemo takole:

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}.$$

V skalarni obliki zgornji vektorski izraz zapišemo:

$$\begin{aligned} f_0(x_0, x_1, \ldots, x_{n-1}) &= 0 \\ f_1(x_0, x_1, \ldots, x_{n-1}) &= 0 \\ &\vdots \\ f_{n-1}(x_0, x_1, \ldots, x_{n-1}) &= 0. \end{aligned}$$

Reševanje sistema n nelinearnih enačb je bistveno bolj zahtevno kot reševanje ene same nelinearne enačbe. Tak sistem enačb ima lahko več rešitev in katero izračunamo, je odvisno od začetnih pogojev. Ponavadi nam pri dobri izbiri začetnih pogojev pomaga fizikalni problem, ki ga rešujemo.

Za računanje rešitve sistema enačb se *Newtonova* metoda izkaže kot najenostavneša in pogosto tudi najboljša (obstajajo tudi druge metode, ki pa so velikokrat variacije Newtonove metode).

Podobno kot pri izpeljavi Newtonove metode za reševanje ene enačbe, tudi tukaj začnemo z razvojem funkcije f_i v Taylorjevo vrsto:

$$f_i(\mathbf{x} + \Delta\mathbf{x}) = f_i(\mathbf{x}) + \sum_{j=1}^{n} \frac{\partial f_i}{\partial x_j} \Delta x_j + O^2(\mathbf{\Delta x}).$$

Naredimo napako metode, ko zanemarimo člene drugega in višjih redov ter zapišemo izraz v matrični obliki:

$$\mathbf{f}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{f}(\mathbf{x}) + \mathbf{J}(\mathbf{x})\, \Delta\mathbf{x},$$

kjer je $\mathbf{J}(\mathbf{x})$ *Jakobijeva* matrika pri vrednostih $\mathbf{x}$. Elementi Jakobijeve matrike so:

$$J_{ij} = \frac{\partial f_i}{\partial x_j}.$$

Če naj bo $\mathbf{x} + \Delta\mathbf{x}$ rešitev sistema enačb, mora veljati:

$$\mathbf{0} = \mathbf{f}(\mathbf{x}) + \mathbf{J}(\mathbf{x})\, \Delta\mathbf{x}$$

in torej sledi:

$$\mathbf{J}(\mathbf{x})\, \Delta\mathbf{x} = -\mathbf{f}(\mathbf{x}).$$

Izpeljali smo sistem linearnih enačb, matrika koeficientov je označena z $\mathbf{J}(\mathbf{x})$, vektor neznank je $\Delta\mathbf{x}$ in vektor konstant $-\mathbf{f}(\mathbf{x})$.

Opomba: analitično računanje Jakobijeve matrike je lahko zamudno in zato jo pogosto približno izračunamo pri $\mathbf{x}$ numerično:

$$\frac{\partial f_i}{\partial x_j} \approx \frac{f_i(\mathbf{x} + \mathbf{e}_j\, h) - f_i(\mathbf{x})}{h},$$

kjer je h majhen premik in je $\mathbf{e}_j$ enotski pomik v smeri x_j. Če se Jakobijeva matrika izračuna numerično, govorimo o sekantni metodi in ne Newtonovi.

Pri numeričnem izračunu si lahko pomagamo s funkcijo `scipy.optimize.approx_fprime` (za podrobnosti glejte dokumentacijo[8]).

[8] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.approx_fprime.html

9.7.1 Numerična implementacija

Algoritem torej je:

1. Izberemo začetni približek $\mathbf{x}_0$, največje število iteracij in postavimo indeks na nič: $i = 0$.
2. Izračunamo Jakobijevo matriko $\mathbf{J}(\mathbf{x_i})$ in rešimo linearni sistem: $\mathbf{J}(\mathbf{x}_i)\,\Delta\mathbf{x}_i = -\mathbf{f}(\mathbf{x}_i)$.
3. Izračunamo nov približek: $\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta\mathbf{x}_i$.
4. Če je napaka manjša od zahtevane, se postopek prekine*. Postopek prekinemo tudi, če je število iteracij večje od dovoljenega, sicer povečamo indeks $i = i + 1$ in se vrnemo v korak 2.

* Opomba:

Napako lahko ocenimo z normo razlike dveh zaporednih približkov:

$$\sum_{j}^{n-1} \left| x_{i,j} - x_{i-1,j} \right| < \varepsilon,$$

kjer je i indeks iteracije in j indeks elementa.

9.7.2 Uporaba `scipy.optimize.root`

Funkcija `scipy.optimize.root` ima obsežno dokumentacijo[9] in omogoča večje število različnih pristopov:

```
root(fun, x0, args=(), method='hybr', jac=None, tol=None, callback=None, options=None)
```

Če uporabimo privzete parametre, moramo definirati zgolj vektorsko funkcijo `fun` in začetno vrednost `x0`.

Uvozimo funkcijo:

```
In [41]: from scipy.optimize import root
```

Poglejmo si uporabo na zgledu (gre za zgled na str. 76, Jože Petrišič, Reševanje enačb, 1996, FS, UNI-LJ):

$$f_0(\mathbf{x}) = x_0^2 + x_0\,x_1 - 10 = 0,$$
$$f_1(\mathbf{x}) = x_1 + 3\,x_0\,x_1^2 - 57 = 0,$$

kjer je vektor $\mathbf{x} = [x_0, x_1]$.

Najprej definirajmo Python funkcijo, ki vrne seznam rezultatov funkcij $[f_0(\mathbf{x}), f_1(\mathbf{x})]$:

```
In [42]: def func(x):
             return [x[0]**2 + x[0]*x[1] -10,
                     x[1] +3*x[0]*x[1]**2 - 57]
```

Definirajmo še Jakobijevo matriko:

```
In [43]: def J(x):
             return np.array([[2*x[0]+x[1], x[0]],
                              [3*x[1]**2,   1+6*x[0]*x[1]]])
```

[9] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.root.html

Uporabimo začetne vrednosti $x_0 = 1,5$ in $x_1 = 3,5$ ter rešimo problem:

```
In [44]: rešitev = root(fun=func, x0=[1.5, 3.5], jac=J)
         rešitev

Out[44]:     fjac: array([[-0.22347867, -0.97470882],
                 [ 0.97470882, -0.22347867]])
              fun: array([ -6.97220059e-12,   3.14770432e-11])
          message: 'The solution converged.'
             nfev: 10
             njev: 1
              qtf: array([ -2.83570821e-08,  -1.32374877e-08])
                r: array([-29.54114937, -37.8158159 ,  -6.93953536])
           status: 1
          success: True
                x: array([ 2.,  3.])
```

Funkcija `root` vrne obsežen rezultat. Najbolj pomembna sta atribut `x`, ki predstavlja iskano rešitev, in atribut `success`, ki pove, ali je rešitev konvergirala:

```
In [45]: rešitev.x

Out[45]: array([ 2.,  3.])

In [46]: rešitev.success

Out[46]: True
```

9.8 Nekaj vprašanj za razmislek!

1. V simbolni obliki definirajte izraz:

   $$f(x) = x^4 - 6.4\,x^3 + 6.45\,x^2 + 20.538\,x - 31.752.$$

2. Narišite funkcijo $f(x)$. Koliko ničel pričakujemo za funkcijo $f(x)$?
3. V simbolni obliki določite ničlo polinoma $f(x)$ (uporabite sympy).
4. Kako preverimo, ali je ničla ekstrem?
5. Numerično najdite vse ničle z bisekcijsko metodo (uporabite `scipy`).
6. Numerično najdite vse ničle s sekantno metodo (uporabite `scipy`).
7. Numerično najdite vse ničle z Newtonovo metodo (uporabite `scipy`).
8. Podatke:

   $$x = [0, 1, 2, 3, 4, 5]$$

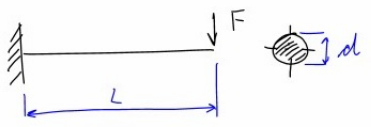
   $$y = [0.95, 0.93, 0.87, 0.77, 0.64, 0.49]$$

   aproksimirajte s funkcijo $\cos(a\,x)$ (z uporabo `scipy.optimize.curve_fit` določite konstanto a).
9. Na intervalu $x = [0, 50]$ z bisekcijsko metodo poiščite vse ničle najdene funkcije $\cos(a\,x)$.

10. Na intervalu $x = [0, 50]$ z Newtonovo metodo poiščite vse ničle najdene funkcije $\cos(a\,x)$.
11. S pomočjo `sympy` najdite simbolno rešitev za:

$$x^2 + y - 2 = 0 \qquad \text{in} \qquad y^2 - 4 = 0.$$

12. Zgoraj definirani problem rešite še numerično s pomočjo `scipy.optimize.root`.
13. Na predhodnem vprašanju preizkusite različne metode (glejte pomoč).
14. Z uporabo bisekcijske metode dimenzionirajte prikazani upogibno obremenjeni nosilec dolžine L (obremenjen s točkovno silo F). Določite velikost polnega krožnega prereza d.



$F = 1\text{kN}$
$L = 1\text{m}\ \sigma_{dop} = 120\text{MPa}$
Reševanje:
$\sigma_u \leq \sigma_{dop} \rightarrow \frac{FL}{W} = \frac{FL32}{\pi d^3} = \sigma_{dop}$

9.9 Dodatno

Tisti, ki ste navdušeni nad Raspberry Pi[10] in uporabljate njihovo kamero (npr. tole brez infrardečega filtra[11]), vas bo morebiti zanimala knjižnica picamera[12].

9.9.1 Uporaba `sympy.solve` za reševanje enačb

Za manjše sistem lahko rešitev najdemo tudi simbolno. Poglejmo si zgornji primer:

```
In [47]: import sympy as sym
         sym.init_printing()
         x, y = sym.symbols('x, y')
         funkcije = [x**2 + x*y -10, y + 3*x*y**2 -57]
         funkcije
```

Out[47]:

$$\left[x^2 + xy - 10, \quad 3xy^2 + y - 57\right]$$

```
In [48]: sol = sym.solve(funkcije, x, y)
         print(f'Število rešitev: {len(sol)}')
         print(f'Prva rešitev: {sol[0]}')
```

```
Število rešitev: 4
Prva rešitev: (2, 3)
```

[10] https://www.raspberrypi.org/
[11] https://www.raspberrypi.org/products/pi-noir-camera/
[12] http://picamera.readthedocs.org/

Poglavje 10

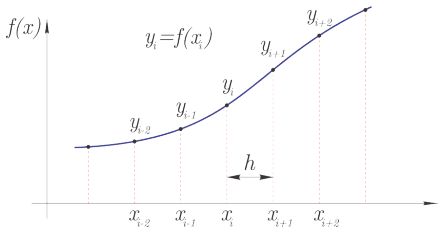
Numerično odvajanje

10.1 Uvod

Vsako elementarno funkcijo lahko analitično odvajamo. Definicija odvoda je:

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}.$$

V okviru tega poglavja se bomo seznanili s tem, kako numerično izračunamo odvod funkcije $f(x)$; pri tem so vrednosti funkcije $f(x)$ podane tabelarično, kakor je prikazano na sliki:



Najprej se bomo osredotočili na ekvidistantno, s korakom h, razporejene vrednosti x_i; vrednosti funkcije pa bodo $y_i = f(x_i)$.

Pri numeričnem odvajanju imamo dva, v principu različna, pristopa:

1. najprej izvedemo **interpolacijo/aproksimacijo**, nato pa na podlagi znanih interpolacijskih/aproksimacijskih funkcij izračunamo odvod (o tej temi smo že govorili pri interpolaciji oz. aproksimaciji) in
2. računanje odvoda **neposredno iz vrednosti iz tabele**.

Slednji pristop temelji na razliki dveh približno enakih funkcijskih vrednosti obremenjeni z zaokrožitveno napako, ki jih delimo z majhno vrednostjo: odvod ima posledično bistveno manj signifikantnih števk kakor pa funkcijske vrednosti v tabeli. Numeričnemu odvajanju se izognemo, če imamo to možnost; je pa v nekaterih primerih (npr. reševanje diferencialnih enačb) nepogrešljivo orodje!

10.2 Aproksimacija prvega odvoda po metodi končnih razlik

Odvod $f'(x)$ lahko aproksimiramo na podlagi razvoja Taylorje vrste. To metodo imenujemo **metoda končnih razlik** ali tudi **diferenčna metoda**.

Razvijmo **Taylorjevo vrsto naprej** (naprej, zaradi člena $+h$):

$$f(x+h) = \sum_{n=0}^{\infty} \frac{h^n}{n!} \frac{d^n}{dx^n} f(x) = f(x) + h\, f'(x) + \underbrace{\frac{h^2}{2} f''(x) + \cdots}_{\mathcal{O}(h^2)}$$

Člen $\mathcal{O}\left(h^2\right)$ označuje napako drugega reda. Če iz enačbe izpostavimo prvi odvod:

$$f'(x) = \frac{1}{h}\left(f(x+h) - f(x)\right) - \underbrace{\frac{h}{2} f''(x) + \cdots}_{\mathcal{O}\left(h^1\right)}$$

Ugotovimo, da lahko ocenimo prvi odvod v točki x_i (to je: $f'_o(x_i)$) na podlagi dveh zaporednih funkcijskih vrednosti:

$$f'_o(x_i) = \frac{1}{h}\left(y_{i+1} - y_i\right)$$

in pri tem naredimo **napako metode**, ki je prvega reda $\mathcal{O}\left(h^1\right)$.

Uporabili smo $y_i = f(x_i)$ (glejte sliko zgoraj).

Napaka je:

$$e = -\frac{h}{2} f''(\xi),$$

kjer je ξ neznana vrednost na intervalu $[x_i, x_{i+1}]$ in smo zanemarili višje člene.

Velja torej izraz:

$$f'(x_i) = f'_o(x_i) + e$$

Sedaj si poglejmo, kako pridemo do istega rezultata s strojno izpeljavo; najprej uvozimo `sympy`:

```
In [1]: import sympy as sym
        sym.init_printing()
```

Definirajmo simbole:

```
In [2]: f, x, h = sym.symbols('f, x, h')
```

Nato nadaljujemo z razvojem **Taylorjeve vrste naprej** (angl. *forward Taylor series*):

```
In [3]: f(x+h).series(h, n=2)
```

```
Out[3]:
```

$$f(x) + h \left.\frac{d}{d\xi_1} f(\xi_1)\right|_{\xi_1 = x} + \mathcal{O}\left(h^2\right)$$

Člen $\mathcal{O}\left(h^2\right)$ vsebuje člene drugega in višjega reda. V zgornji enačbi je uporabljena začasna spremenljivko za odvajanje ξ_1; izvedmo odvajanje in vstavimo $\xi_1 = x$:

In [4]:
```
f(x+h).series(h, n=2).doit()
```

Out[4]:

$$f(x) + h\frac{d}{dx}f(x) + \mathcal{O}\left(h^2\right)$$

Zapišemo enačbo:

In [5]:
```
enačba = sym.Eq(f(x+h), f(x+h).series(h, n=2).doit())
enačba
```

Out[5]:

$$f(h+x) = f(x) + h\frac{d}{dx}f(x) + \mathcal{O}\left(h^2\right)$$

Rešimo jo za prvi odvod $f'(x)$:

In [6]:
```
f1_naprej_točno = sym.solve(enačba, f(x).diff(x))[0]
f1_naprej_točno
```

Out[6]:

$$\frac{1}{h}\left(f(h+x) - f(x) + \mathcal{O}\left(h^2\right)\right)$$

V kolikor odvod drugega in višjih redov ne upoštevamo, smo naredili torej napako:

In [7]:
```
f1_naprej_0 = f1_naprej_točno.expand().getO()
f1_naprej_0
```

Out[7]:

$$\mathcal{O}\left(h\right)$$

Napaka $\mathcal{O}\left(h^1\right)$ je torej prvega reda in če ta člen zanemarimo, naredimo *napako metode* in dobimo oceno odvoda:

In [8]:
```
f1_naprej_ocena = f1_naprej_točno.expand().removeO()
f1_naprej_ocena
```

Out[8]:

$$-\frac{1}{h}f(x) + \frac{1}{h}f(h+x)$$

Ugotovimo, da gre za isti izraz, kakor smo ga izpeljali zgoraj, torej je:

$$y_i' = \frac{1}{h}\left(-y_i + y_{i+1}\right).$$

10.3 Centralna diferenčna shema

10.3.1 Odvod $f'(x)$

Najprej si poglejmo razvoj **Taylorjeve vrste nazaj** (angl. *backward Taylor series*):

```
In [9]: f(x-h).series(h, n=3).doit()
```

Out[9]:

$$f(x) - h\frac{d}{dx}f(x) + \frac{h^2}{2}\frac{d^2}{dx^2}f(x) + \mathcal{O}\left(h^3\right)$$

Ugotovimo, da pri se pri razliki vrste naprej in nazaj odštevajo členi sodega reda; definirajmo:

```
In [10]: def razlika(n=3):
             return f(x+h).series(h, n=n).doit()-f(x-h).series(h, n=n).doit()
         razlika(n=3)
```

Out[10]:

$$2h\frac{d}{dx}f(x) + \mathcal{O}\left(h^3\right)$$

Izvedemo sledeče korake:

1. Taylorjevo vrsto nazaj odštejemo od vrste naprej, sodi odvodi se odštejejo,
2. rešimo enačbo za prvi odvod,
3. določimo napako metode,
4. določimo oceno odvoda.

Izvedimo zgornje korake:

```
In [11]: f1_cent_točno = sym.solve(
                     sym.Eq(f(x+h) - f(x-h), razlika(n=3)), # 1 korak
                     f(x).diff(x))[0]                       # 2.korak
         f1_cent_0 = f1_cent_točno.expand().get0()          # 3.korak
         f1_cent_ocena = f1_cent_točno.expand().remove0()   # 4.korak
```

Ocena 1. odvoda torej je:

```
In [12]: f1_cent_ocena
```

Out[12]:

$$-\frac{1}{2h}f(-h+x) + \frac{1}{2h}f(h+x)$$

Ali:

$$y_i' = \frac{1}{2h}\left(-y_{i-1} + y_{i+1}\right)$$

Napaka metode pa je torej drugega reda:

In [13]:
```
f1_cent_0
```

Out[13]:

$$\mathcal{O}\left(h^2\right)$$

10.3.2 Zgled: $\exp(-x)$

Poglejmo si zgled eksponentne funkcije $f(x) = \exp(-x)$ in za točko $x = 1,0$ izračunajmo prvi odvod $f'(x) = -\exp(-x)$ pri koraku $h_0 = 1$ in $h_1 = 0,1$.

Najprej pripravimo tabelo numeričnih vrednosti in točen rezultat:

In [14]:
```
import numpy as np
x0 = np.array([0., 1., 2.]) # korak h=1
y0 = np.exp(-x0)
h0 = x0[1]-x0[0]

x1 = np.array([0.9, 1.0, 1.1]) # korak h=0.1
y1 = np.exp(-x1)
h1 = x1[1]-x1[0]

f1_točno0 = - np.exp(-1) # točen rezultat
f1_točno1 = - np.exp(-1) # točen rezultat
```

Potem uporabimo shemo naprej:

In [15]:
```
f1_naprej_ocena # da se spomnimo
```

Out[15]:

$$-\frac{1}{h}f(x) + \frac{1}{h}f(h+x)$$

In [16]:
```
f1_naprej0 = (y0[1:]-y0[:-1])/h0    # korak h_0
f1_naprej1 = (y1[1:]-y1[:-1])/h1    # korak h_1
```

Izračunajmo napako pri $x = 1,0$:

In [17]:
```
f1_točno0 - f1_naprej0[1]
```

Out[17]:

$$-0.135335283237$$

In [18]:
```
f1_točno1 - f1_naprej1[1] # korak h1
```

Out[18]:

$$-0.0177958664378$$

Potrdimo lahko, da je napaka pri koraku $h/10$ res približno $1/10$ tiste pri koraku h.

Pogljemo sedaj še napako za centralno diferenčno shemo, ki je drugega reda:

```
In [19]: f1_cent_ocena
```

Out[19]:

$$-\frac{1}{2h}f(-h+x)+\frac{1}{2h}f(h+x)$$

```
In [20]: f1_cent0 = (y0[2:]-y0[:-2])/(2*h0) # korak h_0
         f1_cent1 = (y1[2:]-y1[:-2])/(2*h1) # korak h_1
```

Analizirajmo napako:

```
In [21]: f1_točno0 - f1_cent0[0] # korak h0
```

Out[21]:

0.0644529172103

```
In [22]: f1_točno1 - f1_cent1[0] # korak h1
```

Out[22]:

0.000613439041156

Potrdimo lahko, da je napaka pri koraku $h/10$ res približno $1/100$ tiste pri koraku h.

10.3.3 Odvod $f''(x)$

Če Taylorjevo vrsto naprej in nazaj seštejemo, se odštejejo lihi odvodi:

```
In [23]: def vsota(n=3):
             return f(x+h).series(h, n=n).doit() + f(x-h).series(h, n=n).doit()
         vsota(n=4)
```

Out[23]:

$$2f(x)+h^2\frac{d^2}{dx^2}f(x)+\mathcal{O}\left(h^4\right)$$

Določimo drugi odvod:

```
In [24]: f2_cent_točno = sym.solve(
                     sym.Eq(f(x+h) + f(x-h), vsota(n=4)),    # 1 korak
                     f(x).diff(x,2))[0]                      # 2.korak
         f2_cent_0 = f2_cent_točno.expand().get0()           # 3.korak
         f2_cent_ocena = f2_cent_točno.expand().remove0()    # 4.korak
```

Ocena drugega odvoda je:

```
In [25]: f2_cent_ocena
```

Out[25]:

$$-\frac{2}{h^2}f(x)+\frac{1}{h^2}f(-h+x)+\frac{1}{h^2}f(h+x)$$

Ali:

$$y_i''=\frac{1}{h^2}\left(y_{i-1}-2\,y_i+y_{i+1}\right)$$

Napaka metode pa je ponovno drugega reda:

```
In [26]: f2_cent_0
```

Out[26]:

$$\mathcal{O}\left(h^2\right)$$

10.3.4 Odvod $f'''(x)$

Če želimo določiti tretji odvod, moramo Taylorjevo vrsto razviti do stopnje 5:

```
In [27]: eq_h = sym.Eq(f(x+h)-f(x-h), razlika(n=5))
         eq_h
```

Out[27]:

$$-f(-h+x)+f(h+x)=2h\frac{d}{dx}f(x)+\frac{h^3}{3}\frac{d^3}{dx^3}f(x)+\mathcal{O}\left(h^5\right)$$

Uporaba 1. odvoda, ki smo ga izpeljali zgoraj, nam ne bi koristila, saj je red napake $\mathcal{O}\left(h^2\right)$, kar pomeni, da bi v zgornji pri deljenju s h^3 dobili $\mathcal{O}\left(h^{-1}\right)$.

Uporabimo trik: ponovimo razvoj, vendar na podlagi dodatnih točk, ki sta od x oddaljena za $2h$ in $-2h$:

```
In [28]: eq_2h = eq_h.subs(h, 2*h)
         eq_2h
```

Out[28]:

$$-f(-2h+x)+f(2h+x)=4h\frac{d}{dx}f(x)+\frac{8h^3}{3}\frac{d^3}{dx^3}f(x)+\mathcal{O}\left(h^5\right)$$

Sedaj imamo dve enačbi in dve neznanki; sistem bomo rešili po korakih:

1. enačbo `eq_h` rešimo za prvi odvod,
2. enačbo `eq_2h` rešimo za prvi odvod,
3. enačimo rezultata prvih dveh korakov in rešimo za tretji odvod,
4. določimo napako metode,
5. določimo oceno odvoda.

Izvedimo navedene korake:

```
In [29]: f3_cent_točno = sym.solve(
                 sym.Eq(sym.solve(eq_h, f(x).diff(x))[0],  # 1. korak
                 sym.solve(eq_2h, f(x).diff(x))[0]),       # 2. korak
                 f(x).diff(x,3))[0]                        # 3. korak
         f3_cent_0 = f3_cent_točno.expand().get0()         # 4.korak
         f3_cent_ocena = f3_cent_točno.expand().remove0()  # 5.korak
```

Ocena 3. odvoda je:

```
In [30]: f3_cent_ocena
```

Out[30]:

$$-\frac{1}{2h^3}f(-2h+x)+\frac{1}{h^3}f(-h+x)-\frac{1}{h^3}f(h+x)+\frac{1}{2h^3}f(2h+x)$$

Ali:

$$y_i''' = \frac{1}{h^3}\left(-y_{i-2}/2+y_{i-1}-y_{i+1}+y_{i+2}/2\right)$$

Potrdimo, da je napaka metode drugega reda:

```
In [31]: f3_cent_0
```

Out[31]:

$$\mathcal{O}\left(h^2\right)$$

10.3.5 Odvod $f^{(4)}(x)$

Ponovimo podoben postopek kot za 3. odvod, vendar za 4. odvod seštevamo Taylorjevo vrsto (do stopnje 6) naprej in nazaj:

```
In [32]: eq_h = sym.Eq(f(x+h)+f(x-h), vsota(n=6))
         eq_h
```

Out[32]:

$$f(-h+x)+f(h+x)=2f(x)+h^2\frac{d^2}{dx^2}f(x)+\frac{h^4}{12}\frac{d^4}{dx^4}f(x)+\mathcal{O}\left(h^6\right)$$

Pripravimo dodatno enačbo na podlagi točk, ki sta od x oddaljeni za $2h$ in $-2h$:

```
In [33]: eq_2h = eq_h.subs(h, 2*h)
         eq_2h
```

Out[33]:

$$f(-2h+x)+f(2h+x)=2f(x)+4h^2\frac{d^2}{dx^2}f(x)+\frac{4h^4}{3}\frac{d^4}{dx^4}f(x)+\mathcal{O}\left(h^6\right)$$

Iz dveh enačb določimo 4. odvod:

```
In [34]: f4_cent_točno = sym.solve(
                    sym.Eq(sym.solve(eq_h, f(x).diff(x,2))[0],  # 1. korak
                    sym.solve(eq_2h, f(x).diff(x,2))[0]),       # 2. korak
                    f(x).diff(x,4))[0]                          # 3. korak
         f4_cent_0 = f4_cent_točno.expand().get0()              # 4.korak
         f4_cent_ocena = f4_cent_točno.expand().remove0()       # 5.korak
```

Ocena 4. odvoda je:

```
In [35]: f4_cent_ocena
```

Out[35]:

$$\frac{6}{h^4}f(x)+\frac{1}{h^4}f(-2h+x)-\frac{4}{h^4}f(-h+x)-\frac{4}{h^4}f(h+x)+\frac{1}{h^4}f(2h+x)$$

Ali:

$$y_i^{(4)}=\frac{1}{h^4}\left(y_{i-2}-4\,y_{i-1}+6\,y_i-4\,y_{i+1}+y_{i+2}\right)$$

Potrdimo, da je napaka metode drugega reda:

```
In [36]: f4_cent_0
```

Out[36]:

$$\mathcal{O}\left(h^2\right)$$

10.3.6 Povzetek centralne diferenčne sheme

Zgoraj smo izpeljali prve štiri odvode z napako metode 2. reda. Bistvo zgornjih izpeljav je, da nam dajo uteži, s katerimi moramo množiti funkcijske vrednosti, da izračunamo približek določenega odvoda. Iz tega razloga bomo tukaj te uteži zbrali.

Najprej zberimo vse ocene odvodov v seznam:

```
In [37]: odvodi = [f1_cent_ocena, f2_cent_ocena, f3_cent_ocena, f4_cent_ocena]
         odvodi
```

Out[37]:

$$\left[-\frac{1}{2h}f(-h+x)+\frac{1}{2h}f(h+x),\quad -\frac{2}{h^2}f(x)+\frac{1}{h^2}f(-h+x)+\frac{1}{h^2}f(h+x),\quad -\frac{1}{2h^3}f(-2h+x)+\frac{1}{h^3}f(-h+x)-\frac{1}{h^3}f(h+x)\right.$$

Na razpolago imamo 5 funkcijskih vrednosti (pri legah $x-2h, x-h, x, x+h, x+2h$), ki jih damo v seznam:

```
In [38]: funkcijske_vrednosti = [f(x-2*h), f(x-h), f(x), f(x+h), f(x+2*h)]
```

Utež prvega odvoda za funkcijsko vrednosti $f(x-h)$ izračunamo:

```
In [39]: f1_cent_ocena.expand().coeff(funkcijske_vrednosti[1])
```

Out[39]:

$$-\frac{1}{2h}$$

Sedaj posplošimo in izračunajmo uteži za vse funkcijske vrednosti in za vse ocene odvodov:

```
In [40]: centralna_diff_shema = [[odvod.expand().coeff(fv) for fv in funkcijske_vrednosti] \
                                 for odvod in odvodi]
         centralna_diff_shema
```

Out[40]:

$$\left[\left[0, \; -\frac{1}{2h}, \; 0, \; \frac{1}{2h}, \; 0\right], \; \left[0, \; \frac{1}{h^2}, \; -\frac{2}{h^2}, \; \frac{1}{h^2}, \; 0\right], \; \left[-\frac{1}{2h^3}, \; \frac{1}{h^3}, \; 0, \; -\frac{1}{h^3}, \; \frac{1}{2h^3}\right], \; \left[\frac{1}{h^4}, \; -\frac{4}{h^4}, \; \frac{6}{h^4}, \; -\right.$$

Zgornje povzetke lahko tudi zapišemo v tabelarični obliki:

	y_{i-2}	y_{i-1}	y_i	y_{i+1}	y_{i+2}
$y_i' = \frac{1}{h}\cdot$	0	-0.5	0	0.5	0
$y_i'' = \frac{1}{h^2}\cdot$	0	1	-2	1	0
$y_i''' = \frac{1}{h^3}\cdot$	-0.5	1	0	-1	0.5
$y_i^{(4)} = \frac{1}{h^4}\cdot$	1	-4	6	-4	1

Prikazana centralna diferenčna shema ima napako 2. reda $\mathcal{O}(h^2)$.

10.3.7 Uporaba `scipy.misc.central_diff_weight`

Uteži za centralno diferenčno metodo lahko izračunamo tudi z uporabo `scipy.misc.central_diff_weight()` (dokumentacija[1]):

```
central_diff_weights(Np, ndiv=1)
```

`Np` predstavlja število točk, čez katere želimo izračunati odvod, `ndiv` pa stopnjo odvoda (privzeto 1). Za več informacij glejte dokumentacijo[2].

Uvozimo funkcijo:

```
In [41]: from scipy.misc import central_diff_weights
```

3. odvod (čez pet točk), napaka reda $\mathcal{O}(h^2)$:

[1] https://docs.scipy.org/doc/scipy/reference/generated/scipy.misc.central_diff_weights.html
[2] https://docs.scipy.org/doc/scipy/reference/generated/scipy.misc.central_diff_weights.html

```
In [42]: central_diff_weights(Np=5, ndiv=3)
```

```
Out[42]: array([ -5.00000000e-01,   1.00000000e+00,   4.16333634e-17,
                 -1.00000000e+00,   5.00000000e-01])
```

Opomba: `SymPy.finite_diff_weights` omogoča izračun uteži tudi pri nekonstantnem koraku; v podrobnosti ne bomo šli in radovednega bralca napotimo na dokumentacijo[3].

10.3.8 Izboljšan približek - Richardsova ekstrapolacija

Če je točen odvod je izračunan kot:

$$f'(x_i) = f'_o(x_i) + e,$$

kjer je $f'_o(x_i)$ numerično izračunan odvod v točki x_i in e ocena napake.
Za metodo reda točnosti n: $\mathcal{O}(h^n)$ pri koraku h velja:

$$f'(x_i) = f'_o(x_i, h) + K\,h^n,$$

kjer je K neznana konstanta.
Če korak razpolovimo in predostavimo, da se K ne spremeni, velja:

$$f'(x_i) = f'_o\left(x_i, \frac{h}{2}\right) + K\left(\frac{h}{2}\right)^n.$$

Iz obeh enačb izločimo konstanto K in določimo izboljšan približek:

$$\overline{f}'(x_i) = \frac{2^n\, f'_o\left(x_i, \frac{h}{2}\right) - f'_o(x_i, h)}{2^n - 1}$$

Zgled

Poglejmo si zgled $f(x) = \sin(x)$ (analitični odvod je: $f'(x) = \cos(x)$):

```
In [43]: x = np.linspace(0, 2*np.pi, 9)
         y = np.sin(x)
```

Pri koraku h imamo funkcijske vrednosti definirane pri:

```
In [44]: x
```

```
Out[44]: array([ 0.        ,  0.78539816,  1.57079633,  2.35619449,  3.14159265,
                 3.92699082,  4.71238898,  5.49778714,  6.28318531])
```

Numerični odvod pri $x = \pi$:

```
In [45]: x[4]
```

[3] http://docs.sympy.org/latest/modules/calculus/index.html#sympy.calculus.finite_diff.finite_diff_weights

Out[45]:

$$3.14159265359$$

in koraku h je

```
In [46]: h = x[1] - x[0]
         f_ocena_h = (-0.5*y[3] + 0.5*y[5])/h
         f_ocena_h
```

Out[46]:

$$-0.900316316157$$

Izračun pri koraku $2h$:

```
In [47]: h2 = x[2] - x[0]
         f_ocena_2h = (-0.5*y[2] + 0.5*y[6])/h2
         f_ocena_2h
```

Out[47]:

$$-0.636619772368$$

Izračunajmo izboljšano oceno za $x = \pi$:

```
In [48]: f_ocena_izboljšana = (2**2 * f_ocena_h - f_ocena_2h)/(2**2-1)
         f_ocena_izboljšana
```

Out[48]:

$$-0.988215164087$$

Vidimo, da je izboljšana ocena najbližje teoretični vrednosti $\cos(\pi) = -1$.

10.4 Necentralna diferenčna shema

Centralna diferenčna shema, ki smo jo spoznali zgoraj, je zelo uporabna in relativno natančna. Ker pa je ne moremo vedno uporabiti (recimo na začetku ali koncu tabele), si moramo pomagati z **necentralnimi diferenčnimi shemami** za računanje odvodov.

Poznamo:

- **diferenčno shemo naprej**, ki odvod točke aproksimira z vrednostmi funkcije v naslednjih točkah in
- **diferenčno shemo nazaj**, ki odvod točke aproksimira z vrednostmi v predhodnih točkah.

Izpeljave so podobne, kakor smo prikazali za centralno diferenčno shemo, zato jih tukaj ne bomo obravnavali in bomo prikazali samo končni rezultat.

10.4.1 Diferenčna shema naprej

Diferenčna shema naprej z redom napake $\mathcal{O}(h^1)$:

	y_i	y_{i+1}	y_{i+2}	y_{i+3}	y_{i+4}
$y_i' = \frac{1}{h}\cdot$	-1	1	0	0	0
$y_i'' = \frac{1}{h^2}\cdot$	1	-2	1	0	0
$y_i''' = \frac{1}{h^3}\cdot$	-1	3	-3	1	0
$y_i^{(4)} = \frac{1}{h^4}\cdot$	1	-4	6	-4	1

Diferenčna shema naprej z redom napake $\mathcal{O}(h^2)$:

	y_i	y_{i+1}	y_{i+2}	y_{i+3}	y_{i+4}	y_{i+5}
$y_i' = \frac{1}{2h}\cdot$	-3	4	-1	0	0	0
$y_i'' = \frac{1}{h^2}\cdot$	2	-5	4	-1	0	0
$y_i''' = \frac{1}{2h^3}\cdot$	-5	18	-24	14	-3	0
$y_i^{(4)} = \frac{1}{h^4}\cdot$	3	-14	26	-24	11	-2

10.4.2 Diferenčna shema nazaj

Diferenčna shema nazaj z redom napake $\mathcal{O}(h^1)$:

	y_{i-4}	y_{i-3}	y_{i-2}	y_{i-1}	y_i
$y_i' = \frac{1}{h}\cdot$	0	0	0	-1	1
$y_i'' = \frac{1}{h^2}\cdot$	0	0	1	-2	1
$y_i''' = \frac{1}{h^3}\cdot$	0	-1	3	-3	1
$y_i^{(4)} = \frac{1}{h^4}\cdot$	1	-4	6	-4	1

Diferenčna shema nazaj z redom napake $\mathcal{O}(h^2)$:

	y_{i-5}	y_{i-4}	y_{i-3}	y_{i-2}	y_{i-1}	y_i
$y_i' = \frac{1}{2h}\cdot$	0	0	0	1	-4	3
$y_i'' = \frac{1}{h^2}\cdot$	0	0	-1	4	-5	2
$y_i''' = \frac{1}{2h^3}\cdot$	0	3	-14	24	-18	5
$y_i^{(4)} = \frac{1}{h^4}\cdot$	-2	11	-24	26	-14	3

10.5 Uporaba `numpy.gradient`

Za izračun numeričnih odvodov (centralna diferenčna shema 2. reda) lahko uporabimo tudi `numpy.gradient()` (dokumentacija[4]):

```
gradient(f, *varargs, **kwargs)
```

kjer `f` predstavlja tabelo vrednosti (v obliki numeričnega polja) funkcije, katere odvod iščemo. `f` je lahko ene ali več dimenzij. Pozicijski parametri `varargs` definirajo razdaljo med vrednostmi argumenta funkcije

[4] http://docs.scipy.org/doc/numpy/reference/generated/numpy.gradient.html

`f`; privzeta vrednost je 1. Ta vrednost je lahko skalar, lahko pa tudi seznam vrednosti neodvisne spremenljivke (ali tudi kombinacija obojega). Gradientna metoda na robovih uporabi shemo naprej oziroma nazaj; parameter `edge_order` definira red sheme, ki se uporabi na robovih (izbiramo lahko med 1 ali 2, privzeta vrednost je 1).

Rezultat funkcije `gradient` je numerični seznam (ali seznam numeričnih seznamov) z izračunanimi odvodi.

Za podrobnosti glejte dokumentacijo[5].

Uporabo si bomo pogledali na primeru polinoma $p(x) = 3\,x^2 + x$:

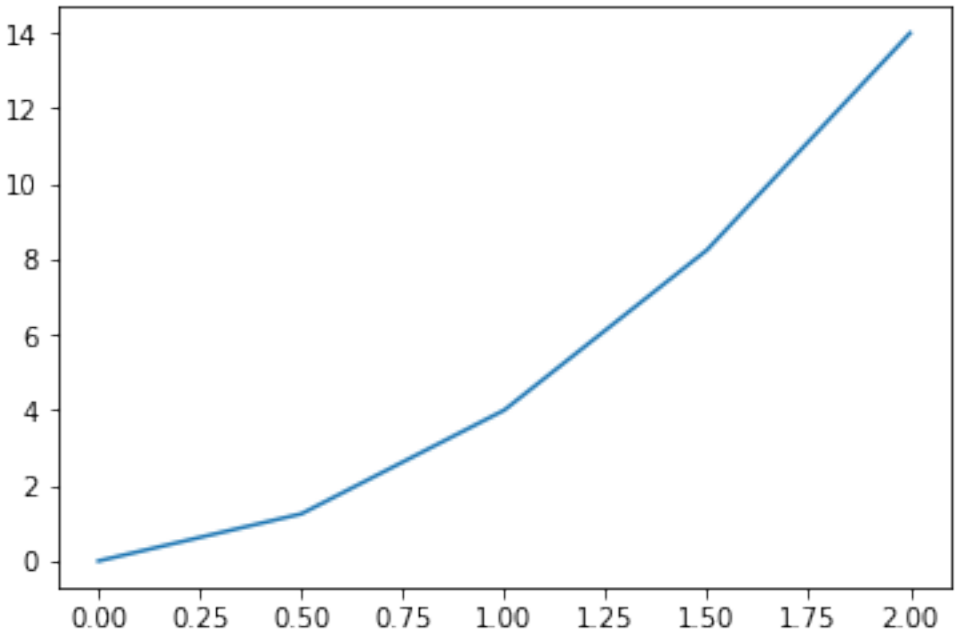
```
In [49]: p = np.poly1d([3,1,0])
         print(p)

   2
3 x + 1 x
```

Pripravimo tabelo vrednosti:

```
In [50]: x = np.array([0, 0.5, 1., 1.5, 2.])
         y = p(x)
         y

Out[50]: array([  0.  ,   1.25,   4.  ,   8.25,  14.  ])
```

Prikažimo podatke

```
In [51]: import matplotlib.pyplot as plt
         %matplotlib inline
         plt.plot(x, y);
```



[5] http://docs.scipy.org/doc/numpy/reference/generated/numpy.gradient.html

Poglejmo, kako izračunamo numerični odvod:

```
In [52]: np.gradient(y, 0.5)

Out[52]: array([  2.5,   4. ,   7. ,  10. ,  11.5])
```

Lahko bi podali tudi seznam vrednosti x:

```
In [53]: np.gradient(y, x)

Out[53]: array([  2.5,   4. ,   7. ,  10. ,  11.5])
```

Ne smemo pa pozabiti, da je privzeti koraj enako $\Delta x = 1$ in nam klic:

```
In [54]: np.gradient(y)

Out[54]: array([ 1.25,  2.  ,  3.5 ,  5.  ,  5.75])
```

vrne seveda napačni rezultat, saj nismo podali pravega koraka Δx.

10.6 Zaokrožitvena napaka pri numeričnem odvajanju

Zgoraj smo se osredotočili na napako metode. Pri numeričnem odvajanju pa moramo biti zelo pozorni tudi na **zaokrožitveno** (ali tudi *upodobitveno*) napako! Pogljemo si prvi odvod (po centralni diferenčni shemi) zapisan z napako metode ($k\,h^2$) in zaokrožitveno napako ε:

$$y_i' = \frac{1}{2h}\left((-y_{i-1} \pm \varepsilon) + (y_{i+1} \pm \varepsilon)\right) + k\,h^2$$

V najslabšem primeru se zaokrožitvena napaka sešteje in je skupna napaka:

$$n = \frac{\varepsilon}{h} + k\,h^2$$

Ko je h velik prevladuje napaka metode $k\,h^2$; ko pa je h majhen, pa prevladuje zaokrožitvena napaka. Napaka ima minimum, ko velja:

$$n' = -\frac{\varepsilon}{h^2} + 2\,k\,h = 0.$$

Sledi:

$$h = \sqrt[3]{\frac{\varepsilon}{2\,k}}.$$

10.6.1 Zgled

Spodaj si bomo pogledali primer, kjer bomo natančnost spreminjali v treh korakih:

1. `float16` - 16-bitni zapis: predznak 1 bit, 5 bitov eksponent, 10 bitov mantisa
2. `float32` - 32-bitni zapis: predznak 1 bit, 8 bitov eksponent, 23 bitov mantisa
3. `float64` - 64-bitni zapis: predznak 1 bit, 11 bitov eksponent, 52 bitov mantisa (to je privzeta natančnost).

Za več o tipih v `numpy` glejte dokumentacijo[6]

Določimo sedaj osnovno zaokrožitveno napako za posamezni tip:

```
In [55]: eps16 = np.finfo(np.float16).eps
         eps32 = np.finfo(np.float32).eps
         eps64 = np.finfo(np.float64).eps
         print(f'Osnovna zaokrožitvena napaka za tipe \
         `float16`, `float32` in `float64` je:\n{[eps16, eps32, eps64]}')
```

```
Osnovna zaokrožitvena napaka za tipe `float16`, `float32` in `float64` je:
[0.00097656, 1.1920929e-07, 2.2204460492503131e-16]
```

Kot primer si poglejmo seštevanje: k številu `1.` prištejemo polovico osnovne zaokrožitvene napake `eps16` in pretvorimo v tip `float16`, ugotovimo, da je nova vrednost še vedno enaka vrednosti `1.`:

```
In [56]: (1.+eps16/2).astype('float16')
```

```
Out[56]: 1.0
```

Definirajmo najprej funkcijo $\exp(x)$, ki bo dala rezultat natančnosti, ki jo definira parameter `dtype`:

```
In [57]: def fun(x, dtype=np.float): # funkcija
             return np.exp(-dtype(x)).astype(dtype)
```

Definirajmo še funkcijo za analitično določljiv odvod (to bomo pozneje potrebovali za določitev relativne napake):

```
In [58]: def f1_fun(x): # "točen" odvod funkcije
             return -np.exp(-x)
```

Podobno kakor zgoraj pri seštevanju, lahko tudi pri vrednosti funkcije ugotovimo, da sprememba vrednosti x, ki je manjša od ϵ, vodi v isti rezultat:

```
In [59]: fun(1., dtype=np.float16)
```

```
Out[59]: 0.36792
```

```
In [60]: fun(1+eps16/2, dtype=np.float16)
```

[6] http://docs.scipy.org/doc/numpy/user/basics.types.html

```
Out[60]: 0.36792
```

Uporabimo sedaj centralno diferenčno shemo za prvi odvod. Pri tem naj bodo števila zapisana z natančnostjo `dtype`, s pomočjo točnega odvoda pa se izračuna še relativna napaka:

```
In [61]: def f1_CDS(fun, x, h, dtype=np.float64):
             f1_ocena = (fun(x+h, dtype=dtype)-fun(x-h,dtype=dtype))/(2*dtype(h))
             f1_točno = f1_fun(x)
             relativna_napaka = (f1_točno - f1_ocena) / f1_točno
             return f1_ocena, relativna_napaka
```

Poglejmo primer odvoda pri $x = 1,0$ (Python funkcija vrne vrednost in relativno napako):

```
In [62]: f1_CDS(fun, x=1., h=.01, dtype=np.float16)
```

```
Out[62]:
```

$$(-0.366132723112, \quad 0.00474807196008)$$

```
In [63]: f1_CDS(fun, x=1., h=.01, dtype=np.float64)
```

```
Out[63]:
```

$$(-0.367885572526, \quad -1.66667500023e-05)$$

Definirajmo sedaj korak:

```
In [64]: h=0.25**np.arange(30)
         h[:10]
```

```
Out[64]: array([  1.00000000e+00,   2.50000000e-01,   6.25000000e-02,
                  1.56250000e-02,   3.90625000e-03,   9.76562500e-04,
                  2.44140625e-04,   6.10351562e-05,   1.52587891e-05,
                  3.81469727e-06])
```

Izračunamo oceno odvodov za različne natančnosti zapisa (zaradi deljenja z 0 dobimo opozorilo):

```
In [65]: f1_16 = f1_CDS(fun, x=1., h=h, dtype=np.float16)
         f1_32 = f1_CDS(fun, x=1., h=h, dtype=np.float32)
         f1_64 = f1_CDS(fun, x=1., h=h, dtype=np.float64)
```

```
C:\Users\Janko\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: RuntimeWarning: invalid value encou
```

Izrišemo različne tipe v odvisnosti od velikosti koraka h. Najprej uvozimo potrebne knjižnice:

```
In [66]: import matplotlib.pyplot as plt
         %matplotlib inline
```

Definirajmo sliko:

```
In [67]: def fig_ocena():
             plt.semilogx(h, f1_16[0], 'b', lw=3, alpha=0.5, label='float16')
             plt.semilogx(h, f1_32[0], 'r', lw=3, alpha=0.5, label='float32')
             plt.semilogx(h, f1_64[0], 'g', lw=3, alpha=0.5, label='float64=float')
             plt.title('Ocena odvoda za različne tipe natančnosti')
             plt.xlabel('$h$')
             plt.ylabel('Ocena odvoda: $-\exp(-1)=-0.3678794$')
             plt.ylim(-0.37, -0.365)
             plt.legend();
```

Prikažimo jo:

```
In [68]: fig_ocena()
```

Pri relativno velikem koraku h prevladuje napaka metode, pri majhnem koraku pa zaokrožitvnea napaka; optimalni korak lahko ocenimo glede na:

$$h = \sqrt[3]{\frac{\varepsilon}{2k}}.$$

V konkretnem primeru velja:

$$k = -\frac{f'''(x)}{6} = -\frac{-e^{-x}}{6} = \frac{1}{6e} \qquad (x = 1)$$

Sledi:

$$h = \sqrt[3]{3\,\varepsilon\,e}.$$

Izračunamo primeren korak za 16, 32 in 64-bitni zapis:

```
In [69]: np.power(3*eps16*np.exp(1),1/3)
```

Out[69]:

0.19969717751

```
In [70]: np.power(3*eps32*np.exp(1),1/3)
```

Out[70]:

0.0099062346767

```
In [71]: np.power(3*eps64*np.exp(1),1/3)
```

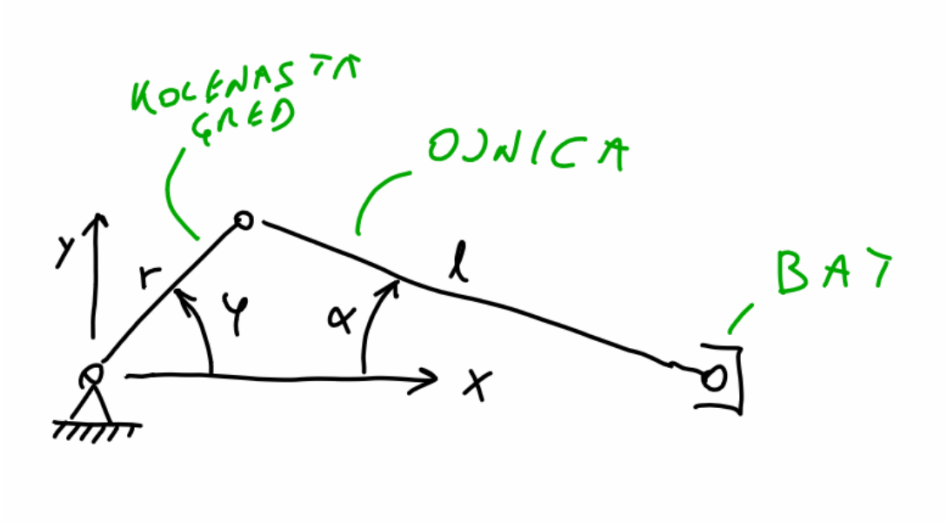
Out[71]:

$1.2188548432e-05$

Najbolje se izkaže 64-bitni zapis, vendar pa tudi pri tem korak manjši od cca `1e-5` ni priporočen!

10.7 Nekaj vprašanj za razmislek!

1. Za batni mehanizem na spodnji sliki strojno izpeljite kinematiko gibanja bata, če se kolenasta gred giblje po zakonu $\varphi(t) = \omega\,t$.



2. Za kotno hitrosti $\omega = 2\,\pi\,50\,\mathrm{rad/s}$ izrišite lego bata v treh obratih gredi. Uporabite: $r = 0,03\,\mathrm{m}$ in $l = 0,1\,\mathrm{m}$.
3. Simbolno odvajajte lego $x(t)$, da pridobite pospešek $\ddot{x}(t)$.
4. Pripravite funkcijo za klicanje simbolnih izrazov za lego $x(t)$ in pospešek $\ddot{x}(t)$ iz `numpy`.
5. S pomočjo `scipy` pripravite centralno diferenčno shemo za 2. odvod čez 3, 5, in 7 točk.
6. Raziščite funkcijo `numpy.convolve` in z njo na podlagi numeričnih vrednosti za x numerično izračunajte pospešek $\ddot{x}$. Kje je odvod pravilen?
7. S centralno diferenčno shemo 2. odvoda čez tri točke ste izračunali notranje točke, nastavite diferenčno shemo naprej za izračun prve točke z natančnostjo $\mathcal{O}(h^2)$.
8. Dodajte podatkom lege, določeno mero šuma in preverite, zakaj ni primerna uporaba numeričnega odvajanja na šumnih podatkih.
9. S centralno diferenčno shemo 2. odvoda čez tri točke ste doslej izračunali notranje točke, nastavite diferenčno shemo nazaj za izračun zadnje točke z natančnostjo $\mathcal{O}(h^2)$.
10. Raziščite vpliv časovnega koraka na izračun 2. odvoda.
11. Izmerjene imamo sledeče pozicije (gibanja) avtomobila:

    $t = [0., 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.]$ [h]

    $s = [0, 3, 10, 13, 17, 25, 33, 46, 58, 75]$ [km]

    Izračunajte hitrost in pospešek avtomobila pri času 0,5 h. Hitrost in pospešek prikažite tudi v grafični obliki.

Opomba: Dodatek k domačim nalogam: 6. vprašanje bi lahko nadaljevali in zašumljene podatke gladili ter nato izvedli odvajanje. Glajenje izvedite s konvolucijo med [0,21194156, 0,57611688, 0,21194156] in x.

10.8 Dodatno

Video predavanja na temo numeričnega odvajanja[7].

[7] https://www.youtube.com/watch?v=ZJkGI5DZQv8&list=PLYdroRCLMg5OvLx1EtY1ByvveJeTEXQd_&index=18

Poglavje 11

Numerično integriranje

11.1 Uvod

V okviru tega poglavja bomo za dano funkcijo $f(x)$ izračunali določen integral:

$$\int_a^b f(x)\,\mathrm{d}x$$

kjer sta a in b meji integriranja, $f(x)$ pa so vrednosti funkcije, ki jih pridobimo iz tabele vrednosti ali s pomočjo analitične funkcije.

Pri numeričnem integriranju integral ocenimo z I in velja

$$\int_a^b f(x)\,\mathrm{d}x = I + E,$$

kjer je E napaka ocene integrala.

Numerični integral bomo računali na podlagi diskretne vsote:

$$I = \sum_{i=0}^{m} A_i\, f(x_i),$$

kjer so A_i uteži, x_i pa vozlišča na intervalu $[a, b]$ in je $m + 1$ število vozlišč.

Ogledali si bomo dva različna pristopa k numerični integraciji:

1. *Newton-Cotesov pristop*, ki temelji na ekvidistantnih vozliščih (konstanten korak integracije) in
2. *Gaussov integracijski pristop*, kjer so vozlišča postavljena tako, da se doseže natančnost za polinome.

11.1.1 Motivacijski primer

Pri numeričnem integriranju si bomo pomagali s konkretnim primerom:

$$\int_1^2 x\,\sin(x)\,\mathrm{d}x$$

Pripravimo si vozlišča. Osnovni korak naj bo $h = 0.25$, v tem primeru imamo štiri štiri podintervale in pet vozlišč, pri koraku $2h$ so tri vozliščne točke in in pri koraku $4h$ samo dve (skrajni):

```
In [1]: import numpy as np
        xg, hg = np.linspace(1, 2, 100, retstep=True) # goste točke (za prikaz)
        x2v, h2v = np.linspace(1, 2, 2, retstep=True)  # korak h2v = 1 (2 vozlišči)
        x3v, h3v = np.linspace(1, 2, 3, retstep=True)  # korak h3v = 0.5 (3 vozlišča)
        x4v, h4v = np.linspace(1, 2, 4, retstep=True)  # korak h4v = 0.33.. (4 vozlišča)
        x5v, h5v = np.linspace(1, 2, 5, retstep=True)  # korak h5v = 0.25 (5 vozlišč)
```

Pripravimo še funkcijske vrednosti:

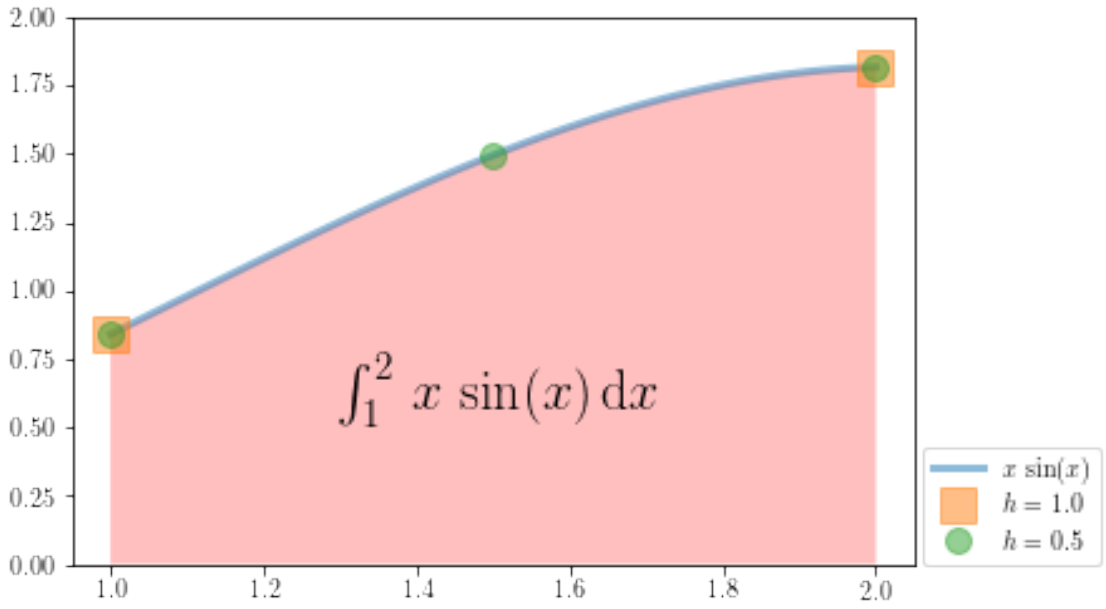
```
In [2]: yg = xg * np.sin(xg)
        y2v = x2v * np.sin(x2v)
        y3v = x3v * np.sin(x3v)
        y4v = x4v * np.sin(x4v)
        y5v = x5v * np.sin(x5v)
```

Pripravimo prikaz podatkov:

```
In [3]: import matplotlib.pyplot as plt
        from matplotlib import rc # to uvozimo, da so fonti na sliki latex ustrezni
        rc('font',**{'family':'sans-serif','sans-serif':['Helvetica']})
        rc('text', usetex=True)
        %matplotlib inline
        def fig():
            plt.fill_between(xg, yg, alpha=0.25, facecolor='r')
            plt.annotate('$\int_1^2\,x\,\sin(x)\,\\textrm{d}x$', (1.3, 0.5), fontsize=22)
            plt.plot(xg, yg, lw=3, alpha=0.5, label='$x\,\sin(x)$')
            plt.plot(x2v, y2v, 's', alpha=0.5, label=f'$h={h2v}$', markersize=14)
            plt.plot(x3v, y3v, 'o', alpha=0.5, label=f'$h={h3v}$', markersize=10)
            plt.legend(loc=(1.01, 0))
            plt.ylim(0, 2)
            plt.show()
```

Prikažimo podatke:

```
In [4]: fig()
```



Analitično izračunajmo točen rezultat:

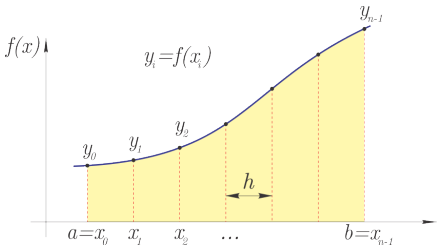
```
In [5]: import sympy as sym
        sym.init_printing()
        x = sym.symbols('x')
        I_točno = float(sym.integrate(x*sym.sin(x), (x, 1, 2)).evalf())
        I_točno
```

Out[5]:

1.4404224209802097

11.2 Newton-Cotesov pristop

Na sliki je prikazan splošen primer, kjer je razdalja med vozlišči x_i enaka h (gre za **ekvidistantno delitev**).



V okviru tega poglavja si bomo najprej pogledali trapezno ter sestavljeno trapezno pravilo, pozneje pa si bomo pogledali Simpsonovo ter Romergovo metode.

11.2.1 Trapezno pravilo

Trapezno pravilo vrednosti podintegralske funkcije $f()x$ na (pod)intervalu $[x_0, x_1]$ interpolira z linearno funkcijo. Za dve vozliščni točki to pomeni, da površino pod grafom funkcije približno izračunamo kot:

$$I_{\text{trapezno}} = \sum_{i=0}^{n=1} A_i f(x_i) = \frac{h}{2} \cdot (f(x_0) + f(x_1)) .$$

In so uteži:

$$A_0 = A_1 = \frac{1}{2} h.$$

Numerična implementacija

Numerična implementacija je:

```
In [6]: def trapezno(y, h):
            """
            Trapezno pravilo integriranja.

            :param y: funkcijske vrednosti
            :param h: korak integriranja
            """
            return (y[0] + y[-1])*h/2
```

Numerični zgled

V konkretnem primeru to pomeni, da prvo in zadnjo funkcijsko vrednost utežimo s $h/2$. V našem primeru je $h = 1$:

```
In [7]: I_trapezno = trapezno(y2v, h=h2v)
        I_trapezno
```

```
Out[7]:
```

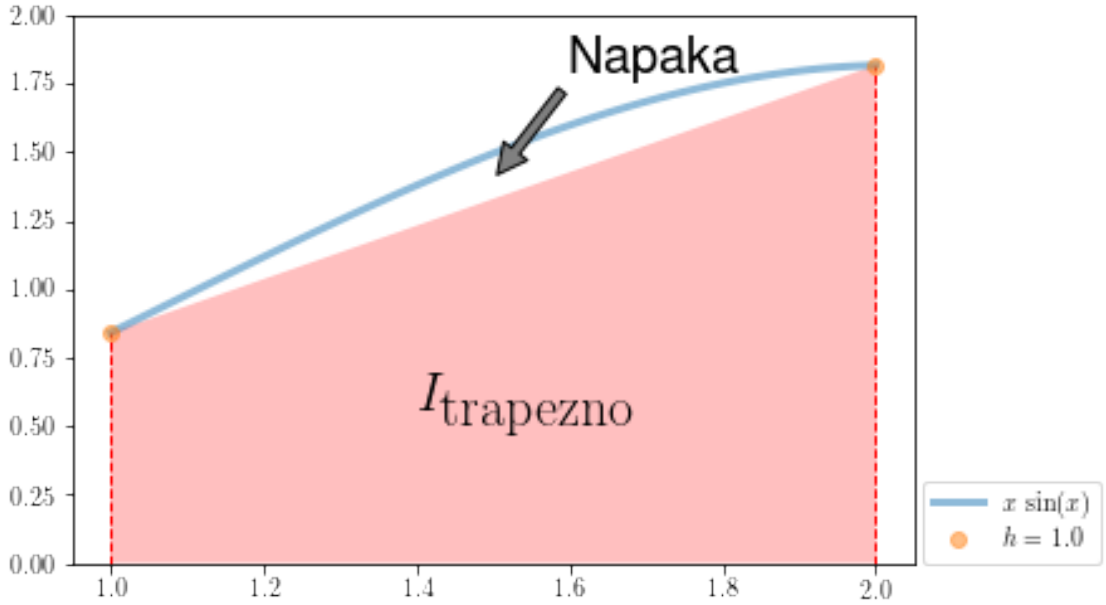
1.33003291923

Pripravimo sliko:

```
In [8]: def fig_trapezno():
            plt.fill_between(x2v, y2v, alpha=0.25, facecolor='r')
            plt.vlines(x2v, 0, y2v, color='r', linestyles='dashed', lw=1)
            plt.annotate('$I_{\\textrm{trapezno}}$', (1.4, 0.5), fontsize=22)
            plt.annotate('Napaka', fontsize=20, xy=(1.5, 1.4), xytext=(1.6, 1.8),
                    arrowprops=dict(facecolor='gray', shrink=0.05))
            plt.plot(xg, yg, lw=3, alpha=0.5, label='$x\,\sin(x)$')
            plt.plot(x2v, y2v, 'o', alpha=0.5, label=f'$h={h2v}$')
            plt.legend(loc=(1.01, 0))
            plt.ylim(0, 2)
            plt.show()
```

Prikažemo:

```
In [9]: fig_trapezno()
```



Napaka trapeznega pravila

Razlika med analitično vrednostjo integrala in numeričnim približkom I je napaka metode:

$$E = \int_a^b f(x)\,dx - I,$$

Če je funkcija $f(x)$ vsaj dvakrat odvedljiva, se lahko (glejte npr. vir: Burden, Faires, Burden: Numerical Analysis 10th Ed) izpelje ocena napake, ki velja samo za trapezni približek prek celega integracijskega intervala:

$$E_{\text{trapezno}} = -\frac{h^3}{12} f''(\xi),$$

kjer je $h = b - a$ in ξ neznana vrednost na intervalu $[a, b]$.

11.2.2 Sestavljeno trapezno pravilo

Če razdelimo interval $[a, b]$ na n podintervalov in na vsakem uporabimo trapezno pravilo integriranja, govorimo o *sestavljenem trapeznem pravilu* (angl. *composite trapezoidal rule*).

V tem primeru za vsak podinterval i uporabimo trapezno pravilo in torej za meje podinterval x_i in x_{i+1} uporabimo uteži $A_i = A_{i+i} = h/2$. Ker so notranja vozlišča podvojena, sledi:

$$A_0 = A_n = \frac{h}{2} \quad \text{in za ostala vozlišča:} \quad A_i = h.$$

Pri tem smo predpostavili podintervale enake širine:

$$h = \frac{x_n - x_0}{n}$$

Sledi torej:

$$I_{\text{trapezno sest}} = \sum_{i=0}^{n} A_i f(x_i) = \left(\frac{y_0}{2} + y_1 + y_2 + \cdots + y_{n-1} + \frac{y_n}{2}\right) h.$$

Numerična implementacija

Numerična implementacija je:

```
In [10]: def trapezno_sest(y, h):
             """
             Sestavljeno trapezno pravilo integriranja.

             :param y: funkcijske vrednosti
             :param h: korak integriranja
             """
             return (np.sum(y) - 0.5*y[0] - 0.5*y[-1])*h
```

Numerični zgled

Zgoraj smo že pripravili podatke za dva podintervala (tri vozlišča):

```
In [11]: x3v
```

```
Out[11]: array([ 1. ,  1.5,  2. ])
```

```
In [12]: h3v
```

```
Out[12]:
```

0.5

Izračunajmo oceno integrala s sestavljenim trapeznim pravilom:

```
In [13]: I_trapezno_sest = trapezno_sest(y3v, h=h3v)
         I_trapezno_sest
```

```
Out[13]:
```

1.41313769957

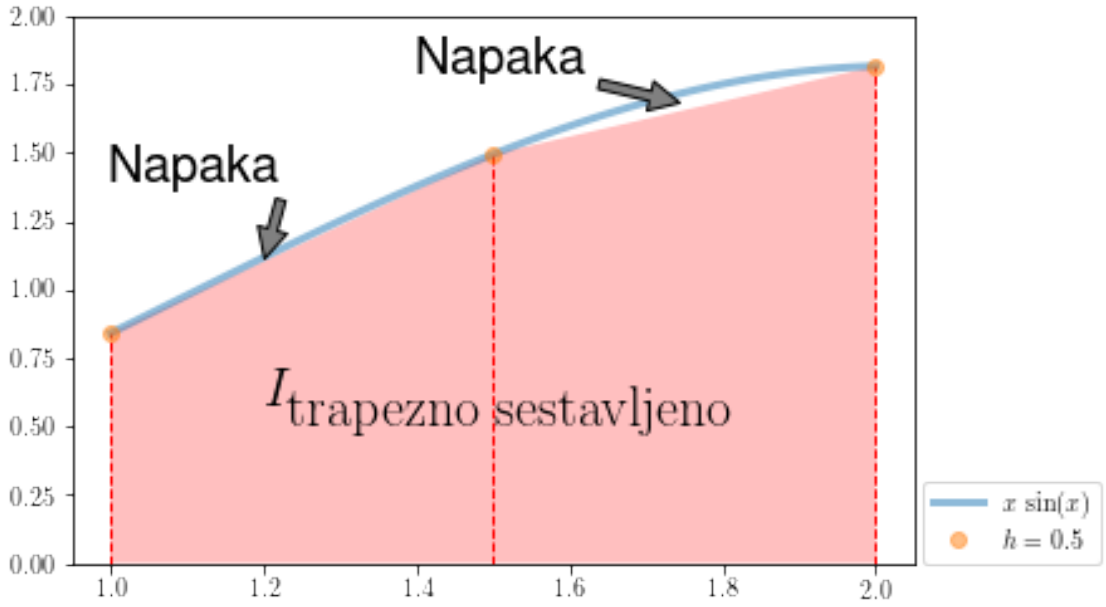
Pripravimo sliko:

```
In [14]: def fig_trapezno_sest():
             plt.fill_between(x3v, y3v, alpha=0.25, facecolor='r')
             plt.vlines(x3v, 0, y3v, color='r', linestyles='dashed', lw=1)
```

```
        plt.annotate('$I_{\\textrm{trapezno sestavljeno}}$', (1.2, 0.5), fontsize=22)
        plt.annotate('Napaka', fontsize=20, xy=(1.75, 1.68), xytext=(1.4, 1.8),
                arrowprops=dict(facecolor='gray', shrink=0.05))
        plt.annotate('Napaka', fontsize=20, xy=(1.2, 1.1), xytext=(1., 1.4),
                arrowprops=dict(facecolor='gray', shrink=0.05))
        plt.plot(xg, yg, lw=3, alpha=0.5, label='$x\,\sin(x)$')
        plt.plot(x3v, y3v, 'o', alpha=0.5, label=f'$h={h3v}$')
        plt.legend(loc=(1.01, 0))
        plt.ylim(0, 2)
        plt.show()
```

Prikažemo:

```
In [15]: fig_trapezno_sest()
```



Napaka sestavljenega trapeznega pravila

Napaka sestavljenega trapeznega pravila izhaja iz napake trapeznega pravila; pri tem tako napako naredimo n-krat.

Ker velja $h \cdot n = b - a$, izpeljemo napako sestavljenega trapeznega pravila kot:

$$E_{\text{trapezno sest}} = -\frac{h^2(b-a)}{12} f''(\eta),$$

η je vrednost na intervalu $[a, b]$. Napaka je drugega reda $\mathcal{O}(h^2)$.

Boljši približek integrala

V nadaljevanju si bomo pogledali t. i. **Richardsonovo ekstrapolacijo**, pri kateri na podlagi ocene integrala s korakom h in $2h$ izračunamo boljši približek.

V kolikor integral I numerično izračunamo pri dveh različnih korakih h in $2\,h$, velja:

$$\int_a^b f(x)\,\mathrm{d}x = I_h + E_h = I_{2h} + E_{2h},$$

kjer sta I_h in I_{2h} približka integrala s korakom h in $2h$ ter E_h in E_{2h} oceni napake pri koraku h in $2h$. Izpeljemo $I_{2h} - I_h = E_h - E_{2h}$

Naprej zapišemo:

$$E_h = -\frac{h^2(b-a)}{12} f''(\eta) = h^2\,K.$$

Ob predpostavki, da je $f''(\eta)$ pri koraku h in $2h$ enak (η je pri koraku h in $2h$ dejansko različen), zapišemo:

$$E_{2h} = -\frac{(2h)^2(b-a)}{12} f''(\eta) = 4\,h^2\,K$$

Sledi:

$$I_{2h} - I_h = -3K\,h^2.$$

Sedaj lahko ocenimo napako pri koraku h:

$$E_h = h^2\,K = \frac{I_h - I_{2h}}{3}.$$

Na podlagi ocene napake lahko izračunamo boljši numerični približek I_h^*:

$$I_h^* = I_h + \frac{1}{3}\,(I_h - I_{2h})$$

ali

$$I_h^* = \frac{4}{3}\,I_h - \frac{1}{3}\,I_{2h}$$

Numerični zgled

Predhodno smo s trapeznim pravilom že izračunali integral pri koraku $h = 1$ in pri koraku $h = 0,5$, rezultata sta bila:

```
In [16]: [I_trapezno, I_trapezno_sest]
```

```
Out[16]:
```

$$[1.33003291923,\quad 1.41313769957]$$

S pomočjo zgornje formule izračunamo boljši približek:

```
In [17]: I_trapezno_boljši = 4/3*I_trapezno_sest - 1/3*I_trapezno
         print(f'Točen rezultat:   {I_točno}\nBoljši približek: {I_trapezno_boljši}')
```

```
Točen rezultat:   1.4404224209802097
Boljši približek: 1.4408392930139313
```

`numpy` implementacija sestavljenega trapeznega pravila

Sestavljeno trapezno pravilo je implementirano tudi v paketu `numpy`, s funkcijo `numpy.trapz`:

```
trapz(y, x=None, dx=1.0, axis=-1)
```

- `y` predstavlja tabelo funkcijskih vrednosti,
- `x` je opcijski parameter in definira vozlišča; če parameter ni definiran, se privzame ekvidistančna vozlišča na razdalji `dx`,
- `dx` definira (konstanten) korak integracije, ima privzeto vrednost 1,
- `axis` definira *os* po kateri se integrira (v primeru, da je `y` večdimenzijsko numerično polje).

Funkcija vrne izračunani integral po sestavljenem trapeznem pravilu. Več informacij lahko najdete v dokumentaciji[1].

Poglejmo si primer:

```
In [18]: #%%timeit
         I_trapezno_np = np.trapz(y3v, dx=h3v)
         I_trapezno_np
```

```
Out[18]:
```

1.41313769957

11.2.3 Simpsonova in druge metode

Zgoraj smo si pogledali trapezno pravilo, ki temelji na linearni interpolacijski funkciji na posameznem podintervalu. Z interpolacijo višjega reda lahko izpeljemo še druge integracijske metode.

Izračunati moramo:

$$\int_a^b f(x)\,dx.$$

Tabeliramo podintegralsko funkcijo $f(x)$ in tabelo interpoliramo z Lagrangevim interpolacijskim polinomom $P_n(x)$ stopnje n:

$$P_n(x) = \sum_{i=0}^{n} f(x_i)\, l_i(x),$$

kjer so $y_i = f(x_i)$ funkcijske vrednosti v vozliščih x_i in je Lagrangev polinom l_i definiran kot:

[1] https://docs.scipy.org/doc/numpy/reference/generated/numpy.trapz.html

$$l_i(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}.$$

Za numerični izračun integrala $\int_a^b f(x)\,dx$ (meje so: $a = x_0$, $b = x_n$) namesto funkcije $f(x)$ vstavimo v integral interpolacijski polinom $P_n(x)$:

$$I = \int_{x_0}^{x_n} P_n(x)\,dx = \int_{x_0}^{x_n} \sum_{i=0}^{n} f(x_i)\, l_i(x)\,dx.$$

Ker je integriranje linearna operacija, lahko zamenjamo integriranje in vsoto:

$$I = \sum_{i=0}^{n} f(x_i) \underbrace{\int_{x_0}^{x_n} l_i(x)\,dx}_{A_i}.$$

Lagrangev polinom integriramo in dobimo uteži A_i:

$$A_i = \int_{x_0}^{x_n} l_i(x)\,dx$$

Izpeljava trapeznega pravila z uporabo Lagrangevih polinomov

Poglejmo si kako z Lagrangevim interpolacijskim polinomom prve stopnje strojno izpeljemo uteži A_i v primeru trapeznega pravila.

Najprej v simbolni obliki definirajmo spremenljivko `x`, vozlišči `x0` in `x1` ter korak `h`:

```
In [19]: x, x0, x1, h = sym.symbols('x x0, x1, h')
```

Pripravimo Python funkcijo, ki v simbolni obliki vrne seznam n koeficientov Lagrangevih polinomov $[l_0(x), l_1(x), \ldots, l_{n-1}(x)]$ stopnje $n-1$:

```
In [20]: def lagrange(n, x, vozlišča_predpona='x'):
             if isinstance(vozlišča_predpona, str):
                 vozlišča = sym.symbols('{0:s}:{1:g}'.format(vozlišča_predpona, n))
             coeffs = []
             for i in range(0, n):
                 numer = []
                 denom = []

                 for j in range(0, n):
                     if i == j:
                         continue

                     numer.append(x    - vozlišča[j])
                     denom.append(vozlišča[i] - vozlišča[j])
```

```
            numer = sym.Mul(*numer)
            denom = sym.Mul(*denom)

            coeffs.append(numer/denom)
        return coeffs
```

Najprej poglejmo Lagrangeva polinoma za linearno interpolacijo ($n = 2$):

```
In [21]: lag = lagrange(2, x)
         lag
```

Out[21]:

$$\left[\frac{x - x_1}{x_0 - x_1}, \quad \frac{x - x_0}{-x_0 + x_1}\right]$$

Sedaj Lagrangev polinom $l_0(x)$ integriramo čez celotni interval:

```
In [22]: int0 = sym.integrate(lag[0], (x, x0, x1))
         int0
```

Out[22]:

$$-\frac{x_0^2}{2x_0 - 2x_1} + \frac{x_0 x_1}{x_0 - x_1} + \frac{x_1^2}{2x_0 - 2x_1} - \frac{x_1^2}{x_0 - x_1}$$

Izraz poenostavimo in dobimo:

```
In [23]: int1 = int0.factor()
         int1
```

Out[23]:

$$-\frac{1}{2}(x_0 - x_1)$$

Ker je širina podintervala konstantna je $x_1 = h_0 + h$, izvedemo zamenjavo:

```
In [24]: zamenjave = {x1: x0+h}
         int1.subs(zamenjave)
```

Out[24]:

$$\frac{h}{2}$$

Zgornje korake za Lagrangev polinom $l_0(x)$ lahko posplošimo za seznam Lagrangevih polinomov:

```
In [25]: x, x0, x1, h = sym.symbols('x, x0, x1, h')
         zamenjave = {x1: x0+h}
         A_trapez = [sym.integrate(li, (x, x0, x1)).factor().subs(zamenjave)
                     for li in lagrange(2, x)]  # za vsak lagrangev polimom `li` v seznamu lagrange(2,x
         A_trapez
```

Out[25]:

$$\left[\frac{h}{2}, \quad \frac{h}{2}\right]$$

Izpeljali smo uteži, ki smo jih uporabili pri trapezni metodi:

$$A_0 = h/2 \qquad A_1 = h/2.$$

Trapezno pravilo je:

$$I_{\text{trapezno}} = \frac{h}{2}\left(y_0 + y_1\right)$$

Ocena napake je (vir: Burden, Faires, Burden: Numerical Analysis 10th Ed):

$$E_{\text{trapezno}} = -\frac{h^3}{12} f''(\xi).$$

Izračun uteži za Simpsonovo pravilo

Potem ko smo zgoraj pokazali strnjen izračun za trapezno pravilo, lahko podobno izvedemo za kvadratno interpolacijo čez tri točke ($n = 3$).

Izračun uteži je analogen zgornjemu:

```
In [26]: x, x0, x1, x2, h = sym.symbols('x, x0, x1, x2, h')
         zamenjave = {x1: x0+h, x2: x0+2*h}
         A_Simpson1_3 = [sym.integrate(li, (x, x0, x2)).factor().subs(zamenjave).factor()
                         for li in lagrange(3, x)]
         A_Simpson1_3
```

Out[26]:

$$\left[\frac{h}{3}, \quad \frac{4h}{3}, \quad \frac{h}{3}\right]$$

Simpsonovo pravilo je:

$$I_{\text{Simpsonovo}} = \frac{h}{3}\left(y_0 + 4\,y_1 + y_2\right)$$

Ocena napake je (vir: Burden, Faires, Burden: Numerical Analysis 10th Ed):

$$E_{\text{Simpsonovo}} = -\frac{h^5}{90} f^{(4)}(\xi).$$

Primer uporabe:

```
In [27]: I_Simps = h3v/3 * np.sum(y3v * [1, 4, 1])
         I_Simps
```

Out[27]:

1.44083929301

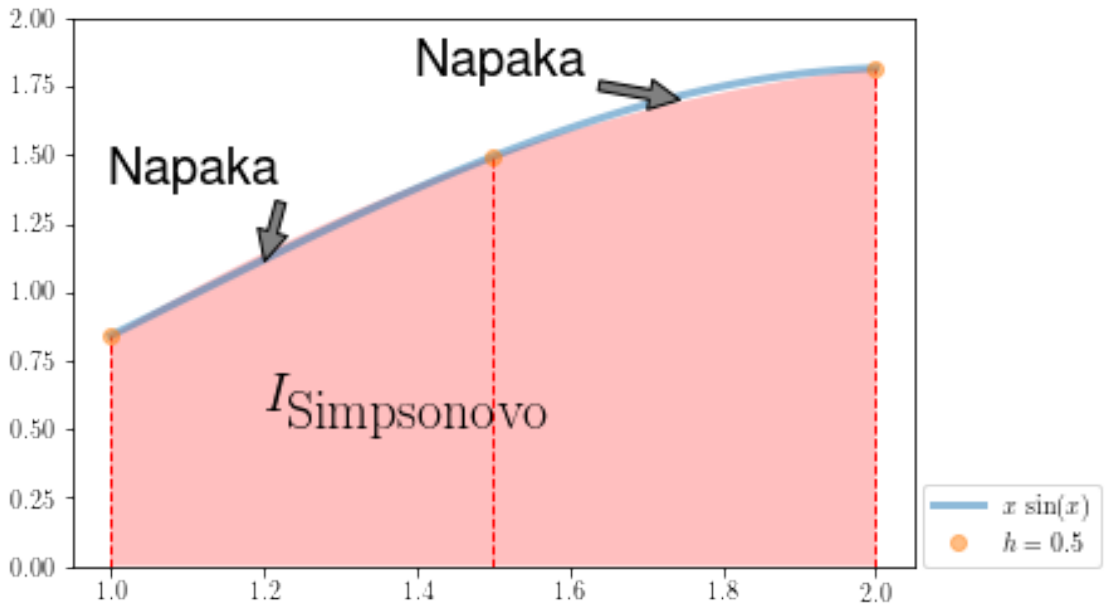
Pripravimo sliko. Ker Simsonovo pravilo temelji na kvadratni interpolaciji, moramo naprej pripraviti interpolacijski polinom (pomagamo si s `scipy.interpolate`):

```
In [28]: from scipy import interpolate
```

```
In [29]: def fig_Simpsonovo():
             y_interpolate = interpolate.lagrange(x3v, y3v)
             plt.fill_between(xg, y_interpolate(xg), alpha=0.25, facecolor='r')
             plt.vlines(x3v, 0, y3v, color='r', linestyles='dashed', lw=1)
             plt.annotate('$I_{\\textrm{Simpsonovo}}$', (1.2, 0.5), fontsize=22)
             plt.annotate('Napaka', fontsize=20, xy=(1.75, 1.7), xytext=(1.4, 1.8),
                     arrowprops=dict(facecolor='gray', shrink=0.05))
             plt.annotate('Napaka', fontsize=20, xy=(1.2, 1.1), xytext=(1., 1.4),
                     arrowprops=dict(facecolor='gray', shrink=0.05))
             plt.plot(xg, yg, lw=3, alpha=0.5, label='$x\,\sin(x)$')
             plt.plot(x3v, y3v, 'o', alpha=0.5, label=f'$h={h3v}$')
             plt.legend(loc=(1.01, 0))
             plt.ylim(0, 2)
             plt.show()
```

Prikažemo:

```
In [30]: fig_Simpsonovo()
```



`scipy.integrate.newton_cotes`

Koeficiente integracijskega pristopa Newton-Cotes pridobimo tudi s pomočjo `scipy.integrate.newton_cotes()` dokumentacija[2]:

```
newton_cotes(rn, equal=0)
```

kjer sta parametra:

- `rn`, ki definira število podintervalov (mogoč je tudi nekonstanten korak, glejte dokumentacijo[3]),
- `equal`, ki definira ali se zahteva konstantno široke podintervale.

Funkcija vrne terko, pri čemer prvi element predstavlja numerično polje uteži in drugi člen oceno napake.

Poglejmo si primer:

```
In [31]: from scipy import integrate
         integrate.newton_cotes(3)
```

```
Out[31]: (array([ 0.375,  1.125,  1.125,  0.375]), -0.0375)
```

Izračun uteži za Simpsonovo 3/8 pravilo

Nadaljujemo lahko s kubično interpolacijo čez štiri točke ($n = 4$):

```
In [32]: x, x0, x1, x2, x3, h = sym.symbols('x, x0, x1, x2, x3, h')
         zamenjave = {x1: x0+h, x2: x0+2*h, x3: x0+3*h}
         A_Simpson3_8 = [sym.integrate(li, (x, x0, x3)).factor().subs(zamenjave).factor()
                         for li in lagrange(4, x)]
         A_Simpson3_8
```

```
Out[32]:
```

$$\left[\frac{3h}{8}, \quad \frac{9h}{8}, \quad \frac{9h}{8}, \quad \frac{3h}{8}\right]$$

Simpsonovo 3/8 pravilo je:

$$I_{\text{Simpsonovo 3/8}} = \frac{3h}{8}\left(y_0 + 3\,y_1 + 3\,y_2 + y_3\right)$$

Ocena napake je (vir: Burden, Faires, Burden: Numerical Analysis 10th Ed):

$$E_{\text{Simpsonovo 3/8}} = -\frac{3\,h^5}{8a0} f^{(4)}(\xi).$$

Poglejmo si primer uporabe. Uporabimo pripravjeno tabelo vrednosti funkcije v štirih točkah:

[2] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.newton_cotes.html#scipy.integrate.newton_cotes

[3] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.newton_cotes.html#scipy.integrate.newton_cotes

```
In [33]: y4v
```

```
Out[33]: array([ 0.84147098,  1.2959172 ,  1.65901326,  1.81859485])
```

```
In [34]: I_Simps38 = 3*h4v/8 * np.sum(y4v * [1, 3, 3, 1])
         I_Simps38
```

```
Out[34]:
```

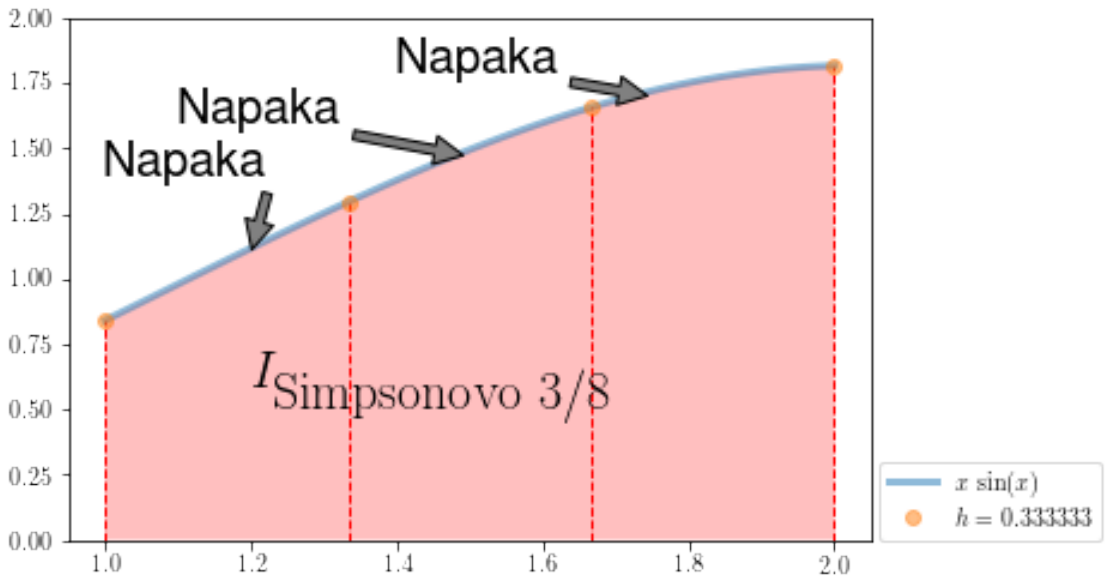
1.44060715408

Pripravimo še prikaz:

```
In [35]: def fig_Simpsonovo38():
             y_interpolate = interpolate.lagrange(x4v, y4v)
             plt.fill_between(xg, y_interpolate(xg), alpha=0.25, facecolor='r')
             plt.vlines(x4v, 0, y4v, color='r', linestyles='dashed', lw=1)
             plt.annotate('$I_{\\textrm{Simpsonovo 3/8}}$', (1.2, 0.5), fontsize=22)
             plt.annotate('Napaka', fontsize=20, xy=(1.75, 1.7), xytext=(1.4, 1.8),
                     arrowprops=dict(facecolor='gray', shrink=0.05))
             plt.annotate('Napaka', fontsize=20, xy=(1.5, 1.47), xytext=(1.1, 1.6),
                     arrowprops=dict(facecolor='gray', shrink=0.05))
             plt.annotate('Napaka', fontsize=20, xy=(1.2, 1.1), xytext=(1., 1.4),
                     arrowprops=dict(facecolor='gray', shrink=0.05))
             plt.plot(xg, yg, lw=3, alpha=0.5, label='$x\,\sin(x)$')
             plt.plot(x4v, y4v, 'o', alpha=0.5, label=f'$h={h4v:.6f}$')
             plt.legend(loc=(1.01, 0))
             plt.ylim(0, 2)
             plt.show()
```

Prikažemo:

```
In [36]: fig_Simpsonovo38()
```



11.2.4 Sestavljeno Simpsonovo pravilo

Interval $[a, b]$ razdelimo na sodo število n podintervalov enake širine $h = (b-a)/n$, s čimer so definirana vozlišča $x_i = a + i\,h$ za $i = 0, 1, \ldots, n$. V tem primeru zapišemo sestavljeno Simpsonovo pravilo:

$$\int_a^b f(x)\,\mathrm{d}x = \frac{h}{3}\left(f(x_0) + 4\sum_{i=1}^{n/2} f(x_{2i-1}) + 2\sum_{i=1}^{n/2-1} f(x_{2i}) + f(x_n)\right) \underbrace{-\frac{b-a}{180}h^4 f^{(4)}(\eta)}_{E_{\text{Sestavljeno Simpsonovo 1/3}}},$$

kjer je η neznana vrednost na intervalu $[a, b]$. Napaka je četrtega reda $\mathcal{O}(h^4)$.

Numerična implementacija:

```
In [37]: def simpsonovo_sest(y, h):
             """
             Sestavljeno Simpsonovo pravilo integriranja.

             :param y: funkcijske vrednosti
             :param h: korak integriranja
             """
             return h/3 * (y[0] + 4*np.sum(y[1:-1:2]) + 2*np.sum(y[2:-1:2]) + y[-1])
```

```
In [38]: I_Simps_sest = simpsonovo_sest(y5v, h=h5v)
         I_Simps_sest
```

Out[38]:

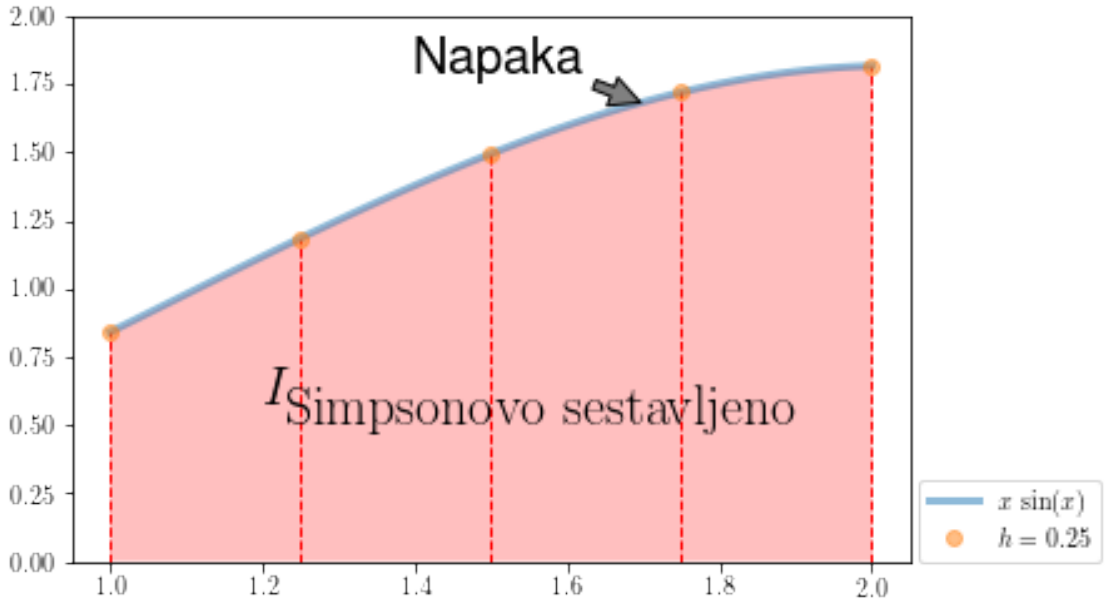
1.44044796026

Pripravimo sliko:

```
In [39]: def fig_Simpsonovo_sest():
             y_interpolate = interpolate.lagrange(x5v, y5v)
             plt.fill_between(xg, y_interpolate(xg), alpha=0.25, facecolor='r')
             plt.vlines(x5v, 0, y5v, color='r', linestyles='dashed', lw=1)
             plt.annotate('$I_{\\textrm{Simpsonovo sestavljeno}}$', (1.2, 0.5), fontsize=22)
             plt.annotate('Napaka', fontsize=20, xy=(1.70, 1.68), xytext=(1.4, 1.8),
                     arrowprops=dict(facecolor='gray', shrink=0.05))
             plt.plot(xg, yg, lw=3, alpha=0.5, label='$x\,\sin(x)$')
             plt.plot(x5v, y5v, 'o', alpha=0.5, label=f'$h={h5v}$')
             plt.legend(loc=(1.01, 0))
             plt.ylim(0, 2)
             plt.show()
```

Prikažemo:

```
In [40]: fig_Simpsonovo_sest()
```



Boljša ocena integrala

Napaka sestavljene Simpsonove metode je definirana z:

$$E_{\text{Sestavljeno Simpsonovo 1/3}} = -\frac{b-a}{180} h^4 f^{(4)}(\eta),$$

kjer je η neznana vrednost na intervalu $[a, b]$.

Izboljšano oceno integrala določimo na podoben način kakor pri sestavljeni trapezni metodi; integral I ocenjujemo pri dveh različnih korakih h in $2\,h$, velja natančno:

$$\int_a^b f(x)\,\mathrm{d}x = I_h + E_h = I_{2h} + E_{2h},$$

kjer je I_h približek integrala s korakom h in E_h ocena napake pri koraku h; analogno velja za I_{2h} in E_{2h}.

Če predpostavimo, da je $f^{(4)}(\eta)$ v obeh primerih enak, lahko določimo razliko $I_{2h} - I_h = E_h - E_{2h}$.

Naprej zapišemo:

$$E_h = -\frac{b-a}{180} h^4 f^{(4)}(\eta) = h^4\,K.$$

Ob predpostavki, da je $f^(4)(\eta)$ pri koraku h in $2h$ enak (η je pri koraku h in $2h$ dejansko različen), zapišemo:

$$E_{2h} = -\frac{(b-a)}{180}(2h)^4 f^{(4)}(\eta) = 16\,h^2\,K$$

Sledi:

$$I_{2h} - I_h = -15K\,h^4.$$

Sedaj lahko ocenimo napako pri koraku h:

$$E_h = h^4\,K = \frac{I_h - I_{2h}}{15}.$$

Na podlagi ocene napake lahko izračunamo boljši približek I_h^*:

$$I_h^* = I_h + \frac{1}{15}\,(I_h - I_{2h})$$

ali

$$I_h^* = \frac{16}{15}\,I_h - \frac{1}{15}\,I_{2h}$$

Numerični zgled

Predhodno smo s Simpsonovim pravilom že izračunali integral pri koraku $h = 0,5$ in pri koraku $h = 0,25$, rezultata sta bila:

```
In [41]: [I_Simps, I_Simps_sest]
```

```
Out[41]:
```

$$[1.44083929301, \quad 1.44044796026]$$

S pomočjo zgornje formule izračunamo boljši približek:

```
In [42]: I_Simps_boljši = 16/15*I_Simps_sest - 1/15*I_Simps
         print(f'Točen rezultat:   {I_točno}\nBoljši približek: {I_Simps_boljši}')
```

```
Točen rezultat:   1.4404224209802097
Boljši približek: 1.4404218714139077
```

Pridobimo boljši numerični približek, izgubimo pa oceno napake!

Simpsonova metoda v `scipy.integrate.simps`

V `scipy` je implementirano sestavljeno Simpsonovo pravilo v `scipy.integrate.simps()` (dokumentacija[4]):

[4] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.simps.html

```
simps(y, x=None, dx=1, axis=-1, even='avg')
```

kjer so parametri:

- `y` tabela funkcijskih vrednosti, ki jih integriramo,
- `x` vozlišča, gre za opcijski parameter, če je `x=None`, se predpostavi ekvidistantne podintervale širine `dx`,
- `dx` širina ekvidistantnih podintervalov oz korak integriranja,
- `axis` os integriranja (pomembno v primeru večdimenzijskega numeričnega polja)
- `even` je lahko {`'avg'`, `'first'`, `'last'`} in definira način integriranja v primeru lihega števila podintervalov.

Poglejmo si primer, najprej uvozimo funkcijo `simps`:

```
In [43]: from scipy.integrate import simps

In [44]: #%%timeit
         simps(y5v, dx=h5v)

Out[44]:
```

1.44044796026

11.2.5 Rombergova metoda*

Rombergova metoda temelji na Richardsovi ekstrapolaciji. Predpostavimo, da integral $\int_a^b f(x)\mathrm{d}x$ integriramo na intervalu $[a, b]$, ki ga razdelimo na $n = 1, 2, 4, 8, \ldots$ podintervalov.

Rezultat trapeznega pravila pri $n = 1$ označimo z $R\underbrace{1}_{n}\underbrace{1}_{j}$, pri čemer j označuje natančnost pridobljenega rezultata $\mathcal{O}(h^{2j})$.

Če uporabimo trapezno integracijsko pravilo pri $n = 1, 2, 4 \ldots$ podintervalih, izračunamo:

- $R_{1,1}$, korak $h_1 = h$, red natančnosti $\mathcal{O}(h_1^2)$,
- $R_{2,1}$, korak $h_2 = h/2$, red natančnosti $\mathcal{O}(h_2^2)$,
- $R_{3,1}$, korak $h_3 = h/4$, red natančnosti $\mathcal{O}(h_3^2)$,
- $R_{4,1}$, korak $h_4 = h/8$, red natančnosti $\mathcal{O}(h_4^2)$,
- $\ldots$
- $R_{n,1}$, korak $h_4 = h/(2^{n-1})$, red natančnosti $\mathcal{O}(h_n^2)$.

Na podlagi Richardsove ekstrapolacije izračunamo boljši približek

- $R_{2,2} = R_{2,1} + \frac{1}{3}(R_{2,1} - R_{1,1})$, korak $h_2 = h/2$, red natančnosti $\mathcal{O}(h_2^4)$,
- $R_{3,2} = R_{3,1} + \frac{1}{3}(R_{3,1} - R_{2,1})$, korak $h_3 = h/4$, red natančnosti $\mathcal{O}(h_3^4)$,
- $\ldots$
- $R_{n,2} = R_{n,1} + \frac{1}{3}(R_{n,1} - R_{n-1,1})$, korak $h_n = h/(2^{n-1})$, red natančnosti $\mathcal{O}(h_n^4)$.

Nato nadaljujemo z Richardsovo ekstrapolacijo:

- $R_{3,3} = R_{3,2} + \frac{1}{15}(R_{3,2} - R_{2,2})$, korak $h_3 = h/4$, red natančnosti $\mathcal{O}(h_3^6)$,

- ...
- $R_{n,3} = R_{n,2} + \frac{1}{15}\left(R_{n,2} - R_{n-1,2}\right)$, korak $h_n = h/(2^{n-1})$, red natančnosti $\mathcal{O}(h_n^6)$.

Richardsovo extrapolacijo lahko posplošimo:

- $R_{n,j} = R_{n,j-1} + \frac{1}{4^{j-1}-1}\left(R_{n,j-1} - R_{n-1,j-1}\right)$, korak $h_n = h/(2^{n-1})$, red natančnosti $\mathcal{O}(h_n^{2j})$

Pri tem je boljši približek $R_{2,2}$ pri koraku $h/2$ enak rezultatu, ki ga dobimo po Simpsonovi metodi pri koraku $h/2$. Podobno je boljši približek $R_{3,2}$ pri koraku $h/4$ enak numeričnemu integralu Simpsonove metode pri koraku $h/4$. In je dalje $R_{3,3}$ enak popravljenemu približku Simpsonove metode pri koraku $h/4$.

Pripravimo numerične podatke:

```
In [45]: x9v, h9v = np.linspace(1, 2, 9, retstep=True)  # korak h9v = 0.125 (9 vozlišč)
         y9v = x9v * np.sin(x9v)
```

Poglejmo si primer od zgoraj. Najprej s sestavljeno trapezno metodo izračunamo integral pri različnih korakih (drugi red napake):

```
In [46]: # O(h^2)
         R1_1 = trapezno_sest(y9v[::8], 8*h9v) # h=1.0
         R2_1 = trapezno_sest(y9v[::4], 4*h9v) # h=0.5
         R3_1 = trapezno_sest(y9v[::2], 2*h9v) # h=0.25
         R4_1 = trapezno_sest(y9v, h9v) # h=0.125
         [R1_1, R2_1, R3_1, R4_1]
```

```
Out[46]:
```

$$[1.33003291923,\quad 1.41313769957,\quad 1.43362039509,\quad 1.43872310575]$$

Nato izračunamo boljše približke (dobimo četrti red napake):

```
In [47]: # O(h^4)
         R2_2 = R2_1 + 1/3 * (R2_1 - R1_1)
         R3_2 = R3_1 + 1/3 * (R3_1 - R2_1)
         R4_2 = R4_1 + 1/3 * (R4_1 - R3_1)
         [R2_2, R3_2, R4_2]
```

```
Out[47]:
```

$$[1.44083929301,\quad 1.44044796026,\quad 1.44042400931]$$

Rezultati predstavljajo rezultat Simpsonove metode pri koraku $h = 0,5$, $h = 0,25$ in $h = 0.125$:

```
In [48]: [simpsonovo_sest(y9v[::4], 4*h9v), simpsonovo_sest(y9v[::2], 2*h9v), simpsonovo_sest(y9v, h9v)]
```

```
Out[48]:
```

$$[1.44083929301,\quad 1.44044796026,\quad 1.44042400931]$$

Ponovno izračunamo boljše približke (dobimo šesti red napake):

```
In [49]: # O(h6)
         R3_3 = R3_2 + 1/15 * (R3_2 - R2_2)
         R4_3 = R4_2 + 1/15 * (R4_2 - R3_2)

         [R3_3, R4_3]
```

Out[49]:

$$[1.44042187141, \quad 1.44042241258]$$

Rezultat predstavlja boljši rezultat Simpsonove pri koraku $h = 0,25$ in $h = 0.125$:

```
In [50]: a4h, a2h, ah = [simpsonovo_sest(y9v[::4], 4*h9v), simpsonovo_sest(y9v[::2], 2*h9v), simpsonovo
         [16/15*a2h-1/15*a4h, 16/15*ah-1/15*a2h]
```

Out[50]:

$$[1.44042187141, \quad 1.44042241258]$$

Ponovno izračunamo boljše približke (dobimo osmi red napake):

```
In [51]: # O(h8)
         R4_4 = R4_3 + 1/63 * (R4_3 - R3_3)
         R4_4
```

Out[51]:

$$1.44042242117$$

Rombergova metoda nam torej ponuja visoko natančnost rezultata za relativno majhno numerično ceno. Napako pa ocenimo:

$$E = |R_{n,n} - R_{n-1,n-1}| .$$

Rombergova metoda v `scipy.integrate.romb`

V `scipy` je implementirana Rombergova metoda v `scipy.integrate.romb()` (dokumentacija[5]):

```
romb(y, dx=1.0, axis=-1, show=False)
```

kjer so parametri:

- `y` tabela funkcijskih vrednosti, ki jih integriramo,
- `dx` širina ekvidistantnih podintervalov oz korak integriranja,
- `axis` or integriranja (pomembno v primeru večdimenzijskega numeričnega polja),
- `show` če je `True` prikaže elemente Richardsove ekstrapolacije.

Poglejmo si primer od zgoraj:

[5] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.romb.html#scipy.integrate.romb

```
In [52]: from scipy.integrate import romb

In [53]: romb(y9v, dx=h9v, show=True)


      Richardson Extrapolation Table for Romberg Integration
==================================================================== 1.33003
 1.41314  1.44084
 1.43362  1.44045  1.44042
 1.43872  1.44042  1.44042  1.44042
====================================================================


Out[53]:
```

1.44042242117

11.3 Gaussov integracijski pristop

Newton-Cotesov pristop temelji na polinomu n-te stopnje in napaka je $n+1$ stopnje. To pomeni, da integracija daje točen rezultat, če je integrirana funkcija polinom stopnje n ali manj.

Gaussov integracijski pristop je v principu drugačen: cilj je integral funkcije $f(x)$ nadomestiti z uteženo vsoto vrednosti funkcije pri diskretnih vrednostih $f(x_i)$:

$$\int_a^b f(x)\,\mathrm{d}x \approx \sum_{i=0}^{n-1} w_i\, f(x_i).$$

Pri tem je neznana utež w_i in tudi lega vozlišča x_i. Za stopnje polinoma n bomo potrebovali tudi več točk $(x_i, f(x_i))$.

V nadaljevanju bomo spoznali, da lahko zelo učinkovito izračunamo numerično točen integral. Prednost Gaussove integracije je tudi, da lahko izračuna integral funkcij s singularnostmi (npr: $\int_0^1 \sin(x)/\sqrt{(x)}\,dx$).

11.3.1 Gaussova kvadratura z enim vozliščem

Predpostavimo, da želimo integrirati polinom stopnje $n = 1$ (linearna funkcija):

$$f(x) = P_1(x) = A_0 + A_1\, x.$$

Izračunajmo:

$$\int_a^b P_1(x)\,\mathrm{d}x = -A_0\, a + A_0\, b - \frac{A_1\, a^2}{2} + \frac{A_1\, b^2}{2}.$$

Po drugi strani pa želimo integral izračunati glede na ustrezno uteženo w_0 vrednost funkcije $f(x_0)$ v neznanem vozlišču x_0 (samo eno vozlišče!):

$$\int_a^b P_1(x)\,\mathrm{d}x = w_0\,P_1(x_0) = w_0\,A_0 + w_0\,A_1\,x.$$

A_0 in A_1 sta koeficienta linearne funkcije, ki lahko zavzame poljubne vrednosti. Z enačenjem zgornjih izrazov izpeljemo:

$$-A_0\,a + A_0\,b - \frac{A_1\,a^2}{2} + \frac{A_1\,b^2}{2} = w_0\,A_0 + w_0\,A_1\,x$$

Gre za sistem linearnih enačb z rešitvijo:

$$w_0 = b - a, \qquad x_0 = \frac{b+a}{2}.$$

Če je integrirana funkcija linearna, bomo samo na podlagi vrednosti v eni točki izračunali pravo vrednost! Da je Gaussov integracijski pristop neodvisen od mej integriranja a, b, pa uvedemo standardne meje.

Standardne meje: $a = -1$ **in** $b = 1$

Zaradi splošnosti meje $x \in [a, b]$ transformiramo v $\xi \in [-1, +1]$ s pomočjo:

$$x = \frac{b+a}{2} + \frac{b-a}{2}\xi$$

in

$$\mathrm{d}x = \frac{b-a}{2}\mathrm{d}\xi.$$

Velja:

$$\int_a^b f(x)\,\mathrm{d}x = \int_{-1}^{1} g\,(\xi)\;\mathrm{d}\xi,$$

kjer je:

$$g(\xi) = \frac{b-a}{2}\,f\left(\frac{b+a}{2} + \frac{b-a}{2}\xi\right).$$

V primeru standardnih mej je pri eni Gaussovi točki utež $w_0 = 2$ in $x_0 = 0$ vrednost, pri kateri moramo izračunati funkcijo $f(x_0)$.

Strojno izpeljevanje uteži in Gaussove točke

Definirajmo simbole in nastavimo enačbo:

```
In [54]: A0, A1, x, a, b, w0, x0 = sym.symbols('A0, A1, x, a, b, w0, x0') # simboli
         P1 = A0 + A1*x                                                   # linearni polinom
         eq = sym.Eq(P1.integrate((x, a, b)).expand(), w0*P1.subs(x, x0)) # teoretični integral = ocen
         eq
```

Out[54]:

$$-A_0 a + A_0 b - \frac{A_1 a^2}{2} + \frac{A_1 b^2}{2} = w_0 \left(A_0 + A_1 x_0\right)$$

Pripravimo dve enačbi (za prvo predpostavimo $A_0 = 0, A_1 = 1$, za drugo predpostavimo $A_0 = 1, A_1 = 0$) ter rešimo sistem za `w0` in `x0`:

```
In [55]: sym.solve([eq.subs(A0, 0).subs(A1, 1), eq.subs(A1, 0).subs(A0, 1)], [w0, x0])
```

Out[55]:

$$\left[\left(-a + b, \quad \frac{1}{2}\left(a + b\right)\right)\right]$$

Za dodatno razlago priporočam video posnetek[6].

11.3.2 Gaussova integracijska metoda z več vozlišči

Izpeljati želimo Gaussovo integracijsko metodo, ki bo upoštevala na intervalu $[a, b]$ v vozlišč in bo točno izračunala integral polinomov do stopnje $n = 2v - 1$. Veljati mora:

$$\int_a^b P_{2v-1}(x)\,\mathrm{d}x = \sum_{i=0}^{v-1} w_i\, P_{2v-1}(x_i).$$

Pri izpeljavi se bomo omejili na standardne meje $a = -1$, $b = 1$,

kjer je polinom stopnje $n = 2v - 1$ definiran kot:

$$P_{2v-1}(x) = \sum_{i=0}^{2v-1} A_i\, x^i.$$

Z dvema Gaussovima točkama/vozliščema točno izračunamo integral polinoma do tretjega reda, s tremi Gausovimi vozlišči pa točno izračunamo integral polinoma do petega reda!

Strojno izpeljevanje

Pripravimo si najprej simbolni zapis polinoma in ustreznih spremenljivk:

```
In [56]: def P_etc(n=1, A='A', x='x'):                          # n je stopnja polinoma
             Ai = sym.symbols('{0:s}:{1:g}'.format(A, n+1))     # seznam A_i
             x = sym.symbols(x)                                 # spremenljivka x
             return Ai, x, sum([Ai[i]*x**i for i in range(n+1)])
```

[6] https://www.youtube.com/watch?v=iQ5-4hx25Rw

Sedaj pa poiščimo uteži w_i in vozlišča x_i za primer dveh Gaussovih vozlišč; polinom je torej tretje stopnje.

```
In [57]: v = 2 # število vozlišč
         Ai, x, P = P_etc(n=2*v-1)
         xi = sym.symbols('x:{0:g}'.format(v)) # seznam x_i
         wi = sym.symbols('w:{0:g}'.format(v)) # seznam w_i
         print(f'Vozlišča: {xi}\nUteži:    {wi}')

Vozlišča: (x0, x1)
Uteži:    (w0, w1)
```

Polinom:

```
In [58]: P
```

Out[58]:

$$A_0 + A_1 x + A_2 x^2 + A_3 x^3$$

Podobno kakor zgoraj za eno vozlišče, tukaj definirajmo enačbe:

```
In [59]: eqs = [sym.Eq(P.integrate((x, -1, 1)).coeff(A_),\
                       sum([wi[i]*P.subs(x, xi[i]) \
                            for i in range(v)]).expand().coeff(A_)) \
                for A_ in Ai]
         eqs
```

Out[59]:

$$\left[2 = w_0 + w_1, \quad 0 = w_0 x_0 + w_1 x_1, \quad \frac{2}{3} = w_0 x_0^2 + w_1 x_1^2, \quad 0 = w_0 x_0^3 + w_1 x_1^3\right]$$

Rešimo jih za za neznane x_i in w_i:

```
In [60]: sol = sym.solve(eqs, sym.flatten((xi, wi)))
         sol
```

Out[60]:

$$\left[\left(-\frac{\sqrt{3}}{3}, \quad \frac{\sqrt{3}}{3}, \quad 1, \quad 1\right), \quad \left(\frac{\sqrt{3}}{3}, \quad -\frac{\sqrt{3}}{3}, \quad 1, \quad 1\right)\right]$$

Določili smo seznam dveh (enakih) rešitev.

Najprej sta definirani vozlišči: $x_0 = -\sqrt{3}/3$ in $x_1 = \sqrt{3}/3$, katerima pripadata uteži $w_0 = w_1 = 1$.

Koda zgoraj je izpeljana v splošnem - število vozlišč `v` lahko povečate ter izračunate vozlišča ter pripadajoče uteži.

Tukaj je podana tabela vozlišč in uteži za eno, dve in tri vozlišča (za meje $a = -1$, $b = 1$):

Število točk	Vozlišče x_i	Utež w_i
1	0	2
2	$-\frac{\sqrt{3}}{3}$	1
	$+\frac{\sqrt{3}}{3}$	1
3	$-\frac{\sqrt{15}}{5}$	$\frac{5}{9}$
	0	$\frac{8}{9}$
	$\frac{\sqrt{15}}{5}$	$\frac{5}{9}$

Za več vozlišč in tudi oceno napake, glejte Mathworld Legendre-Gauss Quadrature[7].
Za primer treh Gaussovih točk numerični integral izračunamo (standardne meje):

$$I_{\text{Gauss3}} = \frac{5}{9} f\left(-\frac{\sqrt{15}}{5}\right) + \frac{8}{9} f\left(0\right) + \frac{5}{9} f\left(\frac{\sqrt{15}}{5}\right).$$

Numerična implementacija

Numerična implementacija (vključno s transformacijo mej) za eno, dve ali tri vozlišča:

```
In [61]: def Gaussova(fun, a, b, vozlišč=1):
             """
             Gaussova integracijska metoda.

             :param fun: objekt funkcije, ki jo integriramo
             :param a:   spodnja meja
             :param b:   zgornja meja
             :param vozlišča: število vozlišč (1, 2 ali 3)
             """
             def g(xi): # funkcija za transformacijo mej
                 return (b-a)/2 * fun((b+a +xi * (b-a))/2)

             if vozlišč == 1:
                 return 2*g(0.)
             elif vozlišč == 2:
                 return 1. * g(-np.sqrt(3)/3) + 1. * g(np.sqrt(3)/3)
             elif vozlišč == 3:
                 return 5/9 * g(-np.sqrt(15)/5) +8/9 * g(0.) + 5/9 * g(np.sqrt(15)/5)
```

Poglejmo si zgled. Najprej definirajmo funkcijo, ki jo želimo integrirati:

```
In [62]: def f(x):
             return x*np.sin(x)
```

Sedaj pa funkcijo (v konkretnem primeru `f`) in ne vrednosti (npr. `f(0.)`) posredujemo funkciji `Gaussova`. Najprej za eno vozlišče, nato dve in tri:

[7] http://mathworld.wolfram.com/Legendre-GaussQuadrature.html

```
In [63]: Gaussova(fun=f, a=1., b=2., vozlišč=1)
```

Out[63]:

1.49624247991

```
In [64]: Gaussova(fun=f, a=1., b=2., vozlišč=2)
```

Out[64]:

1.44014401845

```
In [65]: Gaussova(fun=f, a=1., b=2., vozlišč=3)
```

Out[65]:

1.44042294912

`scipy.integrate.fixed_quad`

Znotraj `scipy` je Gaussova integracijska metoda implementirana v okviru funkcije `scipy.integrate.fixed_quad()` (dokumentacija[8]):

```
fixed_quad(func, a, b, args=(), n=5)
```

kjer so parametri:

- `func` je ime funkcije, ki jo kličemo,
- `a` je spodnja meja,
- `b` je zgornja meja,
- `args` je terka morebitnih dodatnih argumentov funkcije `func`,
- `n` je število vozlišč Gaussove integracije, privzeto `n=5`.

Funkcija vrne terko z rezultatom integriranja `val` in vrednost `None`: `(val, None)`

Poglejmo primer od zgoraj:

```
In [66]: from scipy.integrate import fixed_quad
         fixed_quad(f, a=1, b=2, n=3)[0]
```

Out[66]:

1.44042294912

Rezultat je enak predhodnemu:

```
In [67]: Gaussova(fun=f, a=1., b=2., vozlišč=3)
```

Out[67]:

1.44042294912

[8] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.fixed_quad.html

11.4 scipy.integrate

scipy.integrate je močno orodje za numerično integriranje (glejte dokumentacijo[9]). V nadaljevanju si bomo pogledali nekatere funkcije.

11.4.1 Intergracijske funkcije, ki zahtevajo definicijsko *funkcijo*:

- quad(func, a, b[, args, full_output, ...]) izračuna določeni integral func(x) v mejah [a, b],
- dblquad(func, a, b, gfun, hfun[, args, ...]) izračuna določeni integral func(x,y),
- tplquad(func, a, b, gfun, hfun, qfun, rfun) izračuna določeni integral func(x,y,z),
- nquad(func, ranges[, args, opts, full_output]) izračuna določeni integral n dimenzijske funkcije func(...),
- romberg(function, a, b[, args, tol, rtol, ...]) integracija Romberg za funkcijo function,

scipy.integrate.quad

Poglejmo si zelo uporabno funkcijo za integriranje, scipy.integrate.quad() (dokumentacija[10]):

```
quad(func, a, b, args=(), full_output=0, epsabs=1.49e-08, epsrel=1.49e-08, limit=50,
     points=None, weight=None, wvar=None, wopts=None, maxp1=50, limlst=50)
```

Izbrani parametri so:

- func Python funkcija, ki jo integriramo,
- a spodnja meja integriranja (lahko se uporabi np.inf za mejo v neskončnosti),
- b zgornja meja integriranja (lahko se uporabi np.inf za mejo v neskončnosti),
- full_output za prikaz vseh rezultatov, privzeto 0,
- epsabs dovoljena absolutna napaka,
- epsrel dovoljena relativna napaka.

Poglejmo primer od zgoraj:

```
In [68]: from scipy import integrate

In [69]: integrate.quad(f, a=1, b=2)

Out[69]:
```

$$(1.4404224209802095, \quad 1.5991901369585387e-14)$$

scipy.integrate.dblquad

Gre za podobno funkcijo kot quad, vendar za integriranje po dveh spremenljivkah (dokumentacija[11]). Funkcija scipy.integrate.dblquad() izračuna dvojni integral func(y, x) v mejah od x = [a, b] in y = [gfun(x), hfun(x)].

```
dblquad(func, a, b, gfun, hfun, args=(), epsabs=1.49e-08, epsrel=1.49e-08)
```

[9] https://docs.scipy.org/doc/scipy/reference/integrate.html
[10] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.quad.html#scipy.integrate.quad
[11] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.dblquad.html#scipy.integrate.dblquad

Izbrani parametri so:

- `func` je Python funkcija, ki jo integriramo,
- `a` je spodnja meja integriranja `x` (lahko se uporabi `np.inf` za mejo v neskončnosti),
- `b` je zgornja meja integriranja `x` (lahko se uporabi `np.inf` za mejo v neskončnosti),
- `gfun` je Python funkcija, ki definira spodnjo mejo `y` v odvisnosti od `x`,
- `hfun` je Python funkcija, ki definira zgornjo mejo `y` v odvisnosti od `x`.

Poglejmo primer izračuna površine polkroga s polmerom 1:

$$\int_{-1}^{1} \left(\int_{0}^{\sqrt{1-x^2}} 1 \, \mathrm{d}y \right) \mathrm{d}x = \frac{\pi}{2}$$

Definirajmo ustrezne Python funkcije in izračunajmo rezultat:

```
In [70]: def func(y, x): # integracijska funkcija je enostavna, konstanta = 1!
             return 1.
         def gfun(x):    # spodnja meja = 0
             return 0.
         def hfun(x):    # zgornja meja
             return np.sqrt(1-x**2)
         integrate.dblquad(func=func, a=-1, b=1, gfun=gfun, hfun=hfun)
```

Out[70]:

$$(1.5707963267948986, \quad 1.0002356720661965e-09)$$

11.4.2 Intergracijske funkcije, ki zahtevajo tabelo vrednosti:

- `trapz(y[, x, dx, axis])` sestavljeno trapezno pravilo,
- `cumtrapz(y[, x, dx, axis, initial])` kumulativni integral podintegralske funkcije (vrne rezultat v vsakem vozlišču),
- `simps(y[, x, dx, axis, even])` Simpsonova metoda,
- `romb(y[, dx, axis, show])` Rombergova metoda.

Tukaj si bomo na primeru ogledali funkcijo `cumtrapz`. Kot primer si, ko na maso $m = 1$ (enote izpustimo) deluje sila $F = \sin(t)$. Iz 2. Newtonovega zakona sledi: $F = m\ddot{x}$. Pospešek integriramo, da izračunamo hitrost; nato integriramo še enkrat za pomik. Definirajmo najprej funkcijo za pospešek:

```
In [71]: def pospešek(t):
             F = np.sin(t)
             m = 1
             return F/m
```

Zanima nas dogajanje v času 3 sekund:

```
In [72]: t, h = np.linspace(0, 3, 100, retstep=True)
```

Izračunajmo tabelo pospeškov ter nato integrirajmo za hitrost (pri tem je pomembno, da definiramo začetno vrednost):

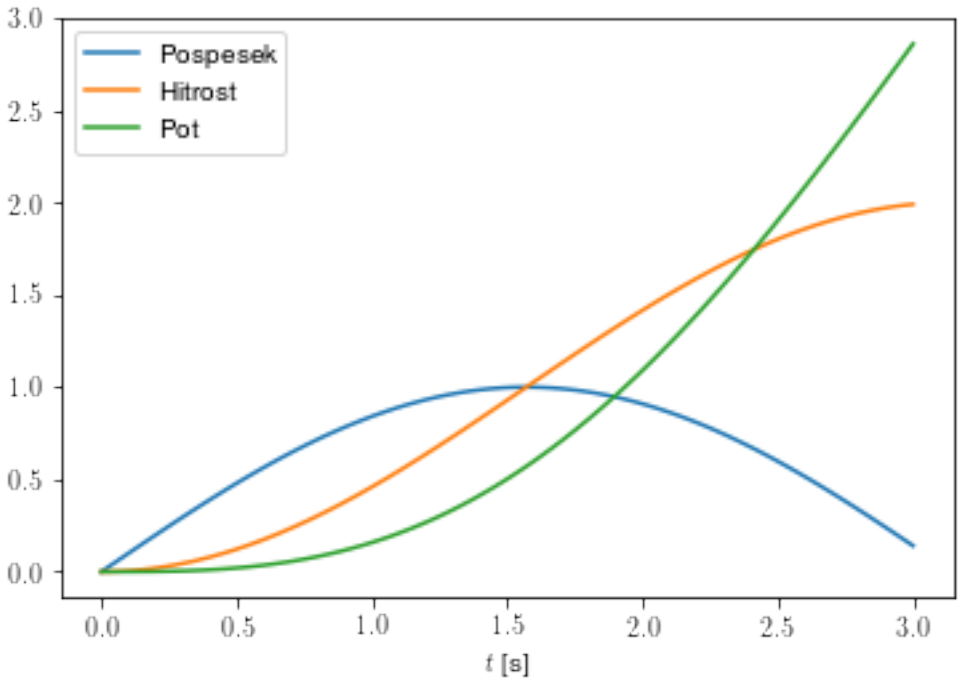
```
In [73]: a = pospešek(t)
         v = integrate.cumtrapz(y=a, dx=h, initial=0)
```

Hitrost sedaj še enkrat integrirajmo, da izračunamo pot:

```
In [74]: s = integrate.cumtrapz(y=v, dx=h, initial=0)
```

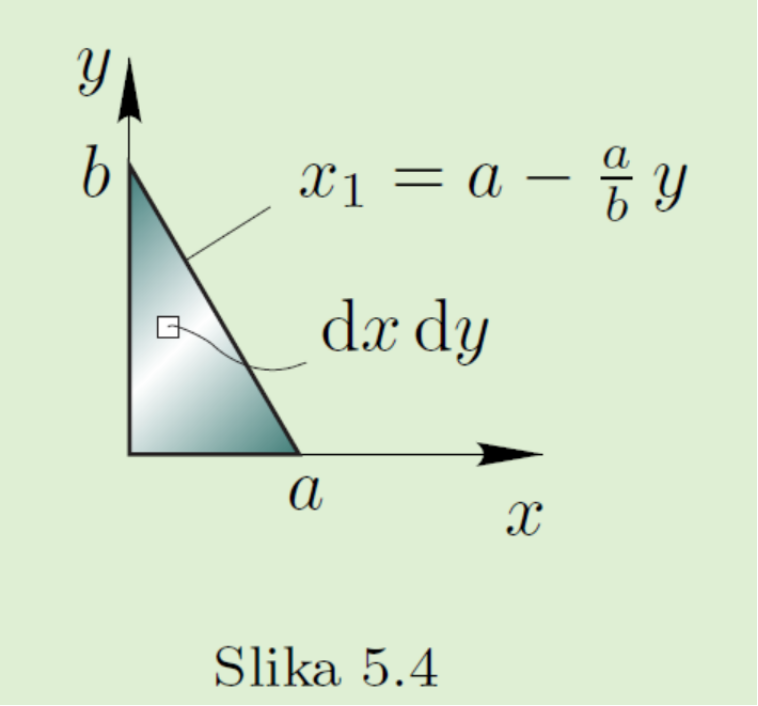
Prikažimo rezultat:

```
In [75]: plt.plot(t, a, label='Pospesek')
         plt.plot(t, v, label='Hitrost')
         plt.plot(t, s, label='Pot')
         plt.xlabel('$t$ [s]')
         plt.legend();
```



11.5 Nekaj vprašanj za razmislek!

1. Na sliki (vir: J. Slavič: Dinamika, meh. nihanja ..., 2014) je prikazan trikotnik s stranicami dolžine a, b, z debelino h in gostoto ρ.



Slika 5.4

V simbolni obliki določite masni vztrajnostni moment glede na prikazano os y:

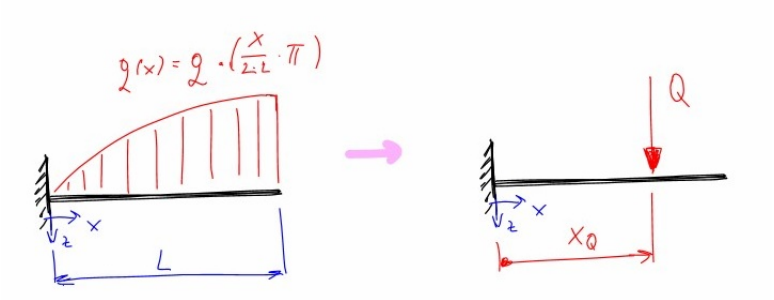
$$J_{yy} = \int_0^b y^2\,\rho\,h\,(a - a/b\,y)\,\mathrm{d}y.$$

Upoštevajte tudi: $m = a\,b\,h\,\rho/2$. Za izmišljene vrednosti izračunajte numerični rezultat.

2. Izračunajte integral tudi numerično. Uporabite `scipy.integrate` in integrirajte glede na pravila: trapezno, Simpsonovo 1/3. Rezultat primerjajte tudi z Gaussovo kvadraturo. Raziščite natančnost in hitrost metod.
3. Preštudirajte `scipy.special.legendre`, ki vam vrne objekt `orthopoly1d`. Ta objekt ima metodo `weights`, ki vrne seznam `[x, w, mu0]` vrednosti, ki jih uporabimo pri Gaussovi kvadraturi. (Če vsega ne razumete, ne skrbite preveč, bo asistent pokazal/komentiral). Opazite lahko, da smo vrednosti izpeljali na predavanjih!
4. S pomočjo zgoraj pridobljenih uteži in vozlišč izračunajte integral s pomočjo Gaussove kvadrature: $\sum_i w_i\,f(x_i)$. Pazite na transformacijo mej.
5. Preprost integral $\int_0^2 x^2\,dx$ izrabite za prikaz trapeznega in Simpsonovega 1/3 pravila (osnovno pravilo, ne sestavljeno). Uteži izračunajte z uporabo `scipy`.
6. Integral predhodne točke razširite za sestavljeno trapezno pravilo (lastna koda). Prikažite vpliv števila podintervalov, primerjajte napako izračuna s predhodnim številom podintervalov in prikažite konvergenco.
7. Integral predhodne točke razširite za sestavljeno Simpsonovo 1/3 pravilo (lastna koda). Prikažite vpliv števila podintervalov, primerjajte napako izračuna s predhodnim številom podintervalov in prikažite konvergenco.
8. Z različnimi metodami izračunajte integrala (namig: vse metode niso primerne):

$$\int_1^2 \frac{\sin(x)}{\sqrt{x}}.$$

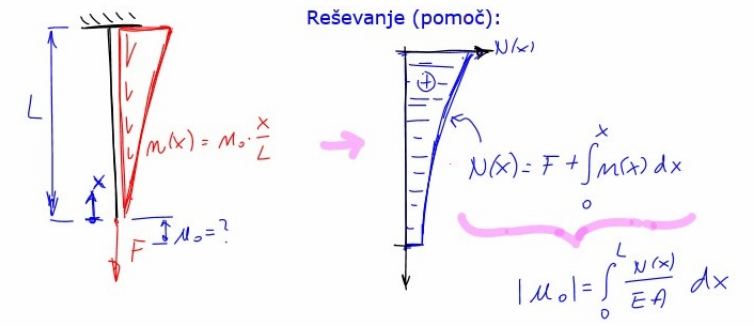
9. S pomočjo numeričnega integriranja določite ekvivalentno silo porazdeljene obremenitve (Q) ter njeno prijemališče vzdolž nosilca (x_Q) dolžine $L = 2\,\mathrm{m}$. Konstanta obremenitve: $q_0 = 5\,\mathrm{kN/m}$.



Rešitev: ekvivalentna obremenitev $Q = \int_0^L q(x)\mathrm{d}x$, pozicija (težišče) $x_Q = \frac{\int_0^L x\, q(x)\mathrm{d}x}{\int_0^L q(x)\mathrm{d}x}$

11.6 Dodatno

1. Obravnavajte prikazan enoosni primer, obremenjen s porazdeljeno obremenitvijo $n(x)$ ter točkovno silo $F = 10\,\mathrm{kN}$. Dolžina palice je $L = 2\,\mathrm{m}$, konstanta $n_0 = 15\,\mathrm{kN/m}$ in $EA = 200000\,\mathrm{MPa} \times 50 \times 50\,\mathrm{mm}^2$.



Naloga:

1. S pomočjo simbolnega integriranja določite funkcijo notranje osne sile $N(x)$.
2. S pomočjo numeričnega integriranja izračunajte pomik prostega konca palice u_0.

Poglavje 12

Numerično reševanje diferencialnih enačb - začetni problem

12.1 Uvod

12.1.1 Zapis (ene) diferencialne enačbe

Predpostavimo, da je mogoče diferencialno enačbo prvega reda zapisati v *eksplicitni* obliki:

$$y' = f(t, y),$$

kjer je $f(t, y)$ podana funkcija in velja $y' = dy/dx$.

Dodatno je podan začetni pogoj:

$$y(t_0) = y_0.$$

Cilj reševanja diferencialne enačbe je izračunati funkcijo $y(t)$, ki reši zgoraj definiran začetni problem. Ob določenih pogojih funkcije $f(t, y)$ ima začetni problem enolično rešitev na intevalu, ki vsebuje a.

Pri numeričnem reševanju vedno računamo tabelo funkcije $y(t_i)$, ki reši dan začetni problem. Pri tem so vozlišča t_i običajno ekvidistantna:

$$t_0, t_0 + h, t_0 + 2h, \ldots$$

in h imenujemo (časovni) korak (integracije).

Tukaj si bomo pogledali nekatere numerične metode za reševanje diferencialnih enačb pri začetnem pogoju.

12.2 Eulerjeva metoda

Eksplicitna Eulerjeva metoda temelji na razvoju funkcije y v Taylorjevo vrsto:

$$y(t + h) = y(t) + y'(t, y(t))\, h + \mathcal{O}(h^2).$$

Naredimo napako metode $\mathcal{O}(h^2)$, ker zanemarimo odvode drugega in višjih redov; sedaj lahko ob znani vrednosti $y(t)$ in odvodu $y'(t) = f(t, y)$ ocenimo vrednosti pri naslednjem časovnem koraku $t + h$. Ko imamo enkrat znane vrednosti pri $t + h$, ponovimo postopek!

Koraki Eulerjeve metode:

1. Postavimo $i = 0$, t_0, $y_0 = y(t_0)$.
2. Izračun vrednosti funkcije pri $t_{i+1} = t_i + h$:

$$y_{i+1} = y_i + f(t_i, y_i))h.$$

3. $i = i + 1$ in nadaljevanje v koraku 2.

Diferencialno enačbo rešujemo na intervalu $[a, b]$ in velja $h = (b - a)/n$. n je število integracijskih korakov (kolikokrat izvedemo korak 2 v zgornjem algoritmu).

Numerična rešitev začetnega problema:

$$y_0, y_1, y_2, \ldots, y_n$$

pri vrednostih neodvisne spremenljivke:

$$t_0, t_1, t_2 \ \ldots \ t_n.$$

12.2.1 Napaka Eulerjeve metode

Napaka Eulerjeve metode na vsakem koraku je reda $\mathcal{O}(h^2)$.

Ker na intervalu od t_0 do t_n tako napako naredimo n-krat, je kumulativna napaka $n\,\mathcal{O}(h^2) = \frac{t_n - t_0}{h}\,\mathcal{O}(h^2) = \mathcal{O}(h)$.

Lokalno je napaka drugega reda, globalno pa je napaka prvega reda in ker je Eulerjeva metoda tako nenatančna jo redko uporabljamo v praksi!

Ocena napake

Točna rešitev $y(t_n)$ pri velikosti koraka h je:

$$y(t_n) = y_{n,h} + E_h,$$

kjer je $y_{n,h}$ numerični približek in E_h napaka metode. Ker je globalna napaka prvega reda, lahko napako zapišemo kot:

$$E_h = k\,h.$$

Podobno lahko za velikost koraka $2h$ zapišemo:

$$y(t_n) = y_{n,2h} + E_{2h},$$

kjer je $y_{n,2h}$ numerični približek in E_{2h} napaka metode:

$$E_{2h} = k\,2\,h.$$

Ob predpostavki, da je konstanta k pri koraku h in koraku $2h$ enaka, lahko določimo oceno napake pri boljšem približku E_h. Očitno velja:

$$y_{n,h} + k\,h = y_{n,2h} + 2\,k\,h$$

nato določimo *oceno napake*:

$$E_h = k\,h = y_{n,h} - y_{n,2h}.$$

12.2.2 Komentar na implicitno Eulerjevo metodo

Pri eksplicitni Eulerjevi metodi računamo rešitev pri t_{i+1} iz izračunane vrednosti pri t_i.

V kolikor bi nastopala neznana vrednost rešitve pri t_{i+1}, to je y_{i+1}, tudi na desni strani, bi govorili o **implicitni Eulerjevi metodi** (ali *povratni Eulerjevi metodi*):

$$y_{i+1} = y_i + f(t_{i+1}, y_{i+1})\, h.$$

Ker se iskana vrednost y_{i+1} nahaja na obeh straneh enačbe, moramo za določitev y_{i+1} rešiti (nelinearno) enačbo. Prednost implicitne Eulerjeve metode je, da je bolj stabilna (npr. v primeru togih sistemov, ki jih bomo spoznali pozneje) kakor eksplicitna oblika, vendar pa je numerično bolj zahtevna (zaradi računanja rešitve enačbe).

12.2.3 Numerična implementacija

Najprej uvozimo potrebne knjižnice:

```
In [2]: import numpy as np
        import matplotlib.pylab as plt
        %matplotlib inline
```

Nato definirajmo Eulerjevo metodo:

```
In [3]: def euler(f, t, y0, *args, **kwargs):
            """
            Eulerjeva metoda za reševanje sistema diferencialnih enačb: y' = f(t, y)

            :param f:  funkcija, ki vrne prvi odvod - f(t, y)
            :param t:  časovni vektor kjer računamo rešitev
            :param y0: začetna vrednosti
            :param args: dodatni argumenti funkcije f (brezimenski)
            :param kwargs: dodatni argumenti funkcije f (poimenovani)
            :return y: vrne np.array ``y`` vrednosti funkcije.
            """
            y = np.zeros_like(t)
            y[0] = y0
            h = t[1]-t[0]
            for i in range(len(t)-1):
                y[i+1] = y[i] + f(t[i], y[i], *args, **kwargs) * h
            return y
```

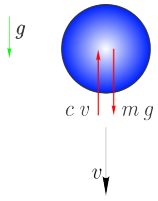
Pripravimo funkcijo za oceno napake (v numeričnem smislu bi bilo bolje oceno napake vključiti v funkcijo `euler`, vendar jo zaradi jasnosti predstavimo ločeno):

```
In [4]: def euler_napaka(f, t, y0, *args, **kwargs):
            """ Ocena napake Eulerjeve metode; argumenti so isti kakor za funkcijo `euler`
            """
            n = len(t)
            if n < 5:
                raise Exception('Vozlišč mora biti vsaj 5.')
```

```
    if n%2==0: # sodo vozlišč; odstrani eno točko in spremeni na liho (da je sodo odsekov)
        n = n - 1
    y_h = euler(f, t[:n], y0, *args, **kwargs)
    y_2h = euler(f, t[:n:2], y0, *args, **kwargs)
    E_h = y_h[-1] - y_2h[-1]
    return E_h
```

12.2.4 Numerični zgled

Kot primer rešimo diferencialno enačbo, ki opisuje padanje telesa, ki je izpostavljeno sili teže in zračnemu uporu:



Glede na II. Newtonov zakon, lahko zapišemo diferencialno enačbo:

$$m\,g - c\,v = m\,v',$$

kjer je m masa, g gravitacijski pospešek, c koeficient zračnega upora in v hitrost. Diferencialno enačbo bi hoteli rešiti glede na začetni pogoj:

$$v(0) = 0\,\mathrm{m/s}.$$

Funkcija desne strani / prvega odvoda $f(t, y)$ je:

$$f(t, y) = g - c\,\frac{v}{m}$$

in začetni pogoj:

$$y_0 = 0.$$

Definirajmo funkcijo desnih strani:

```
In [5]: def f_zračni_upor(t, v, g=9.81, m=1., c=0.5):
            return g-c*v/m
```

Definirajmo začetni pogoj in časovni vektor, kjer nas zanima rezultat:

```
In [6]: v0 = 0
        t = np.linspace(0, 10, 11)
        t
```

```
Out[6]: array([  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
```

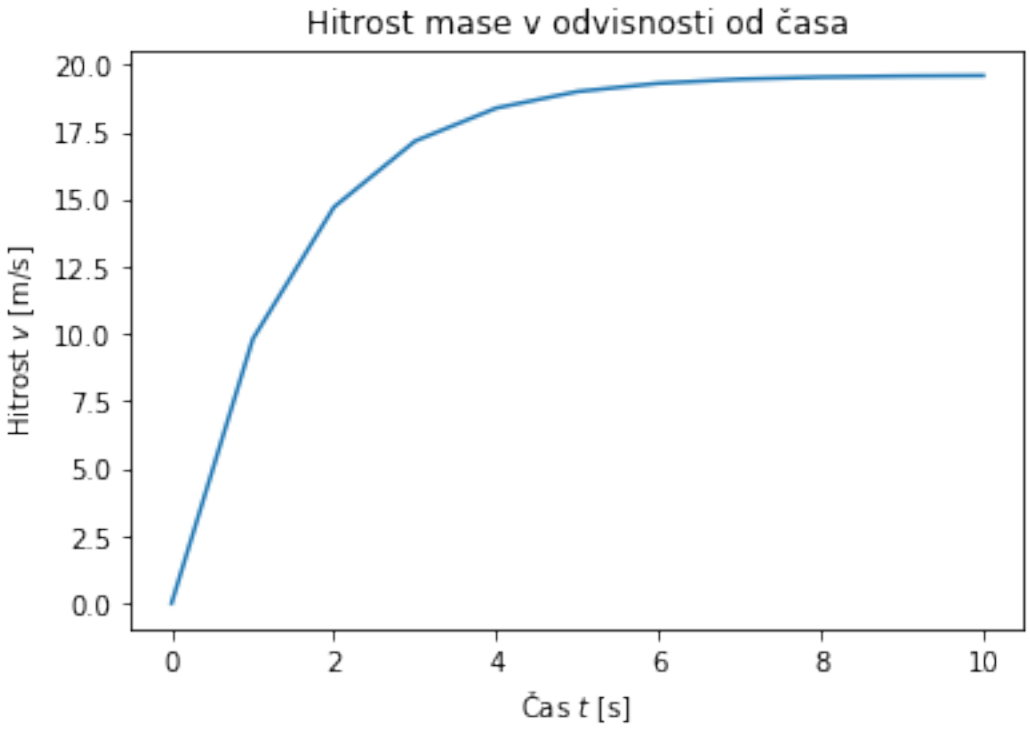
Kličemo funkcijo `euler` za izračun vrednosti y (hitrost v):

```
In [7]: y = euler(f_zračni_upor, t, y0=v0)
        y
```

```
Out[7]: array([  0.        ,   9.81      ,  14.715     ,  17.1675    ,
                18.39375   ,  19.006875  ,  19.3134375 ,  19.46671875,
                19.54335938,  19.58167969,  19.60083984])
```

Prikažemo rezultat:

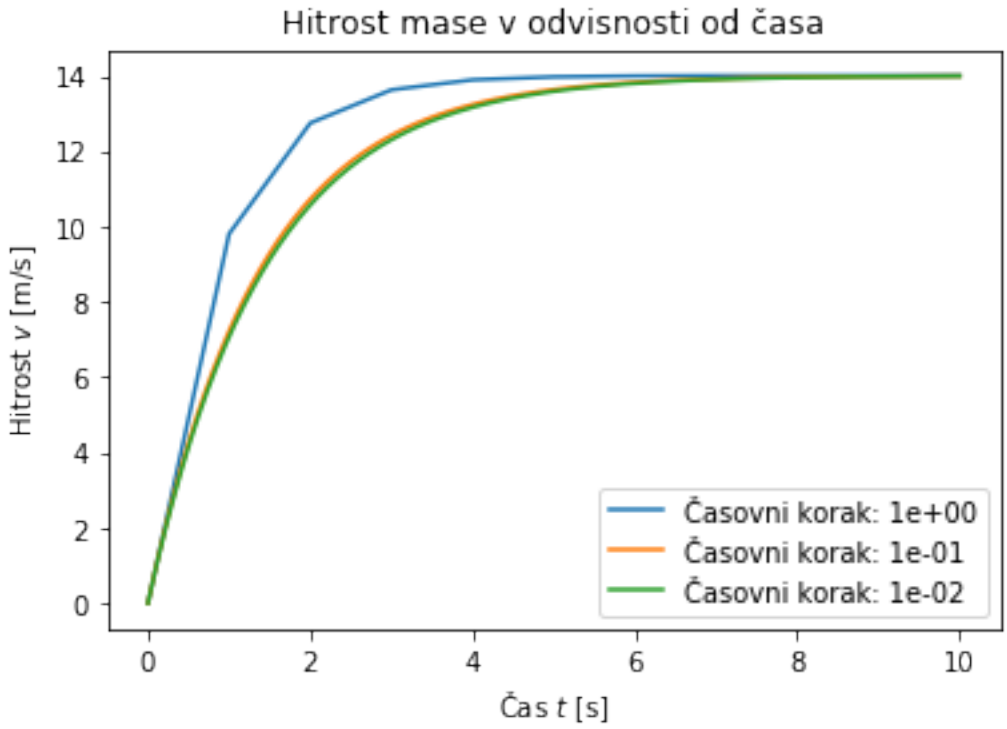
```
In [8]: plt.plot(t, y)
        plt.title('Hitrost mase v odvisnosti od časa')
        plt.xlabel('Čas $t$ [s]')
        plt.ylabel('Hitrost $v$ [m/s]')
        plt.show()
```



Preverimo sedaj vpliv časovnega koraka:

```
In [9]: for n in [11, 101, 1001]:
            t = np.linspace(0, 10, n)
            y = euler(f_zračni_upor, t, y0=v0, c=0.7)
            plt.plot(t, y, label=f'Časovni korak: {t[1]:1.0e}')
        plt.title('Hitrost mase v odvisnosti od časa')
        plt.xlabel('Čas $t$ [s]')
```

```
plt.ylabel('Hitrost $v$ [m/s]')
plt.legend()
plt.show()
```



Opazimo, da se numerična napaka pri spremembi koraka iz 1 na 0,1 bistveno zmanjša!

Ocenimo še napako pri 100 in 1000 odsekih:

```
In [10]: n=101
         t = np.linspace(0, 10, n)
         euler_napaka(f_zračni_upor, t, y0=v0)
```

Out[10]:

$$-0.0150437137353$$

```
In [11]: n=1001
         t = np.linspace(0, 10, n)
         euler_napaka(f_zračni_upor, t, y0=v0)
```

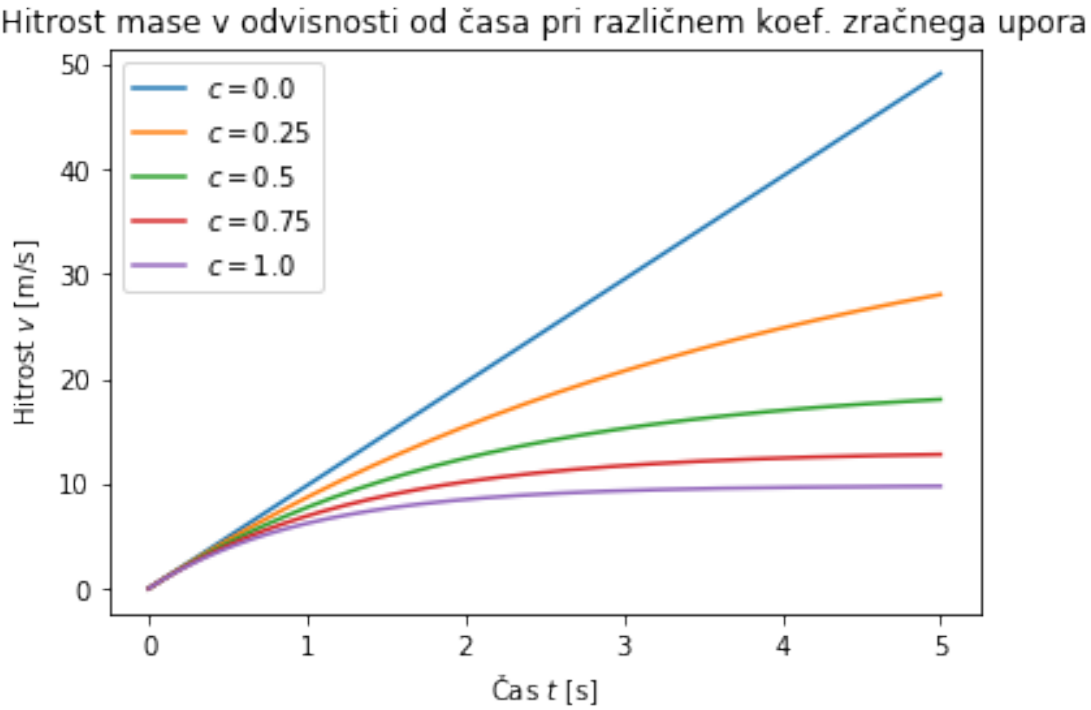
Out[11]:

$$-0.00163798622449$$

Ko smo korak zmanjšali na desetino, se je proporcionalno zmanjšala tudi napaka (prvi red napake).

Poglejmo še primer, ko je zračni upor `c` argument funkcije `euler` in je prek `**kwargs` posredovan v funkcijo `f_zračni_upor()`:

```
In [12]: for c in np.linspace(0, 1, 5):
             t = np.linspace(0, 5, 1001)
             y = euler(f_zračni_upor, t, y0=v0, c=c)
             plt.plot(t, y, label=f'$c={c}$')
         plt.title('Hitrost mase v odvisnosti od časa pri različnem koef. zračnega upora')
         plt.xlabel('Čas $t$ [s]')
         plt.ylabel('Hitrost $v$ [m/s]')
         plt.legend()
         plt.show()
```



12.3 Metoda Runge-Kutta drugega reda

Eulerjeva metoda je prvega reda (prvega reda je namreč globalna napaka $\mathcal{O}(h)$). Če bi želeli izpeljati metodo drugega reda napake, bi si morali pomagati z razvojem $y(t+h)$ v Taylorjevo vrsto, kjer bomo zanemarili tretji in višje odvode:

$$y(t+h) = y(t) + y'(t)\,h + \frac{1}{2}y''(t)\,h^2 + \mathcal{O}(h^3).$$

Lokalna napaka metode bo tako tretjega reda, globalna pa drugega reda.

Uporabimo zamenjavi $y'(t) = f(t,y)$ in $y''(t) = f'(t,y)$:

$$y(t+h) = y(t) + f(t,y)\,h + \frac{1}{2}f'(t,y)\,h^2 + \mathcal{O}(h^3).$$

Ker je desna stran $f(t,y)$ odvisna od neodvisne t in odvisne spremenljivke y, moramo uporabiti pri odvajanju implicitno odvajanje:

$$f'(t,y) = \frac{\partial f(t,y)}{\partial t} + \frac{\partial f(t,y)}{\partial y} \underbrace{\frac{\mathrm{d}y}{\mathrm{d}t}}_{y'=f(t,y)} = \frac{\partial f(t,y)}{\partial t} + \frac{\partial f(t,y)}{\partial y} f(t,y).$$

Vstavimo v izraz za Taylorjevo vrsto:

$$y(t+h)_{\text{Taylor}} = y(t+h) = y(t) + f(t,y)\,h + \frac{1}{2}\,(\frac{\partial f(t,y)}{\partial t} + \frac{\partial f(t,y)}{\partial y}\,f(t,y))\,h^2.$$

Kot je razvidno iz zgornjega izraza, potrebujemo dodatne odvode. To predstavlja določeno težavo, ki se ji lahko izognemo na različne načine; v nadaljevanju si bomo pogledali pristop *Runge-Kutta*. Ker bomo zgornji izraz pozneje še potrebovali, smo ga tukaj poimenovali $y(t+h)_{\text{Taylor}}$.

12.3.1 Ideja pristopa Runge-Kutta

Zgornjo dilemo metoda *Runge-Kutta* (razvita leta 1901) rešuje z idejo, ki smo jo sicer že srečali pri Gaussovi integraciji: točnejšo rešitev poskuša najti s uteženo dodatno vrednostjo funkcije f:

$$y(t+h)_{\text{Runge-Kutta}} = y(t) + c_0\,f\,(t,y)\;h + c_1\,\underbrace{f(t+p\,h, y+q\,h\,f(t,y))}_{A}\;h.$$

kjer so c_0, c_1, p in q neznane konstante (načeloma od 0 do vključno 1). Če bi v zgornjem izrazu uporabili $c_1 = 0$, bi izpeljali metodo prvega reda; z dodatno funkcijsko vrednostjo (A) pa se bo izkazalo, da bomo izpeljali metodo drugega reda.

Iskanje neznanih konstant c_0, c_1, p, q nadaljujemo z zapisom A v obliki Taylorjeve vrste prvega reda:

$$f(t+p\,h, y+q\,h\,f(t,y)) = \underbrace{f(t,y) + \frac{\partial f(t,y)}{\partial t}\;(p\,h) + \frac{\partial f(t,y)}{\partial y}\;(q\,h\,f(t,y))}_{B}.$$

Vstavimo sedaj izpeljani B nazaj izraz za $y(t+h)_{\text{Runge-Kutta}}$:

$$y(t+h)_{\text{Runge-Kutta}} = y(t) + c_0\,f\,(t,y)\;h + c_1(f(t,y) + \frac{\partial f(t,y)}{\partial t}\;(p\,h) + \frac{\partial f(t,y)}{\partial y}\;(q\,h\,f(t,y)))\,h.$$

Nadaljujemo z izpeljevanjem in enačbo preoblikujemo, da bo podobna zgoraj izpeljani s Taylorjevo vrsto $y(t+h)_{\text{Taylor}}$:

$$y(t+h)_{\text{Runge-Kutta}} = y(t) + (c_0+c_1)\,f\,(t,y)\;h + \frac{1}{2}(\frac{\partial f(t,y)}{\partial t}\,2\,c_1\,p + 2\,c_1\,q\,\frac{\partial f(t,y)}{\partial y}\,f(t,y))\,h^2.$$

Primerjajmo sedaj z zgoraj izpeljanim izrazom:

$$y(t+h)_{\text{Taylor}} = y(t) + f(t,y)\,h + \frac{1}{2}\,(\frac{\partial f(t,y)}{\partial t} + \frac{\partial f(t,y)}{\partial y}\,f(t,y))\,h^2.$$

Ugotovimo, da za enakost mora veljati:

$$c_0 + c_1 = 1, \qquad 2\,c_1\,p = 1, \qquad 2\,c_1\,q = 1.$$

Imamo torej tri enačbe in štiri neznanke. Eno od konstant si tako lahko poljubno izberemo, ostale tri pa izračunamo. Če na primer izberemo $c_0 = 0$, bi to imenovali *spremenjena Eulerjeva metoda* in bi ostali parametri bili: $c_1 = 1$, $p = q = 1/2$. Izbira parametrov nima bistvenega vpliva rešitev. Sicer pa velja omeniti, da metodo Runge-Kutta drugega reda redko uporabljamo, saj obstajajo boljše metode.

Parametre c_0, c_1, p in q vstavimo v prvo enačbo tega poglavja. Ko je definiran začetni čas t_0 in začetni pogoj y_0, uporabimo metodo Runge-Kutta drugega reda:

$$y_{i+1} = y_i + f\left(t_i + \frac{1}{2}\,h, y_i + \frac{1}{2}\,h\,f(t_i, y_i)\right)\,h.$$

12.4 Metoda Runge-Kutta četrtega reda

Podobno kot smo izpeljali metodo Runge-Kutta drugea reda, se izpelje metodo Runge Kutta četrtega reda. Tudi pri metodi četrtega reda obstaja več različic in kot metoda Runge-Kutta četrtega reda razumemo naslednjo metodo:

$$y_{i+1} = y_i + \frac{1}{6}(k_0 + 2\,k_1 + 2\,k_2 + k_3),$$

kjer so:

$$k_0 = h\,f(t_i, y_i) \tag{12.1}$$
$$k_1 = h\,f\left(t_i + \frac{h}{2}, y_i + \frac{k_0}{2}\right) \tag{12.2}$$
$$k_2 = h\,f\left(t_i + \frac{h}{2}, y_i + \frac{k_1}{2}\right) \tag{12.3}$$
$$k_3 = h\,f\left(t_i + h, y_i + k_2\right). \tag{12.4}$$

Koraki metode Runge-Kutta četrtega reda so:

1. Določitev $i = 0$ in t_0, $y_0 = y(t_0)$,
2. Izračun koeficintov: k_0, k_1, k_2, k_3,
3. Izračun vrednosti rešitve diferencialne enačbe pri $t_{i+1} = t_i + h$:

   $$y_{i+1} = y_i + \frac{1}{6}(k_0 + 2\,k_1 + 2\,k_2 + k_3),$$

4. $i = i + 1$ in nadaljevanje v koraku 2.

12.4.1 Napaka metode Runge-Kutta četrtega reda

Metodo Runge-Kutta četrtega reda imenujemo tako zato, ker ima lokalno napako petega reda $\mathcal{O}(h^5)$, vendar pa to napako naredimo n-krat, zato je globalna napaka četrtega reda $\mathcal{O}(h^4)$.

Ocena napake

Točen rezultat $y(t_n)$ pri velikosti koraka h je:

$$y(t_n) = y_{n,h} + E_h,$$

kjer je $y_{n,h}$ numerični približek rešitve in E_h napaka metode. Ker je globalna napaka četrtega reda, lahko napako zapišemo tako:

$$E_h = k\,h^4.$$

Podobno lahko za velikost koraka $2h$ zapišemo:

$$y(t_n) = y_{n,2h} + E_{2h},$$

kjer je $y_{n,2h}$ numerični približek rešitve in E_{2h} napaka metode:

$$E_{2h} = k\,(2\,h)^4 = 16\,k\,h^4.$$

Ob predpostavki, da je konstanta k pri koraku h in koraku $2h$ enaka, lahko izračunamo oceno napake pri boljšem približku E_h.

Najprej je res:

$$y_{n,h} + k\,h^4 = y_{n,2h} + 16\,k\,h^4,$$

sledi:

$$15\,k\,h^4 = y_{n,h} - y_{n,2h}$$

in nato določimo *oceno napake* natančnejše rešitve:

$$E_h = \frac{y_{n,h} - y_{n,2h}}{15}.$$

12.4.2 Numerična implementacija

```
In [13]: def runge_kutta_4(f, t, y0, *args, **kwargs):
             """
             Metoda Runge-Kutta 4. reda za reševanje diferencialne enačbe: y' = f(t, y)

             :param f: funkcija, ki jo kličemo s parametroma t in y in vrne
                       vrednost prvega odvoda
             :param t: ekvidistanten časovni vektor oz. neodvisna spremenljivka
             :param y0: začetna vrednost
             :param args: dodatni argumenti funkcije f (brezimenski)
```

```
        :param kwargs: dodatni argumenti funkcije f (poimenovani)
        :return y: funkcijske vrednosti.
        """
        def RK4(f, t, y, *args, **kwargs):
            k0 = h*f(t, y, *args, **kwargs)
            k1 = h*f(t + h/2.0, y + k0/2.0, *args, **kwargs)
            k2 = h*f(t + h/2.0, y + k1/2.0, *args, **kwargs)
            k3 = h*f(t + h, y + k2, *args, **kwargs)
            return (k0 + 2.0*k1 + 2.0*k2 + k3)/6.0

        y = np.zeros_like(t)
        y[0] = y0
        h = t[1]-t[0]

        for i, ti in enumerate(t[1:]):
            y[i+1] = y[i] + RK4(f, ti, y[i], *args, **kwargs)

        return y
```

Funkcija za oceno napake:

```
In [14]: def runge_kutta_4_napaka(f, t, y0, *args, **kwargs):
             """ Ocena napake metode Runge Kutta 4; argumenti isti kakor za `runge_kutta_4`
             """
             n = len(t)
             if n < 5:
                 raise Exception('Vozlišč mora biti vsaj 5.')
             if n%2==0: # sodo vozlišč; odstrani eno točko in spremeni na liho (da je sodo odsekov)
                 n = n - 1
             y_h = runge_kutta_4(f, t[:n], y0, *args, **kwargs)
             y_2h = runge_kutta_4(f, t[:n:2], y0, *args, **kwargs)
             E_h = (y_h[-1] - y_2h[-1])/15
             return E_h
```

12.4.3 Numerični zgled

Poglejmo sedaj primer izračuna hitrosti padajoče mase:

```
In [15]: def f_zračni_upor(t, v, g=9.81, m=1., c=0.5):
             return g-c*v/m
```

Podajmo začetni pogoj in časovni vektor, kjer nas zanima rezultat:

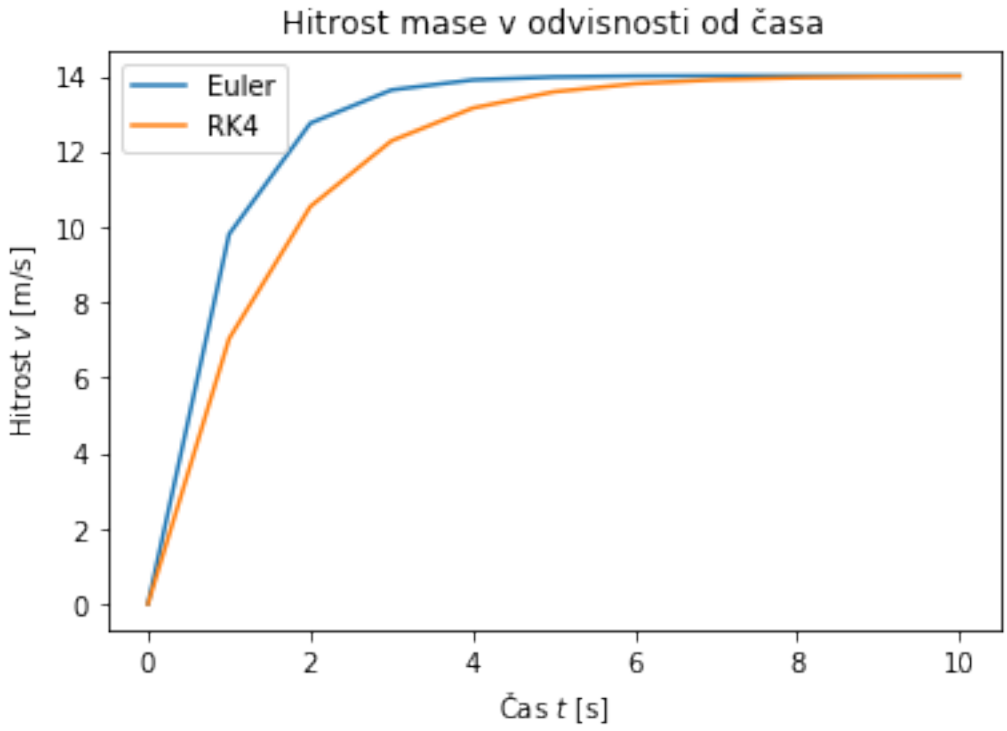
```
In [16]: v0 = 0
         t = np.linspace(0, 10, 11)
         t

Out[16]: array([  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
```

Za primerjavo izračunajmo rešitev s funkcijo `euler` ter `runge_kutta_4`:

```
In [17]: y_euler = euler(f_zračni_upor, t, y0=v0, c=0.7)
         y_rk4 = runge_kutta_4(f_zračni_upor, t, y0=v0, c=0.7)
```

Prikažemo rezultat:

```
In [18]: plt.plot(t, y_euler, label='Euler')
         plt.plot(t, y_rk4, label='RK4')
         plt.title('Hitrost mase v odvisnosti od časa')
         plt.xlabel('Čas $t$ [s]')
         plt.ylabel('Hitrost $v$ [m/s]')
         plt.legend()
         plt.show()
```



Poglejmo še numerično napako:

```
In [19]: n=101
         t = np.linspace(0, 10, n)
         runge_kutta_4_napaka(f_zračni_upor, t, y0=v0, c=0.7)
```

Out[19]:

$$2.01937583692e-08$$

```
In [20]: n=1001
         t = np.linspace(0, 10, n)
         runge_kutta_4_napaka(f_zračni_upor, t, y0=v0, c=0.7)
```

Out[20]:

$$1.81188397619e-12$$

Pri zmanjšanju koraka na desetino, se je napaka zmanjšala za približno 10^4-krat (kar ustreza pričakovanjem za metodo četrtega reda).

12.5 Uporaba `scipy` za reševanje navadnih diferencialnih enačb

Paket `scipy` ima implementiranih veliko numeričnih metod za reševanje začetnih problemov navadnih diferencialnih enačb. Pogledali si bomo dva pristopa:

- `scipy.integrate.odeint` (dokumentacija[1]),
- `scipy.integrate.ode` (dokumentacija[2]).

12.5.1 `scipy.integrate.odeint`

Sintaksa za uporabo:

```
scipy.integrate.odeint(func, y0, t, args=(), Dfun=None,
col_deriv=0, full_output=0, ml=None, mu=None, rtol=None,
atol=None, tcrit=None, h0=0.0, hmax=0.0, hmin=0.0, ixpr=0,
mxstep=0, mxhnil=0, mxordn=12, mxords=5, printmessg=0)
```

Pojasnilo vseh argumentov je v dokumentaciji[3], tukaj bomo izpostavili nekatere:

- `func` je desna stran* (`func(y, t, ...)`),
- `y0` je začetna* vrednost,
- `t` je numerično polje neodvisne spremenljivke, kjer računamo rešitve diferencialne enačbe,
- `args` je terka argumentov, ki jih lahko posredujemo v `func`.

Potrebni sta dve opombi:

1. `func(y, t, ...)`: vrstni red argumentov, najprej `y`, nato `t`!
2. `odeint` se lahko uporablja tudi za reševanje sistema diferencialnih enačb prvega reda (bomo spoznali spodaj) in takrat je `y` numerično polje vrednosti in ne skalar!

Numerični zgled

Poglejmo sedaj primer izračuna hitrosti padajoče mase; najprej pripravimo funkcijo desne strani, ki bo imela zamenjana argumenta `t` in `y` (kakor zahteva `odeint`):

```
In [21]: def f_zračni_upor_odeint(y, t, g=9.81, m=1., c=0.5):
             return g-c*y/m
```

Definirajmo začetni pogoj in časovni vektor, kjer nas zanima rezultat (prikažemo samo prvih deset elementov):

[1] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html
[2] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.ode.html
[3] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html

```
In [22]: v0 = 0
         t = np.linspace(0, 10, 101)
         t[:10]
```

```
Out[22]: array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9])
```

Za primerjavo izračunajmo rešitev s funkcijo `runge_kutta_4` ter `odeint` (koeficient zračnega upora naj bo `c=0.7`):

```
In [23]: from scipy.integrate import odeint

         y_odeint = odeint(f_zračni_upor_odeint, y0=v0, t=t, args=(9.81,1.,0.7))
         y_rk4 = runge_kutta_4(f_zračni_upor, t, y0=v0, c=0.7)
```

Poglejmo rezultat (samo prvih deset):

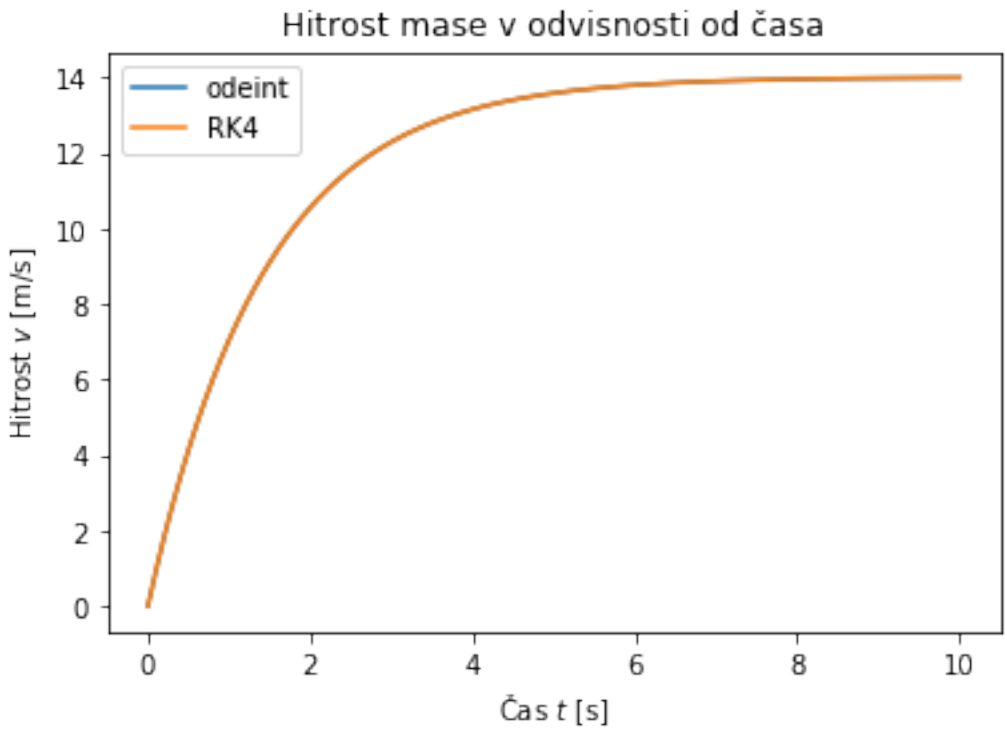
```
In [24]: y_odeint[:10,0]
```

```
Out[24]: array([ 0.        ,  0.94745234,  1.83085103,  2.6545265 ,  3.42251642,
                 4.13858549,  4.80624387,  5.42876441,  6.00919871,  6.55039207])
```

Ker je `odeint` že pripravljen za sistem navadnih diferencialnih enačb prvega reda, je rezultat podan v dveh dimenzijah. Iskan rezultat je torej: `y_odeint[:,0]`.

Prikažemo rezultat:

```
In [25]: plt.plot(t, y_odeint[:,0], label='odeint')
         plt.plot(t, y_rk4, label='RK4')
         plt.title('Hitrost mase v odvisnosti od časa')
         plt.xlabel('Čas $t$ [s]')
         plt.ylabel('Hitrost $v$ [m/s]')
         plt.legend()
         plt.show()
```



12.5.2 `scipy.integrate.ode`

`scipy` ponuja tudi bolj napredni integrator `ode`, ki pa temelji na objektnem pristopu in ponuja več prilagodljivosti kakor `odeint`. Instanco objekta dobimo s klicem:

```
scipy.integrate.ode(f, jac=None)
```

kjer je `f` desna stran diferencialne enačbe.

Metode objekta so:

- `integrate(t[, step, relax])` za izračun funkcijskih vrednosti `y`,
- `set_f_params(*args)` parametri funckije `f`,
- `set_initial_value(y[, t])` definiranje začetnih pogojev,
- `set_integrator(name, **integrator_params)` izbira metode,
- `set_jac_params(*args)` parametri Jacobijeve matrike,
- `set_solout(solout)` funkcija `solout`, ki se jo kliče pri vsakem uspešnem integracijskem koraku,
- `successful()` preveri, ali je reševanje bilo uspešno.

`ode` uporabimo v korakih (dokumentacija[4]):

1. naredimo instanco `ode`: `scipy.integrate.ode(f, jac=None)`,
2. definiramo metodo: metoda `set_integrator()`,

[4] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.ode.html

3. definiramo začetne vrednosti in parametre: metoda `set_initial_value()`,
4. izračunamo rešitev: metoda `integrate()`.

Poudariti moramo, da je pri `ode` vrstni red argumentov funkcije `f(t, y ...)`, kar je drugače kakor pri `odeint`, a konsistentno z izpeljavo pri `euler` in `runge_kutta_4`.

Naprej velja izpostaviti, da objekt `ode` omogoča izbiro različnih metod reševanja diferencialne enačbe; uporabili bomo metodo `'dopri5'`, ki sta jo predstavila Dormand in Prince. Metoda spada med metode četrtega oz. petega reda; razliko uporabi za oceno napake metode četrtega reda; napako potem izkoristi za prilagajanje velikosti koraka (več: J. Petrišič: Uvod v Matlab za inženirje, Fakulteta za strojništvo 2013).

Numerični zgled

Nadaljujemo numerični zgled iz prikaza uporabe funkcije `odeint`; najprej uvozimo razred `ode`:

```
In [26]: from scipy.integrate import ode
```

Pripravimo instanco objekta:

```
In [27]: y_ode = ode(f_zračni_upor)
```

Izberemo metodo, definiramo začetne pogoje in parametre:

```
In [28]: y_ode.set_integrator('dopri5')
         y_ode.set_initial_value(v0, t=t[0])
         y_ode.set_f_params(9.81, 1., 0.7);
```

Izračunamo rezultat:

```
In [29]: y_ode.integrate(t[-1])
```

```
Out[29]: array([ 14.00150497])
```

Primerjamo z rezultatom `y_odeint`:

```
In [30]: y_odeint[-1]
```

```
Out[30]: array([ 14.00150634])
```

Poudariti velja, da objekt `ode` prilagaja korak, da se doseže zahtevana natančnost (podamo jo lahko kot parameter metode `set_integrator()`) in vrne samo končno vrednost rešitve! Če želimo tabelo vrednosti, to naredimo v zanki:

```
In [31]: y_ode = ode(f_zračni_upor)
         y_ode.set_integrator('dopri5')
         y_ode.set_initial_value(v0, t=t[0])
         y_ode.set_f_params(9.81, 1., 0.7)
         t2 = np.array([0., 5., 10.])
         rezultat = np.zeros_like(t2)
         for i,čas in enumerate(t2):
             rezultat[i] = y_ode.integrate(čas)[0]
```

```
C:\Users\Janko\Anaconda3\lib\site-packages\scipy\integrate\_ode.py:1095: UserWarning: dopri5: step size
  self.messages.get(istate, unexpected_istate_msg)))
```

Rezultat ob času:

```
In [32]: t2
```

```
Out[32]: array([  0.,   5.,  10.])
```

torej je:

```
In [33]: rezultat
```

```
Out[33]: array([  0.        ,  13.5910896 ,  14.00150543])
```

12.6 Sistem navadnih diferencialnih enačb

Zgoraj smo si pogledali reševanje začetnega problema ene navadne diferencialne enačbe; sedaj bomo reševanje posplošili na *začetni problem za sistem m navadnih diferencialnih enačb prvega reda*.

Takšen sistem zapišemo:

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{y}),$$

kjer so podani dodatni (začetni) pogoji:

$$\mathbf{y}(t_0) = \mathbf{y}_0.$$

S t smo označili neodvisno spremenljivko (ni nujno, da je to vedno čas) in $\mathbf{f}$ vektor desnih strani.

Računamo rešitev sistema m diferencialnih enačb $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$ pri $a = t_0, t_1, \ldots, t_{n-1} = b$, to je m funkcijskih vrednosti na vsakem koraku $y_k(t_i)$, $k = 0, 1, \ldots, m-1$ in $i = 0, 1, \ldots, n-1$. Lahko se dokaže, da lahko uporabimo vsako metodo, ki smo jo izpeljali za diferencialne enačbe prvega reda tudi za sistem navadnih diferencialnih enačb prvega reda, če zamenjamo skalarne veličine z ustreznimi vektorskimi.

12.6.1 Numerična implementacija

Metodi Euler in Runge-Kutta četrtega reda potrebujeta zgolj malenkostne popravke (`y0` je numerično polje `f` vrne seznam vrednosti odvodov):

```
In [34]: def euler_sistem(f, t, y0, *args, **kwargs):
             """
             Eulerjeva metoda za reševanje sistema navadnih diferencialnih enačb prvega reda : y' = f(t

             :param f:  funkcija, ki jo kličemo s parametroma t in y in vrne seznam
                        funkcij desnih strani
             :param t:  ekvidistantni (časovni) vektor neodvisne spremenljivke
             :param y0: seznam začetnih vrednosti
             :param args: dodatni argumenti funkcije f (brezimenski)
             :param kwargs: dodatni argumenti funkcije f (poimenovani)
             :return y: vrne np.array ``y`` funkcijskih vrednosti.
```

```
    """
    y = np.zeros((t.shape[0], len(y0)))
    y[0] = np.copy(y0)
    h = t[1]-t[0]
    for i, ti in enumerate(t[:-1]):
        # tukaj je bistvo Eulerjeve metode
        y[i+1] = y[i]+f(ti, y[i], *args, **kwargs)*h
    return y
```

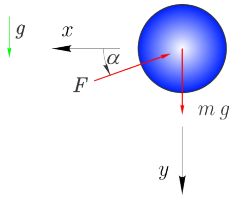
In [35]:
```
def runge_kutta_4_sistem(f, t, y0, *args, **kwargs):
    """
    Metoda Runge-Kutta 4. reda za reševanje sistema navadnih diferencialnih enačb prvega reda:

    :param f:  funkcija, ki jo kličemo s parametroma t in y in vrne seznam
               funkcij desnih strani
    :param t:  ekvidistantni (časovni) vektor neodvisne spremenljivke
    :param y0: seznam začetnih vrednosti
    :param args: dodatni argumenti funkcije f (brezimenski)
    :param kwargs: dodatni argumenti funkcije f (poimenovani)
    :return y: vrne np.array ``y`` funkcijskih vrednosti.
    """
    def RK4(g, t, y, *args, **kwargs):
        k0 = h*f(t, y, *args, **kwargs)
        k1 = h*f(t + h/2.0, y + k0/2.0, *args, **kwargs)
        k2 = h*f(t + h/2.0, y + k1/2.0, *args, **kwargs)
        k3 = h*f(t + h, y + k2, *args, **kwargs)
        return (k0 + 2.0*k1 + 2.0*k2 + k3)/6.0

    y = np.zeros((t.shape[0], len(y0)))
    y[0] = np.copy(y0)
    h = t[1]-t[0]

    for i, ti in enumerate(t[1:]):
        y[i+1] = y[i] + RK4(f, ti, y[i], *args, **kwargs)

    return y
```

Numerični zgled

Padanje mase nadgradimo v ravninsko gibanje; velikost sile upora zraka naj bo definirana kot:

$$|\mathbf{F}(\mathbf{v})| = c\,|\mathbf{v}|^2.$$

Sila upora zraka deluje v nasprotno stran kot kaže vektor hitrosti.



Ob pomoči slike, definiramo silo v x smeri:

$$F_x = -c\left(v_x^2 + v_y^2\right)\cos(\alpha) = -c\left(v_x^2 + v_y^2\right)\frac{v_x}{\sqrt{v_x^2 + v_y^2}} = -c\,v_x\sqrt{v_x^2 + v_y^2}.$$

Podobno je sila v y smeri:

$$F_y = -c\,v_y\sqrt{v_x^2 + v_y^2}.$$

Glede na drugi Newtonov zakon zapišemo sistem dveh (vezanih) diferencialnih enačb prvega reda:

$$F_x = m\,v_x',$$

$$m\,g + F_y = m\,v_y',$$

m je masa, g gravitacijski pospešek, c koeficient zračnega upora ter v_x in v_y hitrost v x oz y smeri. Diferencialno enačbo bi želeli rešiti glede na začetni pogoj:

$$v_x(0) = v_y(0) = 5\,\mathrm{m/s}.$$

Definirajmo seznam desnih strani:

```
In [36]: def f_zračni_upor_sila(t, y, F=1, g=9.81, m=1., c=0.5):
             vx, vy = y
             return np.array([-c*vx*np.sqrt(vx**2+vy**2)/m, g-c*vy*np.sqrt(vx**2+vy**2)/m])
```

Reševali bomo tudi z `odeint`, zato še obrnemo argumenta: `y` in `t`:

```
In [37]: def f_zračni_upor_sila_odeint(y, t, *args, **kwargs):
             return f_zračni_upor_sila(t, y, *args, **kwargs)
```

Definirajmo začetni pogoj in časovni vektor, kjer nas zanima rezultat:

```
In [38]: y0 = np.array([5., 5.])
         t = np.linspace(0, 5, 101)
         t[:5]
```

```
Out[38]: array([ 0.  ,  0.05,  0.1 ,  0.15,  0.2 ])
```

Za primerjavo izračunajmo rešitev s funkcijo `runge_kutta_4` ter `odeint` (koeficient zračnega upora naj bo `c=0.7`):

```
In [39]: y_RK4 = runge_kutta_4_sistem(f_zračni_upor_sila, y0=y0, t=t, F=1., c=0.7)
         y_odeint = odeint(f_zračni_upor_sila_odeint, y0=y0, t=t, args=(1., 9.81,1.,0.7))
```

Poglejmo rezultat:

```
In [40]: y_odeint[:5]
```

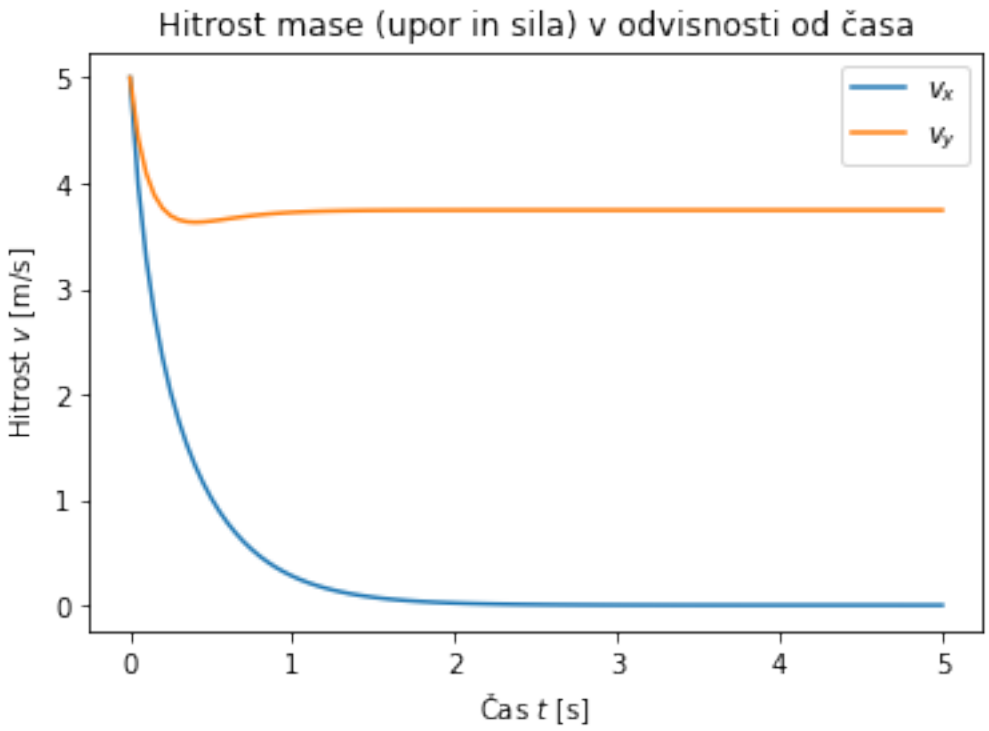
```
Out[40]: array([[ 5.        ,  5.        ],
                [ 3.98667382,  4.42701334],
                [ 3.27997334,  4.08879472],
                [ 2.75468453,  3.88468136],
                [ 2.34604393,  3.76199108]])
```

```
In [41]: y_RK4[:5]
```

```
Out[41]: array([[ 5.        ,  5.        ],
                [ 3.9867591 ,  4.42716544],
                [ 3.28006124,  4.08896796],
                [ 2.75475845,  3.88484045],
                [ 2.34610274,  3.76212744]])
```

Prikažemo rezultat:

```
In [42]: plt.plot(t, y_odeint[:,0], label='$v_x$')
         plt.plot(t, y_odeint[:,1], label='$v_y$')
         plt.title('Hitrost mase (upor in sila) v odvisnosti od časa')
         plt.xlabel('Čas $t$ [s]')
         plt.ylabel('Hitrost $v$ [m/s]')
         plt.legend()
         plt.show()
```



12.6.2 Preoblikovanje diferencialne enačbe višjega reda v sistem diferencialnih enačb prvega reda

Pogledali si bomo, kako navadno diferencialno enačbo poljubnega reda n:

$$y^{(n)} = f(t, y, y', y'', \ldots, y^{(n-1)}),$$

pri začetnih pogojih:

$$y(t_0) = k_0, \quad y'(t_0) = k_1, \quad \ldots \quad y^{(n-1)}(t_0) = k_{n-1}$$

preoblikujemo v sistem diferencialnih enačb prvega reda.
Najprej namesto odvodov vpeljemo nove spremenljivke:

$$y_i = y^{(i)}, \qquad i = 0, 1, \ldots, n-1.$$

Diferencialna enačba n-tega reda zapisana z novimi spremenljivkami je:

$$y'_{n-1} = f(t, y_0, y_1, y_2, \ldots, y_{n-1}).$$

Nove funkcije odvajamo po neodvisni spremenljivki:

$$y'_i = y^{(i+1)} = y_{i+1}, \qquad i = 0, 1, \ldots, n-2$$

in

$$y'_{n-1} = f(t, y_0, y_1, y_2, \ldots, y_{n-1}).$$

Dobili smo sistem navadnih diferencialnih enačb prvega reda:

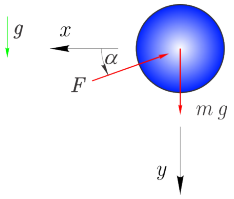
$$\begin{aligned} y'_0 &= y_1 \\ y'_1 &= y_2 \\ &\ldots \\ y'_{n-1} &= f(t, y_0, y_1, y_2, \ldots, y_{n-1}) \end{aligned}$$

pri začetenih pogojih:

$$y_0(t_0) = k_0, \quad y_1(t_0) = k_1, \quad \ldots \quad y_{(n-1)}(t_0) = k_{n-1}.$$

Numerični zgled

Vrnemo se k padajoči masi:



Vendar tokrat drugi Newtonov zakon zapišimo glede na pomik (diferencialna enačba drugega reda):

$$F_x = m\,x'',$$
$$m\,g + F_y = m\,y'',$$

m je masa, g gravitacijski pospešek, c koeficient zračnega upora ter x'' in y'' pospešek v izbranem koordinatnem sistemu. Začetni pogoji:

$$x(0) = y(0) = 0\,\mathrm{m} \qquad \text{in} \qquad x'(0) = y'(0) = 5\,\mathrm{m/s}.$$

Imamo sistem dveh diferencialnih enačb drugega reda. Z uvedbo novih spremenljivk y_i:

$$y_0 = x, \quad y_1 = x', \quad y_2 = y, \quad y_3 = y'.$$

Pripravimo sistem diferencialnih enačb prvega reda:

$$\begin{aligned} y_0' &= y_1 \\ y_1' &= F_x/m \\ y_2' &= y_3 \\ y_3' &= g + F_y/m \end{aligned}$$

Definirajmo Pythonovo funkcijo desnih strani / prvih odvodov (za funkcijo `odeint`):

```
In [43]: def f_zračni_upor_vezana_odeint(y, t, g=9.81, m=1., c=0.5):
             x, vx, y, vy = y
             return np.array([vx, -c*vx*np.sqrt(vx**2+vy**2)/m, vy, g-c*vy*np.sqrt(vx**2+vy**2)/m])
```

Definirajmo začetni pogoj in časovni vektor, kjer nas zanima rezultat:

```
In [44]: y0 = np.array([0., 5., 0., 5.])
         t = np.linspace(0, 5, 101)
         t[:5]

Out[44]: array([ 0.  ,  0.05,  0.1 ,  0.15,  0.2 ])
```

Rešimo s funkcijo `odeint` (koeficient zračnega upora naj bo `c=0.7`):

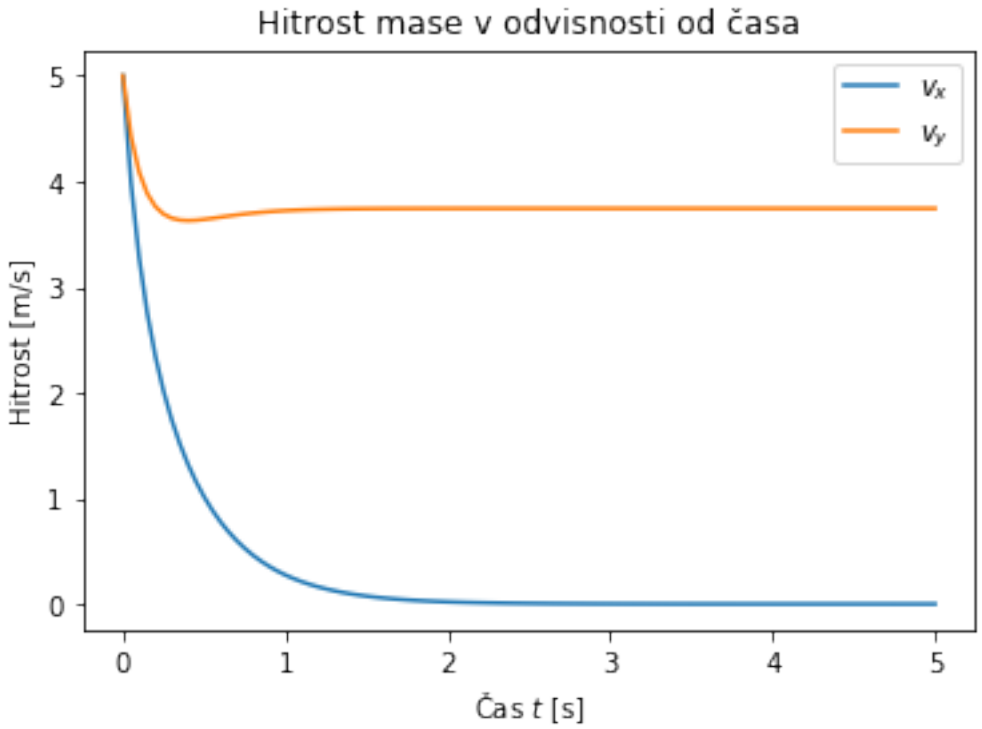
```
In [45]: y_odeint = odeint(f_zračni_upor_vezana_odeint, y0=y0, t=t, args=(9.81,1.,0.7))
```

Poglejmo rezultat ($[x, x', y, y']$):

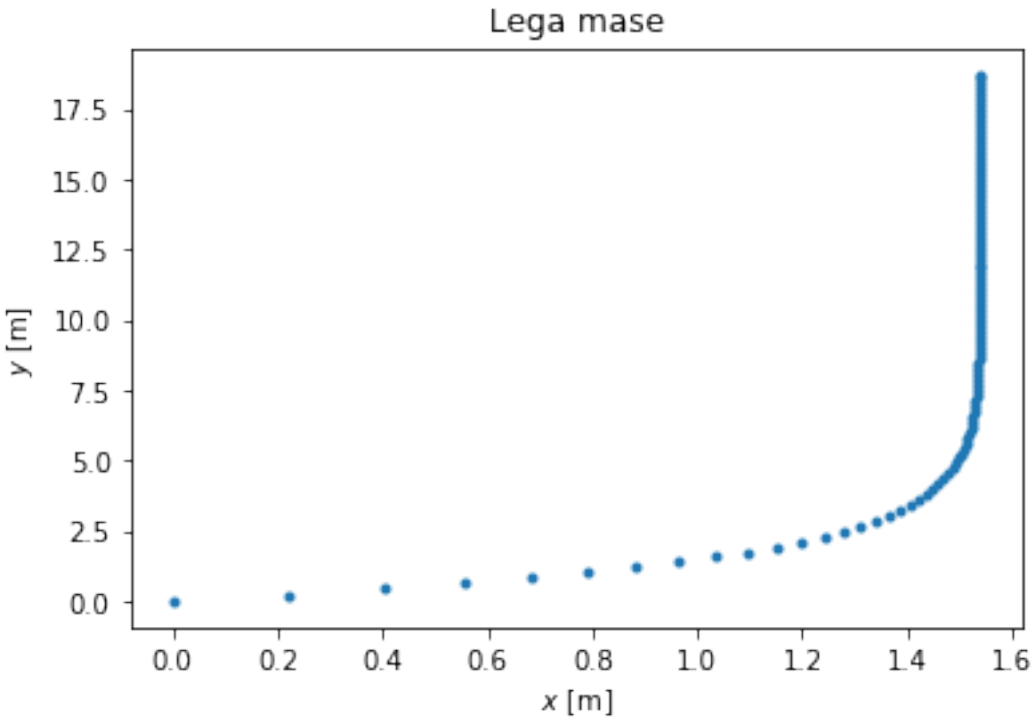
```
In [46]: y_odeint[:5]

Out[46]: array([[ 0.        ,  5.        ,  0.        ,  5.        ],
                [ 0.22298906,  3.98667382,  0.23437708,  4.42701334],
                [ 0.40370455,  3.27997334,  0.44655614,  4.08879472],
                [ 0.55397936,  2.75468453,  0.64546737,  3.88468136],
                [ 0.6811026 ,  2.34604393,  0.83636969,  3.76199108]])
```

Prikažemo rezultate; najprej hitrost, nato lego (y koordinata je pozitivna navzdol)!

```
In [47]: plt.plot(t, y_odeint[:,1], label='$v_x$')
         plt.plot(t, y_odeint[:,3], label='$v_y$')
         plt.title('Hitrost mase v odvisnosti od časa')
         plt.xlabel('Čas $t$ [s]')
         plt.ylabel('Hitrost [m/s]')
         plt.legend()
         plt.show()
```




```
In [48]: plt.scatter(y_odeint[:,0], y_odeint[:,2], marker='.')
         plt.xlabel('$x$ [m]')
         plt.ylabel('$y$ [m]')
         plt.title('Lega mase')
         plt.show()
```



12.7 Stabilnost reševanja diferencialnih enačb*

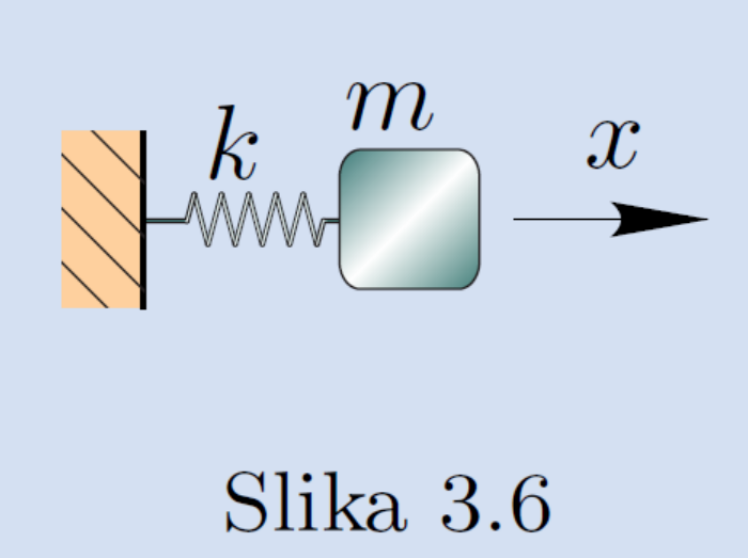
Numerično reševane diferencialnih enačb je izpostavljeno *napaki metode* in *zaokrožitveni napaki*. Te napake so pri različnih numeričnih metodah različne.

Reševanje diferencialne enačbe je **stabilno**, če majhna sprememba začetnega pogoja vodi v majhno spremembo izračunane rešitve; sicer govorimo o **nestabilnosti reševanja**.

Stabilnost je odvisna od diferencialne enačbe, od uporabljene numerične metode in od koraka integracije h.

12.7.1 Primer preprostega nihala

Poglejmo si najprej primer reševanja diferencialne enačbe preprostega nihala.



Slika 3.6

Slika (vir: Slavič, Dinamika, mehanska nihanja in mehanika tekočin, 2014) prikazuje dinamski sistem (masa m, togost k), katerega diferencialna enačba je

$$m\,x'' + k\,x = 0.$$

Tako diferencialno enačbo preoblikujemo v standardno obliko lastnega nihanja:

$$x'' + \omega_0^2\,x = 0,$$

kjer je lastna krožna frekvenca:

$$\omega_0 = \sqrt{\frac{k}{m}}$$

in pričakujemo odziv oblike:

$$x(t) = A\,\cos(\omega_0\,t) + B\,\sin(\omega_0\,t).$$

Če so začetni pogoji:

$$x(0\,\mathrm{s}) = x_0 \qquad \text{in} \qquad x'(0\,\mathrm{s}) = 0\,\mathrm{m/s},$$

je rešitev začetnega problema:

$$x(t) = x_0\,\cos(\omega_0\,t).$$

Numerični zgled

Najprej definirajmo vektor začetnih pogojev in funkcijo desnih strani / prvih odvodov (diferencialno enačbo drugega reda pretvorimo v sistem diferencialnih enačb prvega reda $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$), kjer velja $y_0 = x, y_1 = x'$:

```
In [49]: def f_nihalo(t, y, omega0=2*np.pi):
             """
             Funkcija desnih strani za nihalo z eno prostostno stopnjo

             :param t: čas
             :param y: seznam začetnih vrednosti
             :param omega: lastna krožna frekvenca
             :return y': seznam vrednosti odvodov
             """
             return np.array([y[1], -omega0**2*y[0]])
```

Definirajmo podatke in analitično rešitev:

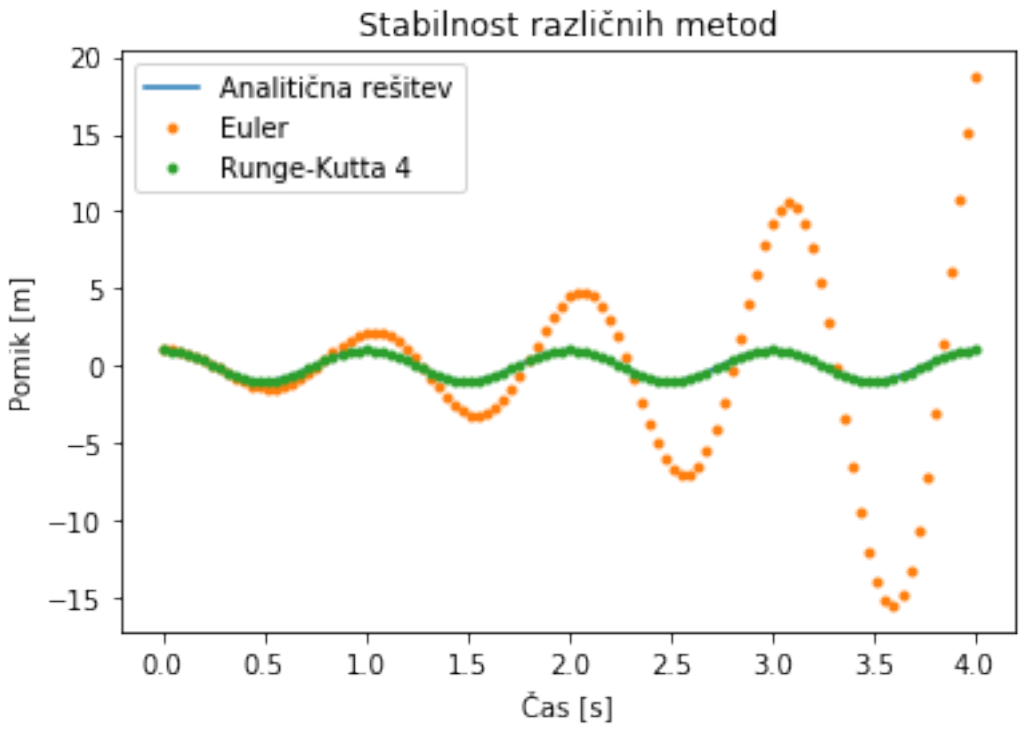
```
In [50]: x0 = 1.
         omega0 = 2*np.pi
         x_zacetni_pogoji = np.array([x0, 0.])
         t1 = 4.
         cas = np.linspace(0, t1, 500)
         pomik = x0*np.cos(omega0*cas) # analitična rešitev
         hitrost = -x0*omega0*np.sin(omega0*cas) # analitična rešitev
```

Rešitev s pomočjo metod Euler in Runge-Kutta četrtega reda:

```
In [51]: t_Eu = np.linspace(0, t1, 101)
         t_RK4 = t_Eu
         dt = t_Eu[1]

         x_Eu = euler_sistem(f_nihalo, t_Eu, x_zacetni_pogoji)
         x_RK4 = runge_kutta_4_sistem(f_nihalo, t_RK4, x_zacetni_pogoji)
```

Prikažimo rezultate:

```
In [52]: plt.plot(cas, pomik, label='Analitična rešitev')
         plt.plot(t_Eu, x_Eu[:,0], '.', label='Euler')
         plt.plot(t_RK4, x_RK4[:,0], '.', label='Runge-Kutta 4')
         plt.legend()
         plt.title('Stabilnost različnih metod')
         plt.ylabel('Pomik [m]')
         plt.xlabel('Čas [s]')
         plt.show()
```



Opazimo, da je Eulerjeva metoda nestabilna in če bi povečali korak, bi postala nestabilna tudi metoda Runge-Kutta četrtega reda.

Zakaj je Eulerjeva metoda tako nestabilna?

```
In [53]: x_Eu[:10,0]
```

```
Out[53]: array([ 1.        ,  1.        ,  0.93683453,  0.8105036 ,  0.62499707,
                 0.3882947 ,  0.1121141 , -0.18859332, -0.49638247, -0.79225904])
```

Spomnimo se Eulerjeve metode:

$$x(t+h) = x(t) + x'(t)\,h,$$

ki nam pove, da pomik $x(t+h)$ določimo glede na lego $x(t)$ in hitrost $x'(t)$. Začetni pogoji izhajajo iz skrajnega odmika $x(t=0) = 1$ in takrat je hitrost $x'(t=0) = 0$, kar pomeni, da bo $x(t+h) = 1$. Že v prvem koraku torej naredimo razmeroma veliko napako. Vendar zakaj potem začne vrednost alternirajoče naraščati?

Spomnimo se, da je analitična rešitev $x(t) = x_0\,\cos(\omega_0\,t)$ in je torej $x'(t) = -\omega_0\,x_0\,\sin(\omega_0\,t)$.

Vstavimo pripravljena izraza v Eulerjevo metodo in uredimo:

$$x(t+h) = x(t) + x'(t)\,h = x_0\,\left(\cos(\omega_0\,t) - \omega_0\,h\,\sin(\omega_0\,t)\right).$$

Predpostavimo, da gledamo stanje ob takem času $t = \pi/(2\omega_0)$, ko velja $\cos(\omega_0 t) = 0$ in $\sin(\omega_0 t) = 1$:

$$x(t+h) = x_0 \underbrace{(-\omega_0 h)}_{A}.$$

V kolikor bo absolutna vrednost izraza A večja kot 1, bo pri času $t + h$ vrednost večja kot v predhodnem koraku in v sledečem verjetno spet. Sledi, da lahko pride do nestabilnosti. Da se je izognemo, mora veljati:

$$|A| < 1 \qquad \rightarrow \qquad h < \frac{1}{\omega_0}.$$

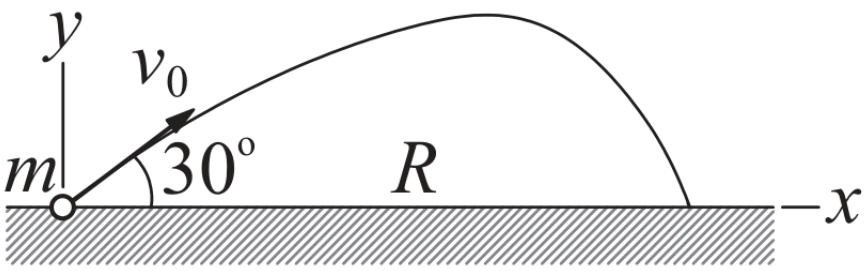
Opomba: v nekaterih knjigah boste videli tudi vrednost $h < 2/\omega_0$; enoliče meje za vse diferencialne enačbe ni mogoče definirati; v splošnem pa velja, da je korak definiran relativno glede na najkrajšo periodo T v diferencialni enačbi (npr.: $h < 2/\omega_0$ je v bistvu enako $h < 2/(2\pi/T)$ oziroma $h < T/\pi$). Perioda T je definirana glede na najvišjo lastno frekvenco sistema $T = 1/f_{\max}$, ki jo izračunamo iz lastne vrednosti sistema.

12.8 Nekaj vprašanj za razmislek!

1. Na sliki (vir: Numerical Methods in Engineering With Python 3, 3rd Ed, Jaan Kiusalaas) je prikazan izstrelek mase m, ki ga izstrelimo s hitrosjo v_0 pod kotom α. Če je sila upora zraka: $F = c\, v^{3/2}$, potem sta gibalni enačbi:

   $$\ddot{x}(t) = -F\,\cos(\alpha)/m \qquad \ddot{y}(t) = -F\,\sin(\alpha)/m - g.$$

   Opomba: $v = \sqrt{\dot{x}^2 + \dot{y}^2}$. Ustrezne parametre si izmislite.



   Sistem dveh diferencialnih enačb drugega reda zapišite v sistem diferencialnih enačb prvega reda.
2. Določite vektor začetnih pogojev, ki smo ga zgoraj označili z **y**.
3. Določite funkcijo desnih strani, c naj bo parameter.
4. Definirajte začetne pogoje in rešite nalogo s poljubnimi podatki.
5. Prikažite (x, y) lego masne točke, spreminjajte koeficient upora c.
6. Prikažite hitrost v odvisnosti od časa. Določite minimum hitrosti in čas, pri katerem nastane.

12.8.1 Primer Van der Polovega nihala

Namen tega primera je pokazati, kako lahko izbira integratorja vpliva na hitrost reševanja problema! Van der Polovo nihalo je opisano tukaj[5].

[5] http://en.wikipedia.org/wiki/Van_der_Pol_oscillator

Definirajmo seznam odvodov:

```
In [54]: def f_van_der_pol(t, y, mu=1000):
             """
             Funkcija desnih strani za Van der Pol nihalo

             :param t: čas
             :param y: seznam začetnih vrednosti
             :param mu: parameter dušenja in nelinearnosti
             :return y': seznam vrednosti odvodov
             """
             return np.array([y[1], mu*(1-y[0]**2)*y[1]-y[0]])
```

```
In [55]: x_zacetni_pogoji = np.array([1.5, 0.])
         dt = 0.1
         t1 = 3000
```

Rešitev po metodi RK45:

```
In [56]: #%%timeit -n 1
         solver = ode(f_van_der_pol).set_integrator('dopri5').set_initial_value(x_zacetni_pogoji)
         t_RK4_sci = [0]
         x_RK4_sci = [x_zacetni_pogoji]
         while solver.successful() and solver.t < t1/6:# računamo samo do 1/6 časa!!!
             solver.integrate(solver.t+dt)
             t_RK4_sci.append(solver.t)
             x_RK4_sci.append(solver.y)
         t_RK4_sci = np.array(t_RK4_sci)
         x_RK4_sci = np.array(x_RK4_sci)
```

Rešitev po implicitni metodi tipa Adams:

```
In [57]: #%%timeit -n 1
         solver = ode(f_van_der_pol).set_integrator('lsoda').set_initial_value(x_zacetni_pogoji)
         t_imp_sci = [0]
         x_imp_sci = [x_zacetni_pogoji]
         while solver.successful() and solver.t < t1:
             solver.integrate(solver.t+dt)
             t_imp_sci.append(solver.t)
             x_imp_sci.append(solver.y)
         t_imp_sci = np.array(t_imp_sci)
         x_imp_sci = np.array(x_imp_sci)
         solver.successful()
```

```
Out[57]: True
```

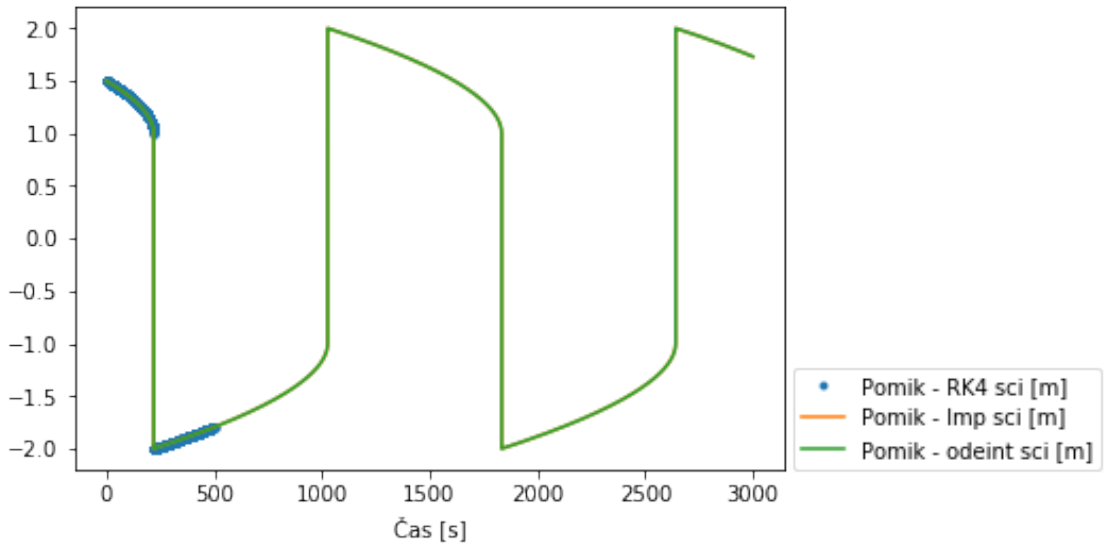
Rešitev s pomočjo funkcije `odeint` (uporablja isto implicitno metodo `lsoda` kot zgoraj, poleg tega moramo definirati, pri katerih časovnih korakih nas zanima rešitev).

```
In [58]: def f_van_der_pol_2(y, t):#odeint pričakuje najprej odvisne spremenljivke, nato neodvisne
             return f_van_der_pol(t, y, mu=1000)
```

```
In [59]: #%%timeit -n 1
         time = dt*np.arange(t1/dt)
         result = odeint(f_van_der_pol_2, x_zacetni_pogoji, time)
```

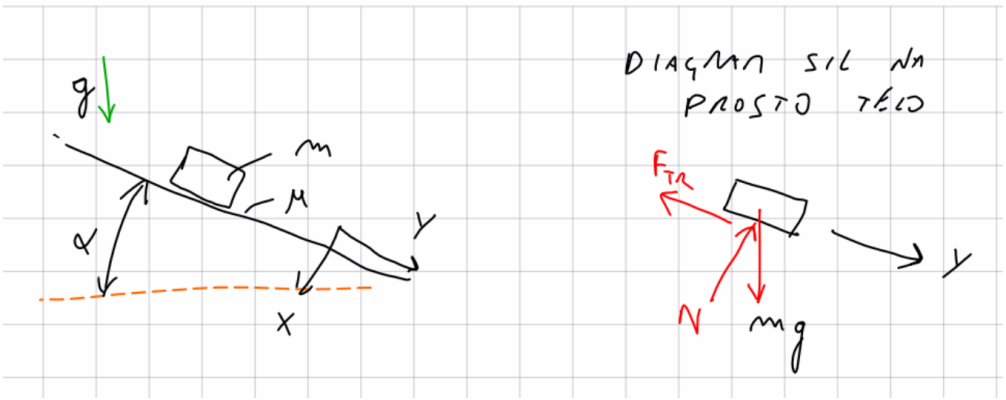
Prikaz rezultatov:

```
In [60]: plt.plot(t_RK4_sci, x_RK4_sci[:, 0], 'C0.', label='Pomik - RK4 sci [m]')
         plt.plot(t_imp_sci, x_imp_sci[:, 0], 'C1-', label='Pomik - Imp sci [m]')
         plt.plot(time, result[:, 0], 'C2-', label='Pomik - odeint sci [m]')
         plt.xlabel('Čas [s]')
         plt.legend(loc=(1.01, 0));
```



12.8.2 Simbolno reševanje diferencialne enačbe drugega reda

Pogledali si bomo primer, prikazan na sliki, kjer je masa m na klancu naklona α. Koeficient trenja je μ, težnostni pospešek pa g. Začetna hitrost je $\dot{y}(0\,\mathrm{s}) = v_0$, pomik $y(0\,\mathrm{s}) = 0\,\mathrm{m}$.



Gibalna enačba (samo za smer y) je definirana glede na II. Newtonov zakon (glejte diagram sil na prosto telo).

Izpeljava gibalne enačbe

```
In [61]: m, mu, g, alpha, y, t, v0 = sym.symbols('m, mu, g, alpha, y, t, v0')
         eq = sym.Eq(m*y(t).diff(t,2),  m*g*sym.sin(alpha)-m*g*sym.cos(alpha)*mu)
         eq
```

Out[61]:

$$m\frac{d^2}{dt^2}y(t) = -gm\mu\cos(\alpha) + gm\sin(\alpha)$$

Rešitev enačbe je:

```
In [62]: dsol = sym.dsolve(eq, y(t))
         dsol
```

Out[62]:

$$y(t) = C_1 + C_2 t + \frac{gt^2}{2}\left(-\mu\cos(\alpha) + \sin(\alpha)\right)$$

Da določimo C_1 in C_2, vstavimo $t = 0\,\mathrm{s}$:

```
In [63]: dsol.args[1].subs(t, 0)
```

Out[63]:

$$C_1$$

Nato odvajamo po času in ponovno vstavimo $t = 0\,\mathrm{s}$:

```
In [64]: dsol.args[1].diff(t).subs(t, 0)
```

Out[64]:

$$C_2$$

Glede na začetne pogoje smo torej določili konstante:

```
In [65]: zacetni_pogoji = {'C1': 0, 'C2': v0}
```

Sledi rešitev:

```
In [66]: resitev = dsol.args[1].subs(zacetni_pogoji)
         resitev
```

Out[66]:

$$\frac{gt^2}{2}\left(-\mu\cos\left(\alpha\right)+\sin\left(\alpha\right)\right)+tv_0$$

Pripravimo si funkciji za numerični klic:

```
In [67]: podatki = {mu: 0.3, alpha: 15*np.pi/180, v0: 1., g: 9.81} #tukaj uporabimo np.pi, da imamo num

         pomik = sym.lambdify(t, resitev.subs(podatki), 'numpy')
         hitrost = sym.lambdify(t, resitev.diff(t).subs(podatki), 'numpy')

         print('Pomik pri 0s: {:g}m'.format(pomik(0)))
         print('Hitrost pri 0s: {:g}m/s'.format(hitrost(0)))
```

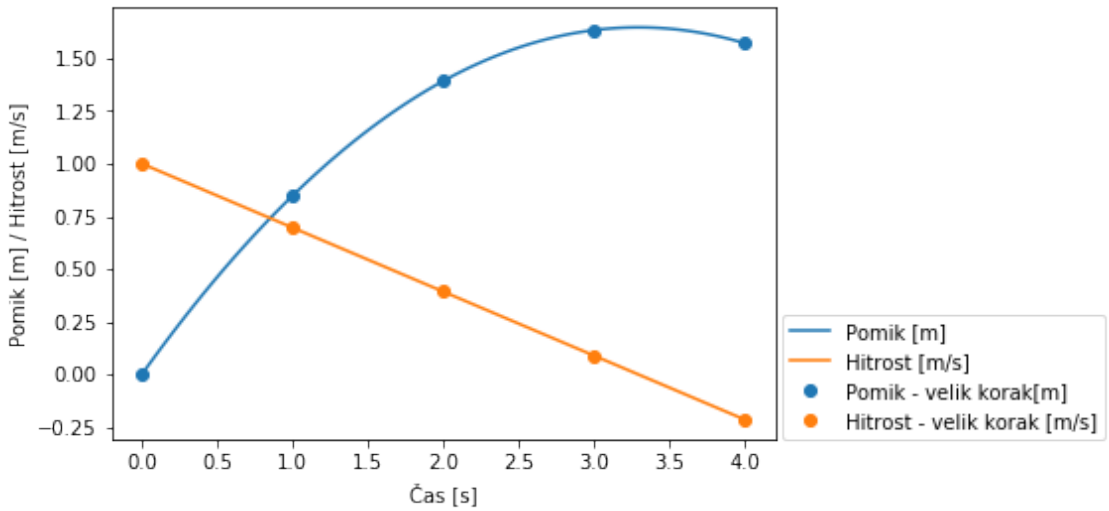
```
Pomik pri 0s: 0m
Hitrost pri 0s: 1m/s
```

Pripravimo prikaz:

```
In [68]: cas = np.linspace(0, 4, 100)
         cas2 = np.linspace(0, 4, 5)
```

```
In [69]: def slika():
             plt.plot(cas, pomik(cas), 'C0', label='Pomik [m]')
             plt.plot(cas, hitrost(cas), 'C1', label='Hitrost [m/s]')
             plt.plot(cas2, pomik(cas2), 'C0o', label='Pomik - velik korak[m]')
             plt.plot(cas2, hitrost(cas2), 'C1o', label='Hitrost - velik korak [m/s]')
             plt.xlabel('Čas [s]')
             plt.ylabel('Pomik [m] / Hitrost [m/s]')
             plt.legend(loc=(1.01, 0));
             plt.show()
```

```
In [70]: slika()
```



Simbolno preoblikovanje diferencialne enačbe v sistem diferencialnih enačb prvega reda

Spomnimo se izvorne diferencialne enačbe:

```
In [71]: eq
```

Out[71]:

$$m\frac{d^2}{dt^2}y(t) = -gm\mu\cos(\alpha) + gm\sin(\alpha)$$

Definirajmo nove spremenljivke in pripravimo funkcijo f:

```
In [72]: y0, y1 = sym.symbols('y:2')
         f = sym.simplify(eq.args[1]/m)
         f
```

Out[72]:

$$g\left(-\mu\cos(\alpha) + \sin(\alpha)\right)$$

Povežimo sedaj nove spremenljivke.

dy_0/dt naj bo enako y_1:

```
In [73]: eq1 = sym.Eq(y0(t).diff(t), y1(t))
         eq1
```

Out[73]:

$$\frac{d}{dt}\mathrm{y}_0(t) = \mathrm{y}_1(t)$$

Odvod dy_1/dt (v bistvu je to y'') naj bo enak funkciji f:

```
In [74]: eq2 = sym.Eq(y1(t).diff(t), f)
         eq2
```

Out[74]:

$$\frac{d}{dt}\mathrm{y}_1(t) = g\left(-\mu\cos(\alpha) + \sin(\alpha)\right)$$

Zgornje izraze zapišemo v vektorski obliki:

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{y}).$$

```
In [75]: y_odvod = [y0(t).diff(t), y1(t).diff(t)]
         y_odvod
```

Out[75]:

$$\left[\frac{d}{dt}\mathrm{y}_0(t),\quad \frac{d}{dt}\mathrm{y}_1(t)\right]$$

In [76]:
```
f_vec = [y1(t), f]
f_vec
```

Out[76]:

$$[\mathrm{y}_1(t), \quad g\left(-\mu\cos(\alpha)+\sin(\alpha)\right)]$$

Spomnimo se sedaj `f_vec`:

In [77]:
```
f_vec
```

Out[77]:

$$[\mathrm{y}_1(t), \quad g\left(-\mu\cos(\alpha)+\sin(\alpha)\right)]$$

Če rešujemo numerično, potem funkcijo $\mathbf{f}(t, \mathbf{y})$ zapišemo:

In [78]:
```
pospesek = float((eq.args[1]/m).simplify().subs(podatki))
pospesek #raziščite zakaj smo tukaj tako definirali! namig: type(pospesek)
```

Out[78]:

$$-0.3037048743129991$$

In [79]:
```
def F_klada(t, y):
    return np.array([y[1], pospesek],dtype=float)
```

Preverimo funkcijo pri začetnem času $t = 0\,\mathrm{s}$ in pri začetnih pogojih $[y_0, y_1] = [0, v_0]$:

In [80]:
```
y_zacetni_pogoji = np.array([0, podatki[v0]])
y_zacetni_pogoji
```

Out[80]:
```
array([ 0.,  1.])
```

In [81]:
```
F_klada(0., y_zacetni_pogoji)
```

Out[81]:
```
array([ 1.        , -0.30370487])
```

Uporabimo sedaj Eulerjevo metodo:

In [82]:
```
#%%timeit
x_Eu = np.linspace(0, 4, 5)
y_Eu = euler_sistem(F_klada, x_Eu, np.array([0, 1.]))
y_Eu
```

Out[82]:
```
array([[ 0.        ,  1.        ],
       [ 1.        ,  0.69629513],
       [ 1.69629513,  0.39259025],
       [ 2.08888538,  0.08888538],
       [ 2.17777075, -0.2148195 ]])
```

Prikažemo in primerjamo z analitično rešitvijo:

```
In [83]: def narisi_euler(n=5):
             x_Eu = np.linspace(0, 4, n)
             y_Eu = euler_sistem(F_klada, x_Eu, np.array([0, 1.]))
             plt.title('Eulerjeva metoda s korakom $h={:g}$'.format(x_Eu[1]-x_Eu[0]))
             plt.plot(cas, pomik(cas), 'C0', label='Pomik - analitično [m]')
             plt.plot(cas, hitrost(cas), 'C1', label='Hitrost - analitično [m/s]')
             plt.plot(x_Eu, y_Eu[:, 0], 'C0.', label='Pomik - Euler [m]')
             plt.plot(x_Eu, y_Eu[:, 1], 'C1.', label='Hitrost - Euler [m/s]')
             plt.xlabel('Čas [s]')
             plt.ylabel('Pomik [m] / Hitrost [m/s]')
             plt.ylim(-0.5, 2.5)
             plt.legend(loc=(1.01, 0))
             plt.show();

In [84]: interact(narisi_euler, n=(3, 10, 1));

A Jupyter Widget
```

Poglavje 13

Numerično reševanje diferencialnih enačb - robni problem

13.1 Reševanje dvotočkovnih robnih problemov

Pod dvotočkovnim robnim problemom razumemo navadno diferencialno enačbo drugega reda oblike:

$$\ddot{y} = f(t, y, \dot{y}),$$

ob predpisanih *robnih pogojih*:

$$y(a) = \alpha \qquad \text{in} \qquad y(b) = \beta.$$

Metode, ki smo jih spoznali pri reševanju *začetnih problemov*, tukaj neposredno niso uporabne, ker nimamo podanega odvoda v začetni točki pri $t = a$.

V nadaljevanju si bomo pogledali dva različna pristopa k reševanju robnih problemov:

1. t. i. **strelska metoda**,
2. **metoda končnih razlik**.

13.2 Strelska metoda

Rešujemo robni problem:

$$\ddot{y} = f(t, y, \dot{y}), \qquad y(a) = \alpha, \quad y(b) = \beta,$$

ki ga prevedemo na začetni problem tako, da si izberemo:

$$\dot{y}(a) = u.$$

Problem rešimo z numeričnimi metodami reševanja začetnega problema in rešitev označimo z $\theta(u,t)$.

Robni problem rešimo, ko izberemo u tako, da velja:

$$r(u) = \theta(u,b) - \beta = 0.$$

Dobili smo nelinearno enačbo, ki jo moramo rešiti; za izračun vrednosti mejnih preostankov $r(u)$ moramo numerično rešiti začetni problem.

Za rešitev enačbe $r(u) = 0$ lahko uporabimo sekantno metodo. Izberemo u_0 in u_1 in na i-tem koraku izračunamo:

$$u_{i+1} = u_i - r(u_i)\,\frac{u_i - u_{i-1}}{r(u_i) - r(u_{i-1})}, \qquad i = 2,3,\ldots$$

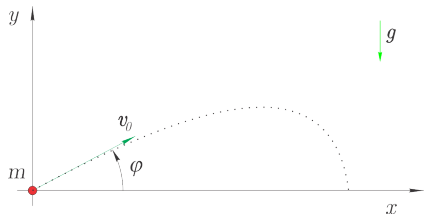
Zaključimo, ko je:

$$|r(u_{i+1})| < \epsilon.$$

Rešitev strelske metode je obremenjena z napako metode reševanja nelinearne enačbe ϵ in z napako numerične metode za reševanje začetnega problema.

13.2.1 Numerični zgled: poševni met

Na sliki je prikazan izstrelek mase m, ki ga izstrelimo s hitrostjo $\mathbf{v}_0$.



Velikost sile upora zraka je $|\mathbf{F}| = c\,|\mathbf{v}|^2$, potem sta gibalni enačbi:

$$x''(t) = -F_x/m \qquad \ddot{y}(t) = -F_y/m - g.$$

Komponente sile so (glejte izpeljavo pri poglavju iz reševanja začetnega problema sistema diferencialnih enačb):

$$F_x = -c\,x'\sqrt{x'^2 + y'^2}, \qquad F_y = -c\,y'\sqrt{x'^2 + y'^2}.$$

Vertikalni met

Najprej predpostavimo, da je $\varphi = 90°$ in torej v x smeri nimamo gibanja. Zanima nas rešitev, ko izstrelek izstrelimo iz višine $y = 0\,\mathrm{m}$ in mora pri času $t = b = 1\,\mathrm{s}$ biti na višini $y(b) = 10\,\mathrm{m}$. Definirali smo robni problem:

$$y''(t) = F_y/m - g, \qquad y(0) = 0, \quad y(1) = 10.$$

Najprej moramo enačbo drugega reda:

$$\ddot{y} = f(t, y, \dot{y}) = F_y/m - g$$

preoblikovati na sistem dveh enačb prvega reda. Uporabimo $y_i = y^i$ ter upoštevamo $F_y = -c\, y' \sqrt{y'^2}$. Odvajamo $\dot{y}_i = y_{i+1}$ in pripravimo sistem enačb prvega reda:

$$\begin{aligned} y_0' &= y_1 \\ y_1' &= -c\, y' \sqrt{y'^2}/m - g \end{aligned}$$

Pripravimo seznam funkcij desnih strani:

```
In [1]: def f_vert(y, t, g=9.81, m=1., c=0.5):
            return np.array([y[1], -g-c*y[1]*np.sqrt(y[1]**2)/m])
```

Uvozimo modula `numpy` in `scipy.integrate.odeint`:

```
In [2]: import numpy as np
        from scipy.integrate import odeint
```

ter pripravimo funkcijo za izračun mejnega preostanka pri času b v odvisnosti od začetne hitrosti $v0$ (privzeti zračni upor je $c = 0.1$):

```
In [3]: def r(v0=100., t=None, ciljna_lega=10., g=9.81, m=1., c=0.1):
            y_ode = odeint(f_vert, y0=np.array([0., v0]), t=t, args=(g, m, c))
            r = y_ode[-1,0] - ciljna_lega
            return r
```

Preverimo mejni preostanek pri začetnem pogoju $v_0 = y' = 50$ m/s:

```
In [4]: t = np.linspace(0, 1, 100)
        r(v0=50., t=t)
```

```
Out[4]: 5.6234300989312302
```

Opazimo, da je masa 1 sekundi 5,62 m nad ciljno višino.

Naš cilj je, da pri 1 sekundi masa doseže lego 10 m z natančnostjo `1e-6`:

```
In [5]: ciljna_lega = 10
        epsilon = 1e-6
```

Izvedimo sedaj sekantno metodo:

```
In [6]: x0 = 100
        x1 = 50
        for i in range(10):
            f0 = r(v0=x0, t=t, ciljna_lega=ciljna_lega)
            f1 = r(v0=x1, t=t, ciljna_lega=ciljna_lega)
```

```
        x2 = x1 - f1 * (x1 - x0)/(f1 - f0)
        err = np.abs(r(v0=x2, t=t, ciljna_lega=ciljna_lega))
        print(f'Novi približek je {x2}, napaka je {err}.')
        x0 = x1
        x1 = x2
        if err<epsilon:
            rešitev = x2
            print(f'Rešitev {rešitev}')
            break
```

```
Novi približek je 5.647986386236646, napaka je 9.668030082366473.
Novi približek je 33.68955970543041, napaka je 2.2166642331885793.
Novi približek je 28.459408136648648, napaka je 0.8222440076866793.
Novi približek je 25.375358789059757, napaka je 0.10146604219394106.
Novi približek je 25.714129901565347, napaka je 0.004272903915547133.
Novi približek je 25.700440183005067, napaka je 2.1606281411123973e-05.
Novi približek je 25.700370608033587, napaka je 4.616973470206176e-09.
Rešitev 25.700370608033587
```

Poglejmo si sedaj izračunano rešitev:

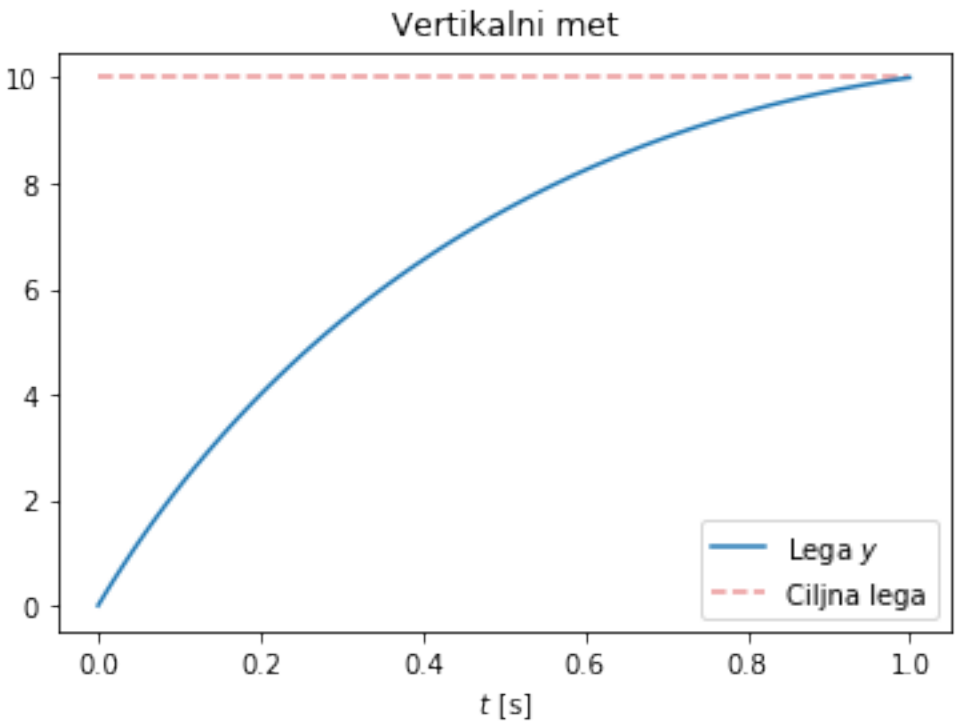
```
In [7]: y_vert = odeint(f_vert, y0=np.array([0., rešitev]), t=t, args=(9.81, 1, 0.1))
```

Uvozimo `matplotlib`:

```
In [8]: import matplotlib.pyplot as plt
        %matplotlib inline
```

Prikažemo rezultat:

```
In [9]: plt.title('Vertikalni met')
        plt.hlines(ciljna_lega,0., 1, 'C3', linestyles='dashed', label='Ciljna lega', alpha=0.5)
        plt.plot(t, y_vert[:,0], label='Lega $y$')
        plt.xlabel('$t$ [s]')
        plt.legend(loc=4)
        plt.show()
```



Poševni met

Poglejmo si sedaj splošen poševni met (diferencialne enačbe so definirana že zgoraj). Najprej moramo sistem diferencialnih enačb drugega reda preoblikovati v sistem enačb prvega reda.

Uporabimo:

$$y_0 = x,\ y_1 = x',\ y_2 = y,\ y_3 = y'$$

in dobimo sistem diferencialnih enačb prvega reda:

$$\begin{aligned} \dot{y}_0 &= y_1 \\ \dot{y}_1 &= F_x/m \\ \dot{y}_2 &= y_3 \\ \dot{y}_3 &= F_y/m - g. \end{aligned}$$

Pripravimo seznam funkcij desnih strani:

```
In [10]: def f_poševno(y, t, g=9.81, m=1., c=0.1):
             x, vx, yy, vy = y
             return np.array([vx, -c*vx*np.sqrt(vx**2+vy**2)/m, vy, -g-c*vy*np.sqrt(vx**2+vy**2)/m])
```

Pripravimo še funkcijo za izračun mejnega preostanka pri času $b = 1$ v odvisnosti od vektorja začetne hitrosti $\mathbf{v}_0$:

```
In [11]: def r_poševno(v0=[5., 100.], t=None, ciljna_lega=np.array([10, 5.]), g=9.81, m=1., c=0.1):
             y_ode = odeint(f_poševno, y0=[0, v0[0], 0, v0[1]], t=t, args=(g, m, c))
             r = y_ode[-1,0:3:2] - ciljna_lega
             return r
```

Preverimo mejni preostanek pri začetnem pogoju $\mathbf{v}_0 = [100., 100.]\,\mathrm{m/s}$:

```
In [12]: r_poševno(v0=[100., 100.],t=t)
```

```
Out[12]: array([  9.64208677,  11.82617591])
```

Za iskanje korena sistema nelinearnih funkcij smo že spoznali funkcijo `scipy.optimize.root` (dokumentacija[1]):

```
root(fun, x0, args=(), method='hybr', jac=None, tol=None, callback=None, options=None)
```

Najprej jo uvozimo:

```
In [13]: from scipy import optimize
```

Potem uporabimo z začetnim ugibanjem $\mathbf{v}_0$:

```
In [14]: rešitev = optimize.root(r_poševno, np.array([100., 100.]), args=(t))
```

Rešitev je:

```
In [15]: rešitev
```

```
Out[15]:     fjac: array([[-0.93420153,  0.35674572],
                 [-0.35674572, -0.93420153]])
              fun: array([  2.75335310e-13,  -3.28626015e-14])
          message: 'The solution converged.'
             nfev: 17
              qtf: array([  9.64776823e-10,  -1.20885598e-09])
                r: array([-0.40653983,  0.24107452, -0.37581076])
           status: 1
          success: True
                x: array([ 19.38918337,  16.56317825])
```

Atribut `rešitev.x` vsebuje vektor izračunanih rešitev (`[19.3892, 16.5632]`). Preverimo mejni preostanek pri izračunani rešitvi:

```
In [16]: r_poševno(v0=rešitev.x, t=t)
```

```
Out[16]: array([  2.75335310e-13,  -3.28626015e-14])
```

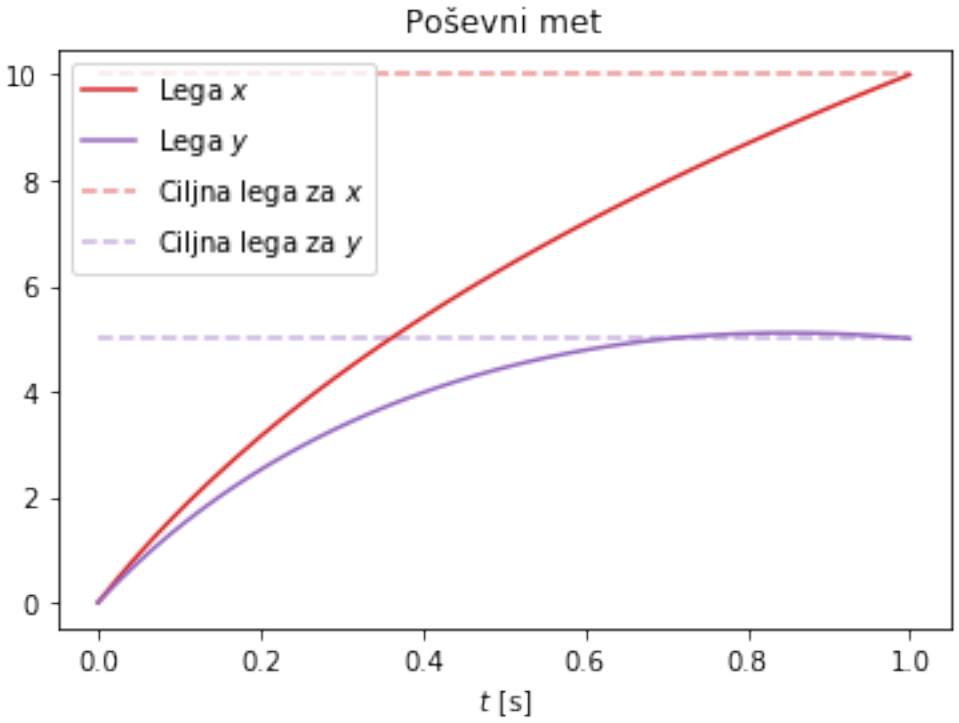
Poglejmo si sedaj izračunano rešitev:

```
In [17]: y_poševni = odeint(f_poševno, y0=[0, rešitev.x[0], 0, rešitev.x[1]], t=t, args=(9.81, 1., 0.1)
```

[1] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.root.html

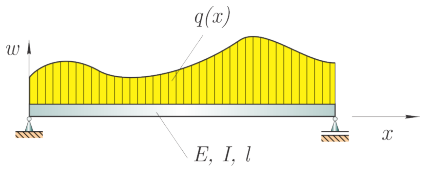
Prikažemo rezultat:

```
In [18]: plt.title('Poševni met')
         plt.hlines(10,0., 1, 'C3', linestyles='dashed', label='Ciljna lega za $x$', alpha=0.5)
         plt.hlines(5,0., 1, 'C4', linestyles='dashed', label='Ciljna lega za $y$', alpha=0.5)
         plt.plot(t, y_poševni[:,0], label='Lega $x$', color='C3')
         plt.plot(t, y_poševni[:,2], label='Lega $y$', color='C4')
         plt.xlabel('$t$ [s]')
         plt.legend()
         plt.show()
```



13.2.2 Numerični zgled: nosilec z obremenitvijo

Poglejmo si nosilec:



Poves $w(x)$ nosilca popiše diferencialna enačba četrtega reda:

$$-E\,I\frac{\mathrm{d}^4}{\mathrm{d}x^4}w(x)+q(x)=0.$$

Znane konstante so E, I, l in je $q(x)$ porazdeljena obremenitev.

Robni pogoji (členkasto vpet nosilec):

$$w(0) = w(l) = 0 \quad \text{in} \quad w''(0) = w''(l) = 0$$

Parametri so:

- $I = 2.1 \cdot 10^{-5}\,\mathrm{m}^4$,
- $E = 2.1 \cdot 10^{11}\,N/\mathrm{m}^2$,
- $l = 10\,\mathrm{m}$.

Porazdeljena obremenitev $q(x)$ bo definirana pozneje.

Najprej moramo diferencialno enačbo četrtega reda preoblikovati v sistem diferencialnih enačb prvega reda. Uporabimo:

$$y_0 = w,\ y_1 = w',\ y_2 = w'',\ y_3 = w'''$$

in dobimo sistem diferencialnih enačb prvega reda:

$$\begin{aligned} y_0' &= y_1 \\ y_1' &= y_2 \\ y_2' &= y_3 \\ y_3' &= q(x)/(EI). \end{aligned}$$

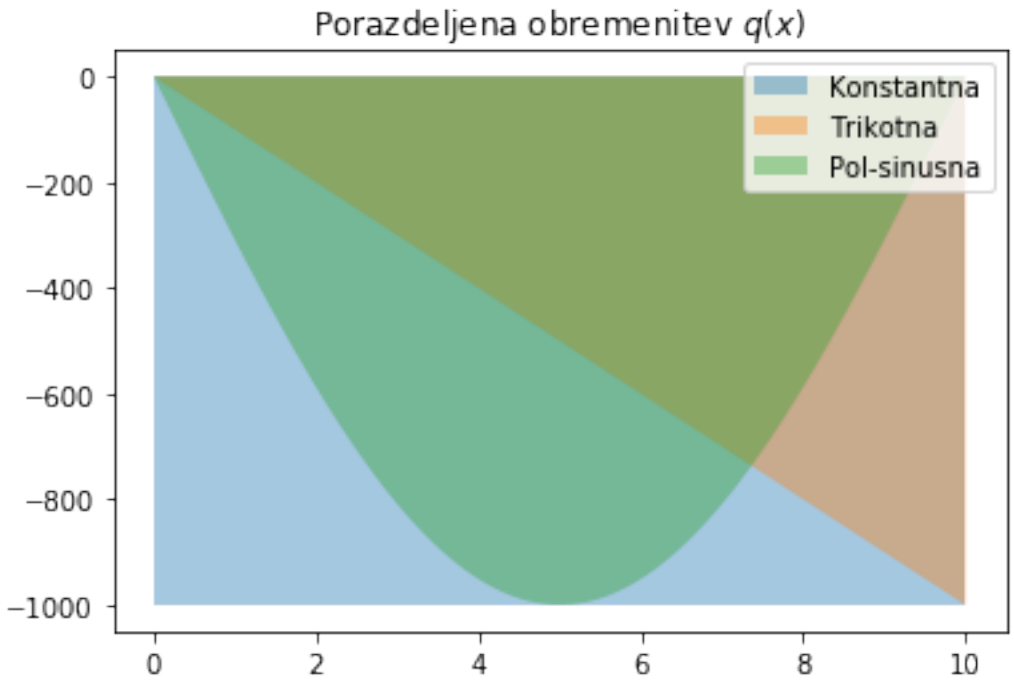
Pripravimo različne porazdeljene obremenitve:

```
In [19]: def q_konstanta(x, F_0=1e3, l=10):
             return -F_0

         def q_trikotna(x, F_0=1e3, l=10):
             return -F_0*x/l

         def q_pol_sinusna(x, F_0=1e3, l=10):
             return -F_0*np.sin(np.pi*x/l)
```

Definirajmo parametre:

```
In [20]: l = 10.
         E = 2.1e11
         I = 2.1e-5
```

```
In [21]: x = np.linspace(0, l, 100)
         plt.fill_between(x, q_konstanta(x), alpha=0.4, label='Konstantna')
         plt.fill_between(x, q_trikotna(x), alpha=0.4, label='Trikotna')
         plt.fill_between(x, q_pol_sinusna(x), alpha=0.4, label='Pol-sinusna')
         plt.title('Porazdeljena obremenitev $q(x)$')
         plt.legend()
         plt.show()
```



Pripravimo seznam funkcij desne strani:

```
In [22]: def f_nosilec_konstanta(y, x, E=2.1e11, I=0.4):
             return np.array([y[1], y[2], y[3], q_konstanta(x)/(E*I)])
         def f_nosilec_pol_sinusna(y, x, E=2.1e11, I=0.4):
             return np.array([y[1], y[2], y[3], q_pol_sinusna(x)/(E*I)])
         def f_nosilec_trikotna(y, x, E=2.1e11, I=0.4):
             return np.array([y[1], y[2], y[3], q_trikotna(x)/(E*I)])
```

Prikažimo primer izračuna:

```
In [23]: y_ode = odeint(f_nosilec_konstanta, y0=[0, 0.1, 0, 1], t=x, args=(E, I))
         y_ode[:4]

Out[23]: array([[ 0.        ,  0.1       ,  0.        ,  1.        ],
                [ 0.01027279,  0.10510148,  0.10100894,  0.9999771 ],
                [ 0.02157617,  0.12040577,  0.20201557,  0.99995419],
                [ 0.03494071,  0.14591263,  0.30301989,  0.99993129]])
```

ter pripravimo funkcijo za izračun mejnega preostanka pri času $x = L$:

```
In [24]: def r_nosilec_konstanta(y0=[1., 1.], x=None, ciljne_vrednosti=np.array([0, 0.]), E=2.1e11, I=2
             y_ode = odeint(f_nosilec_konstanta, y0=[0, y0[0], 0, y0[1]], t=x, args=(E, I))
             r = y_ode[-1,0:3:2] - ciljne_vrednosti
             return r
```

Za konstantno obremenitev preverimo mejni preostanek pri $w'(0) = 1$ in $w'''(0) = 1$ (za naklon w' je to kar velika vrednost, za tretji odvod w''' pa nimamo občutka):

```
In [25]: r_nosilec_konstanta(y0=[1., 1.], x=x)
```

```
Out[25]: array([ 176.57218444,    9.98866213])
```

Mejni preostanek je velik, vendar lahko pričakujemo, da bomo koren enačbe mejnega preostanka lahko izračunali. Sistem nelinearnih enačb bomo rešili s `scipy.optimize.root()`:

```
In [26]: rešitev = optimize.root(r_nosilec_konstanta, np.array([0., 0.]), args=(x))
```

Rešitev je:

```
In [27]: rešitev.x
```

```
Out[27]: array([-0.00944822,  0.00113379])
```

Mejni preostanek pri najdeni rešitvi:

```
In [28]: r_nosilec_konstanta(rešitev.x, x)
```

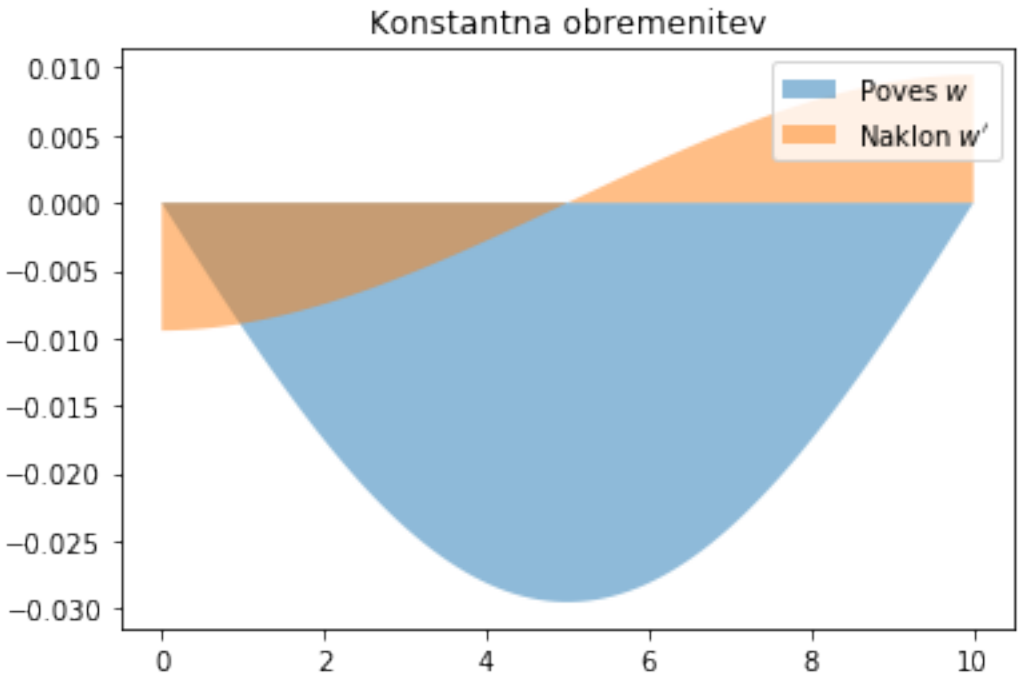
```
Out[28]: array([  1.00353753e-15,  -1.06251813e-17])
```

Pripravimo sedaj še izračun rezultatov pri najdenih začetnih pogojih:

```
In [29]: y = odeint(f_nosilec_konstanta, y0=[0, rešitev.x[0], 0, rešitev.x[1]], t=x, args=(E, I))
```

Rezultat prikažimo:

```
In [30]: plt.fill_between(x, y[:,0], label='Poves $w$', alpha=0.5)
         plt.fill_between(x, y[:,1], label='Naklon $w\'$', alpha=0.5)
         plt.title('Konstantna obremenitev')
         plt.legend()
         plt.show()
```



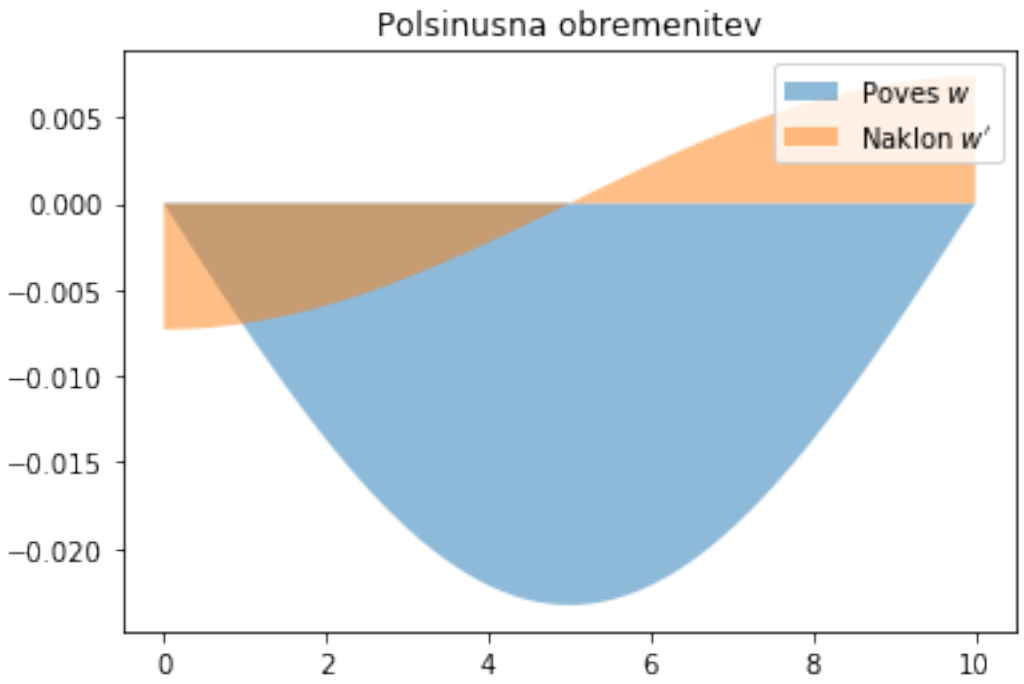
Polsinusna obremenitev

Poglemo še rešitev za polsinusno obremenitev. Prilagoditi moramo funkcijo mejnih preostankov:

```
In [31]: def r_nosilec_pol_sinusna(y0=[1., 1.], x=None, ciljne_vrednosti=np.array([0, 0.]), E=2.1e11, I=
             y_ode = odeint(f_nosilec_pol_sinusna, y0=[0, y0[0], 0, y0[1]], t=x, args=(E, I))
             r = y_ode[-1,0:3:2] - ciljne_vrednosti
             return r
         rešitev = optimize.root(r_nosilec_pol_sinusna, np.array([0., 0.]), args=(x))
         y_pol_sin = odeint(f_nosilec_pol_sinusna, y0=[0, rešitev.x[0], 0, rešitev.x[1]], t=x, args=(E,
```

Prikažemo rezultat w, w':

```
In [32]: plt.fill_between(x, y_pol_sin[:,0], label='Poves $w$', alpha=0.5)
         plt.fill_between(x, y_pol_sin[:,1], label='Naklon $w\'$', alpha=0.5)
         plt.title('Polsinusna obremenitev')
         plt.legend()
         plt.show()
```



13.3 Metoda končnih razlik

Rešujemo robni problem:

$$\ddot{y} = f(t, y, \dot{y}), \qquad y(a) = \alpha, \quad y(b) = \beta.$$

Velja torej:

$$a = t_0, \quad b = t_{n-1}.$$

Pri metodi končnih razlik za reševanje robnega problema uporabimo diferenčno shemo. Predpostavimo, da imamo interval $[a, b]$, na katerem rešujemo diferencialno enačbo (neodvisna spremenljivka) razdeljeno na enake podintervale (točk je n):

$$t = [t_0, t_1, \ldots, t_{n-1}].$$

Odvode nadomestimo s centralno diferenčno shemo:

	y_{i-2}	y_{i-1}	y_i	y_{i+1}	y_{i+2}
$y_i' = \frac{1}{h}\cdot$	0	-0.5	0	0.5	0
$y_i'' = \frac{1}{h^2}\cdot$	0	1	-2	1	0
$y_i''' = \frac{1}{h^3}\cdot$	-0.5	1	0	-1	0.5
$y_i^{(4)} = \frac{1}{h^4}\cdot$	1	-4	6	-4	1

Prikazana centralna diferenčna shema ima napako drugega reda $\mathcal{O}(h^2)$.

V i-ti točki navadno differencialno enačbo drugega reda s centralimi diferencami zapišemo:

$$\frac{1}{h^2}\left(y_{i-1} - 2\,y_i + y_{i+1}\right) + \mathcal{O}(h^2) = f\left(t_i, y_i, \frac{1}{2h}\left(-y_{i-1} + y_{i+1}\right) + \mathcal{O}(h^2)\right).$$

Če zanemarimo napako metode:

$$\frac{1}{h^2}\left(y_{i-1} - 2\,y_i + y_{i+1}\right) = f\left(t_i, y_i, \frac{1}{2h}\left(-y_{i-1} + y_{i+1}\right)\right), \qquad i = 1, 2, \ldots, n-2.$$

Zgornjo enačbo lahko zapišemo za $n-2$ notranjih točk, kar pomeni, da nam do rešljivega sistema enačb za n neznank manjkata še dve enačbi. Ti dve enačbi sta robna pogoja:

$$y_0 = \alpha, \quad y_{n-1} = \beta.$$

V primeru linearnega robnega problema moramo za izračun n neznank y_i rešiti sistem n linearnih enačb (če je pa nelinearen, pa sistem nelinearnih enačb).

Ocena napake

Točen rezultat $y(t_i)$ pri velikosti koraka h je:

$$y(t_i) = y_{i,h} + E_{i,h},$$

kjer je $y_{i,h}$ numerični približek in E_h napaka metode. Ker je globalna napaka drugega reda, lahko napako zapišemo:

$$E_{i,h} = k\,h^2,$$

Podobno lahko za velikost koraka $2h$ zapišemo:

$$y(t_j) = y_{j,2h} + E_{j,2h},$$

kjer je $y_{j,2h}$ numerični približek in E_{2h} napaka metode in je:

$$E_{j,2h} = k\,(2\,h)^2 = 4\,k\,h^2$$

Ob predpostavki, da je konstanta k pri koraku h in koraku $2h$ enaka, lahko določimo oceno napake pri boljšem približku E_h. Najprej izenačimo točna rezultat $y(t_i)$ pri koraku h in rezultat $y(t_j)$ pri koraku $2h$ (velja $i = 2\,j$, $j = 1, 2, \ldots$):

$$y_{i,h} + k\,h^2 = y_{j,2h} + 4\,k\,h^2$$

sledi:

$$3\,k\,h^2 = y_{2j,h} - y_{j,2h}$$

in nato izračunamo *oceno napake*:

$$E_{j,h} = \frac{y_{2j,h} - y_{j,2h}}{3}.$$

13.3.1 Numerični zgled: vertikalni met

Zgoraj smo vertikalni met rešili s strelsko metodo; uporabimo sedaj metodo končnih razlik. Najprej robni problem:

$$y''(t) = F_y/m - g, \qquad y(0) = 0, \quad y(1) = 10.$$

zapisati s pomočjo centralne diferenčne sheme ($F_y = -c\,y'$):

$$\frac{1}{h^2}\,(y_{i-1} - 2\,y_i + y_{i+1}) = -c\left(\frac{1}{2h}\,(-y_{i-1} + y_{i+1})\right)/m - g.$$

(Tukaj smo predpostavili, da je zračni upor linearno odvisen od hitrost. V nasprotnem primeru bi imeli nelinearni robni problem, in posledično razvili sistem nelinearnih enačb.)

Zgornji izraz preoblikujemo:

$$(2 - c\,h/m)\,y_{i-1} - 4\,y_i + (2 + c\,h/m)\,y_{i+1} = -2\,g\,h^2, \qquad i = 1, 2, \ldots, n-2.$$

Robna pogoja sta:

$$y_0 = 0, \qquad y_{n-1} = 10$$

Robni problem smo torej preoblikovali na sistem n linearnih enačb. Poglejmo si sedaj konkreten izračun za $n = 11$; najprej definirajmo konstante, časovni vektor `t` in korak `h`:

```
In [33]: n = 11 # liho število
         c = 0.5
         m = 1.0
         g = 9.81
         t = np.linspace(0, 1, n)
         h = t[1]
```

Nato nadaljujemo z izračunom tridiagonalne matrike koeficientov `A`.

Pomagamo si s funkcijo `numpy.diag()` (dokumentacija[2]):

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

s parametroma:

- `v` vektor, ki bo prirejen diagonali,
- `k` diagonala, kateri se priredi `v`. `k=0` uporabimo za glavno diagonalo, `k<0` oz. `k>0` uporabimo za diagonale pod oz. nad glavno diagonalo.

```
In [34]: A = np.diag(-4*np.ones(n), 0) + \
             (2-c*h/m)*np.diag(np.ones(n-1), -1) + \
             (2+c*h/m)*np.diag(np.ones(n-1), 1)
         A[:4,:4]

Out[34]: array([[-4.  ,  2.05,  0.  ,  0.  ],
                [ 1.95, -4.  ,  2.05,  0.  ],
                [ 0.  ,  1.95, -4.  ,  2.05],
                [ 0.  ,  0.  ,  1.95, -4.  ]])
```

Definirajmo še vektor konstant:

```
In [35]: b = -2*g*h**2 * np.ones(n)
```

Sedaj *popravimo* matriko koeficientov `A` in vektor konstant `b`, da zadostimo robnim pogojem:

$$y_0 = 0, \quad y_{n-1} = 10.$$

```
In [36]: A[0,0] = 1
         A[0,1] = 0
         A[-1,-2] = 0
         A[-1,-1] = 1

In [37]: b[0] = 0.
         b[-1] = 10.
```

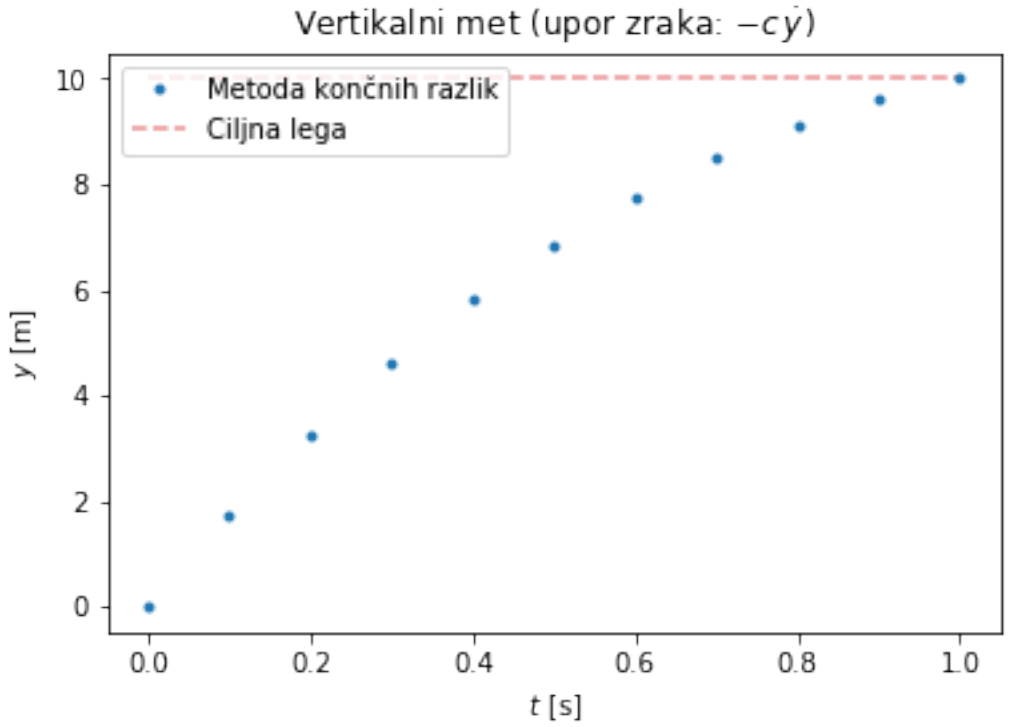
Rešimo sistem linearnih enačb:

```
In [38]: y_mkr = np.linalg.solve(A, b)
```

Prikažemo rezultat:

[2] https://docs.scipy.org/doc/numpy/reference/generated/numpy.diag.html

```
In [39]: plt.title('Vertikalni met (upor zraka: $-c\,\dot y$)')
         plt.hlines(10, 0, 1, 'C3', label='Ciljna lega', linestyles='dashed', alpha=0.5)
         plt.plot(t, y_mkr, '.', label='Metoda končnih razlik')
         plt.xlabel('$t$ [s]')
         plt.ylabel('$y$ [m]')
         plt.legend()
         plt.show()
```



S pomočjo funkcije `numpy.gradient()`[3] izračunamo še hitrost in pospešek:

```
In [40]: v_mkr = np.gradient(y_mkr,h, edge_order=2)
         a_mkr = np.gradient(v_mkr,h, edge_order=2)
         v_mkr[:4]
```

```
Out[40]: array([ 17.99109496,  16.20009044,  14.45276895,  12.79068266])
```

Izračunajmo sedaj še rezultat z dvojnim korakom:

```
In [41]: n2h = n//2+1
         t2h = np.linspace(0, 1, n2h)
         h2h = t[1]
         A2h = np.diag(-4*np.ones(n2h), 0) + \
             (2-c*h2h/(m))*np.diag(np.ones(n2h-1), -1) + \
             (2+c*h2h/(m))*np.diag(np.ones(n2h-1), 1)
```

[3] https://docs.scipy.org/doc/numpy/reference/generated/numpy.gradient.html

```
b2h = -2*g*h2h**2 * np.ones(n2h)
A2h[0,0] = 1
A2h[0,1] = 0
A2h[-1,-1] = 1
A2h[-1,-2] = 0
b2h[0] = 0.
b2h[-1] = 10.
y2h_mkr = np.linalg.solve(A2h, b2h)
```

Primerjajmo prvih šest rezultatov pri koraku $2h$:

```
In [42]: y2h_mkr[:6]

Out[42]: array([ -1.27658352e-16,   2.40584652e+00,   4.59862736e+00,
                  6.58873597e+00,   8.38605878e+00,   1.00000000e+01])
```

z vsakim drugim rezultatom pri koraku h:

```
In [43]: y_mkr[:12:2]

Out[43]: array([  4.55476113e-16,   3.24001809e+00,   5.79815462e+00,
                  7.73931207e+00,   9.12221538e+00,   1.00000000e+01])
```

Sedaj lahko ocenimo napako:

```
In [44]: (y_mkr[::2]-y2h_mkr)/3

Out[44]: array([  1.94378155e-16,   2.78057188e-01,   3.99842419e-01,
                  3.83525367e-01,   2.45385533e-01,   0.00000000e+00])
```

13.3.2 Numerični zgled: nosilec z obremenitvijo

Vrnimo se k robnemu problemu nosilca s polsinusno obremenitvijo ($q(x) = -F_0 \sin(\pi x/l)$), ki smo ga že obravnavali s strelsko metodo.

Diferencialno enačbo četrtega reda zapišemo s pomočjo centralne diferenčne sheme (za i-to točko):

$$-\frac{E\,I}{h^4}\left(1\,w_{i-2} - 4\,w_{i-1} + 6\,w_i - 4\,w_{i+1} + 1\,w_{i+2}\right) - F_0 \sin(\pi\, x_i/l) = 0.$$

Robni pogoji so štirje, najprej poves na robovih:

$$w_0 = w_{n-1} = 0$$

Ker na robovih ni momenta, je drugi odvod nič. S centralno diferenčno shemo torej zapišemo dodatne enačbe:

$$w_{-1} - 2\,w_0 + w_1 = 0 \qquad w_{n-2} - 2\,w_{n-1} + w_n = 0.$$

Če za neodvisno spremenljivko x uporabimo n ekvidistantnih točk, potem diferencialno enačbo četrtega reda zapišemo za $n-2$ notranje točke. S tem pridobimo dodatni nefizikalni točki w_{-1} in w_n. Če dodamo še štiri robne pogoje, imamo rešljiv sistem linearnih enačb z $n+2$ neznakami in $n+2$ enačbami.

Najprej pripravimo podatke, neodvisno spremenljivko `x` in korak `h`:

```
In [45]: l = 10.
         E = 2.1e11
         I = 2.1e-5
         F_0= 1e3
         n = 100
         x = np.linspace(0, l, n)
         h = x[1]
```

Nato pripravimo matriko koeficientov (matrika je dimenzije `(n+2, n+2)`:

```
In [46]: A = +1*np.diag(np.ones(n-2+2), -2) \
             -4*np.diag(np.ones(n-1+2), -1) \
             +6*np.diag(np.ones(n+2), 0)  \
             -4*np.diag(np.ones(n-1+2), +1)  \
             +1*np.diag(np.ones(n-2+2), +2)
         A[:4,:4]

Out[46]: array([[ 6., -4.,  1.,  0.],
                [-4.,  6., -4.,  1.],
                [ 1., -4.,  6., -4.],
                [ 0.,  1., -4.,  6.]])
```

Definirajmo še vektor konstant (dodamo en element na koncu in en na začetku):

```
In [47]: b = np.zeros(n+2)
         b[1:-1] = - h**4 * F_0*np.sin(np.pi*x/l)/(E*I)
         b[:4]

Out[47]: array([  0.00000000e+00,  -0.00000000e+00,  -7.48966545e-10,
                 -1.49717894e-09])
```

Sedaj *popravimo* matriko koeficientov `A` in vektor konstant `b`, da zadostimo robnim pogojem.

Najprej $w_0 = w_{n-1} = 0$:

```
In [48]: A[0,:3]= np.array([0, 1, 0])
         b[0] = 0
         A[-1,-3:]= np.array([0, 1, 0])
         b[-1] = 0
```

Nato $w_{-1} - 2\,w_0 + w_1 = 0$ in $w_{n-2} - 2\,w_{n-1} + w_n = 0$:

```
In [49]: A[1,:4] = np.array([1, -2, 1,0])
         b[1] = 0
         A[-2,-4:] = np.array([0, 1, -2, 1])
         b[-2] = 0

In [50]: A[:5,:5]
```

```
Out[50]: array([[ 0.,  1.,  0.,  0.,  0.],
                [ 1., -2.,  1.,  0.,  0.],
                [ 1., -4.,  6., -4.,  1.],
                [ 0.,  1., -4.,  6., -4.],
                [ 0.,  0.,  1., -4.,  6.]])
```

```
In [51]: A[-5:,-5:]
```

```
Out[51]: array([[ 6., -4.,  1.,  0.,  0.],
                [-4.,  6., -4.,  1.,  0.],
                [ 1., -4.,  6., -4.,  1.],
                [ 0.,  0.,  1., -2.,  1.],
                [ 0.,  0.,  0.,  1.,  0.]])
```

```
In [52]: b[:5]
```

```
Out[52]: array([  0.00000000e+00,   0.00000000e+00,  -7.48966545e-10,
                 -1.49717894e-09,  -2.24388381e-09])
```

```
In [53]: b[-5:]
```

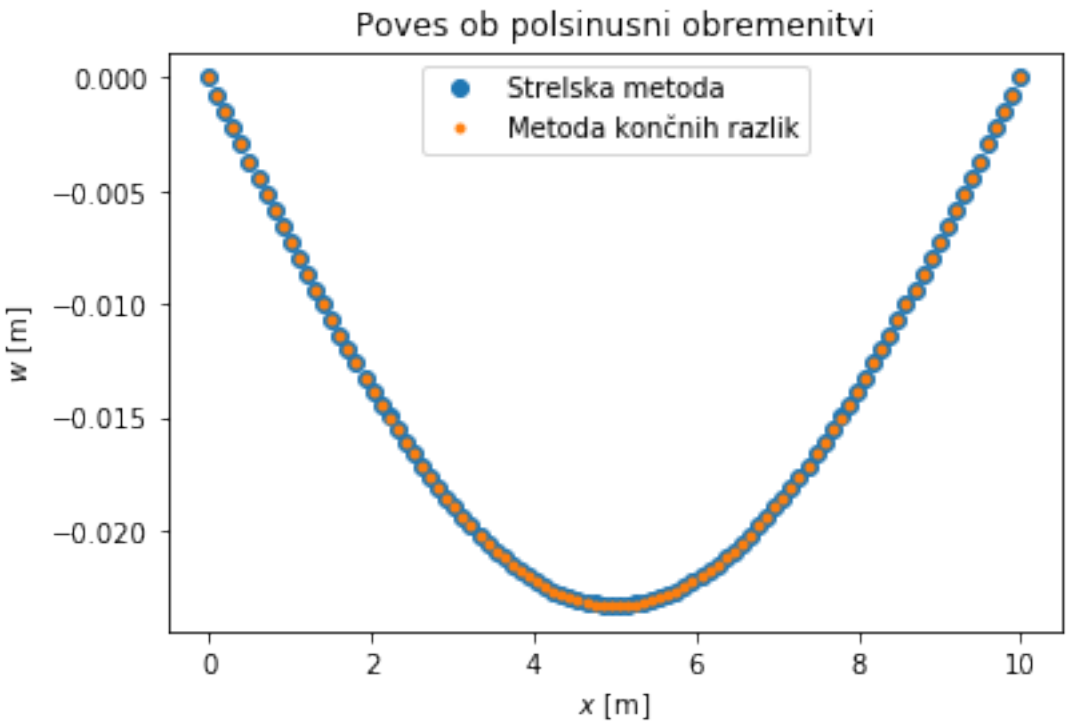
```
Out[53]: array([ -2.24388381e-09,  -1.49717894e-09,  -7.48966545e-10,
                  0.00000000e+00,   0.00000000e+00])
```

Rešimo sistem linearnih enačb:

```
In [54]: y_mkr = np.linalg.solve(A, b)
```

Prikažemo rezultat:

```
In [55]: plt.title('Poves ob polsinusni obremenitvi')
         plt.plot(x, y_pol_sin[:,0], 'o', label='Strelska metoda')
         plt.plot(x, y_mkr[1:-1], '.', label='Metoda končnih razlik')
         plt.xlabel('$x$ [m]')
         plt.ylabel('$w$ [m]')
         plt.legend()
         plt.show()
```



13.4 Dodatno: simbolna rešitev nosilca

Tukaj si bomo pogledali simbolno reševanje robnega problema. Poudariti je treba, da gre tukaj zgolj za zgled, ki ga lahko naredimo za obravnavani nosilec z relativno enostavno polsinusno obremenitvijo. **V praksi so seveda obremenitve in tudi oblike nosilca lahko bistveno bolj zahtevne in takrat druge poti kot numeričnega reševanja skoraj nimamo na voljo.**

Najprej uvozimo `sympy`:

```
In [56]: import sympy as sym
         sym.init_printing()
```

Definirajmo spremenljivke:

```
In [57]: w, x, E, I, F_0, l = sym.symbols('w, x, E, I, F_0, l')
```

Definirajmo differencialno enačbo (robne pogoje dodamo pozneje):

```
In [58]: eq = sym.Eq(-E*I*w(x).diff(x,4)-F_0*sym.sin(sym.pi*x/l))
         eq
```

```
Out[58]:
```

$$-EI\frac{d^4}{dx^4}w(x) - F_0 \sin\left(\frac{\pi x}{l}\right) = 0$$

Rešimo:

```
In [59]: sol = sym.dsolve(eq).args[1]
         sol
```

Out[59]:

$$C_1 + C_2 x + C_3 x^2 + C_4 x^3 - \frac{F_0 l^4}{\pi^4 EI} \sin\left(\frac{\pi x}{l}\right)$$

Iz vrednosti rešitve pri $x = 0$:

```
In [60]: sol.subs(x, 0)
```

Out[60]:

$$C_1$$

in drugega odvoda rešitve pri $x = 0$:

```
In [61]: sol.diff(x, 2).subs(x, 0)
```

Out[61]:

$$2C_3$$

določimo konstanti `C1` in `C3` (pomik in moment na levi strani sta enaka nič):

```
In [62]: c1c3 = {'C1': 0, 'C3': 0}
```

Robni pogoji na desni strani ($x = l$):

```
In [63]: eq1 = sol.subs(c1c3).subs(x, l)
         eq1
```

Out[63]:

$$C_2 l + C_4 l^3$$

Ker na desni strani ni pomika, sledi rešitev za konstanto `C2`:

```
In [64]: C2 = sym.solve(eq1, 'C2')[0]
         C2
```

Out[64]:

$$-C_4 l^2$$

Ker ni momenta, sledi še:

```
In [65]: eq2 = sol.diff(x, 2).subs(c1c3).subs(x, l)
         eq2
```

Out[65]:

$$6C_4 l$$

Določimo konstanto `C4`:

```
In [66]: C4 = sym.solve(
               eq2.subs('C2', C2),
               'C4'
             )[0]
         C4
```

Out[66]:

$$0$$

Upoštevajoč izračunane konstante je rešitev:

```
In [67]: resitev = sol.subs(c1c3).subs('C2', C2).subs('C4', C4)
         resitev
```

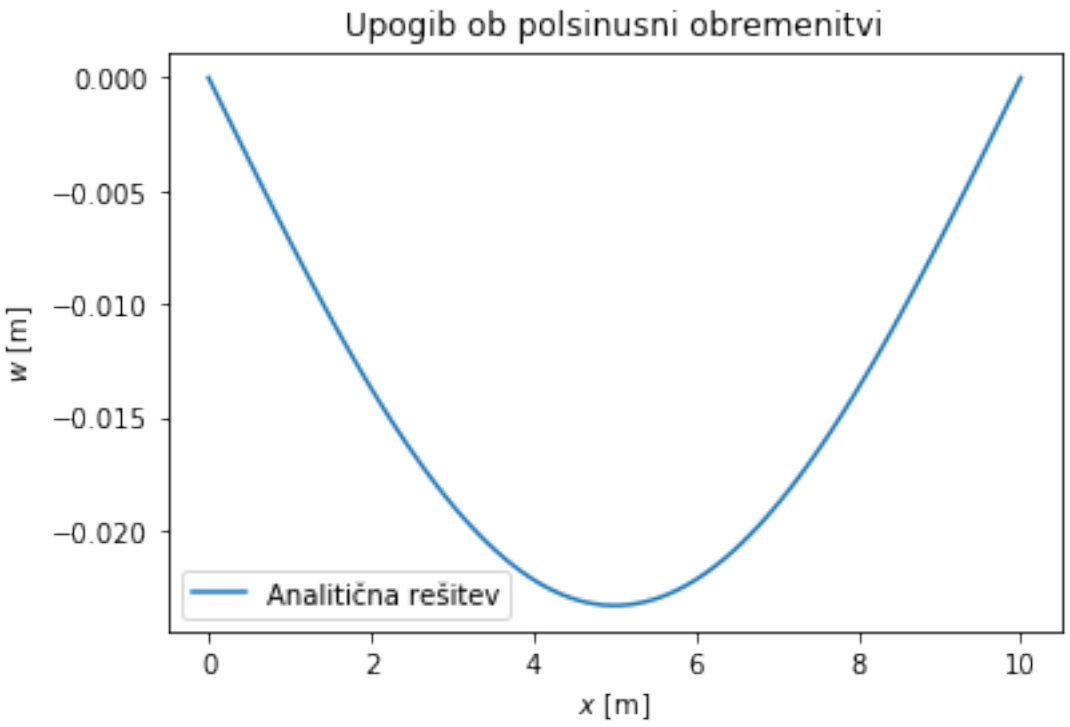
Out[67]:

$$-\frac{F_0 l^4}{\pi^4 EI} \sin\left(\frac{\pi x}{l}\right)$$

Pripravimo še numerično funkcijo `resitev_np()`:

```
In [68]: podatki = {E: 2.1e11, I: 2.1e-5, l: 10, F_0:1e3}
         resitev_np = sym.lambdify(x, resitev.subs(podatki), modules='numpy')
         x_ana = np.linspace(0, podatki[l], 100)
         y_ana = resitev_np(x_ana)
```

```
In [69]: plt.title('Upogib ob polsinusni obremenitvi')
         plt.plot(x_ana, y_ana, '-', label='Analitična rešitev')
         plt.xlabel('$x$ [m]')
         plt.ylabel('$w$ [m]')
         plt.legend()
         plt.show()
```



Primerjajmo sedaj analitično rešitev, z rešitivijo z metodo končnih razlik in strelsko metodo:

```
In [70]: [np.min(y_ana), np.min(y_mkr), np.min(y_pol_sin)]
```

Out[70]:

$$[-0.0232759411601, \quad -0.0232798479985, \quad -0.0232759509119]$$

Poglavje 14

Testiranje pravilnosti kode, uporabniški vmesnik

14.1 Testiranje pravilnosti kode

Kadar razvijamo bolj obsežno kodo in pri tem sodeluje več oseb, se pojavi potreba po avtomatskem testiranju pravilnosti kode. To pomeni, da poleg kode, ki ima določen rezultat, definiramo tudi testirne funkcije; slednje preverjajo ali prve vračajo pričakovani rezultat.

Preprosti primer je:

```
# vsebina datoteke test_prvi.py (mapa moduli)
def kvadrat(x):
    return x**2 + 1 # očitna napaka glede na ime funkcije

def test_kvadrat():
    assert kvadrat(2) == 4
```

Če poženemo funkcijo `test_kvadrat()` se testira pravilnost `kvadrat(2) == 4`; v primeru, da je rezultat `True` se ne zgodi nič, če pa je rezultat `False` se sproži `AssertionError()` (glejte dokumentacijo[1] za ukaz `assert`). Ker funkcija `kvadrat()` vrne `x**2+1`, bo testiranje neuspešno.

Verjetno se sprašujete v čem je smisel takega *testiranja pravilnosti*. Ko koda postaja obsežna in na njej dela veliko oseb, postane nujno tudi prepletena. Tako se lahko zgodi, da razvijalec doda novo funkcionalnost in nehote poruši obstoječo. Če so vse funkcionalnosti testirane, bo testiranje zaznalo neustreznost predlagane spremembe.

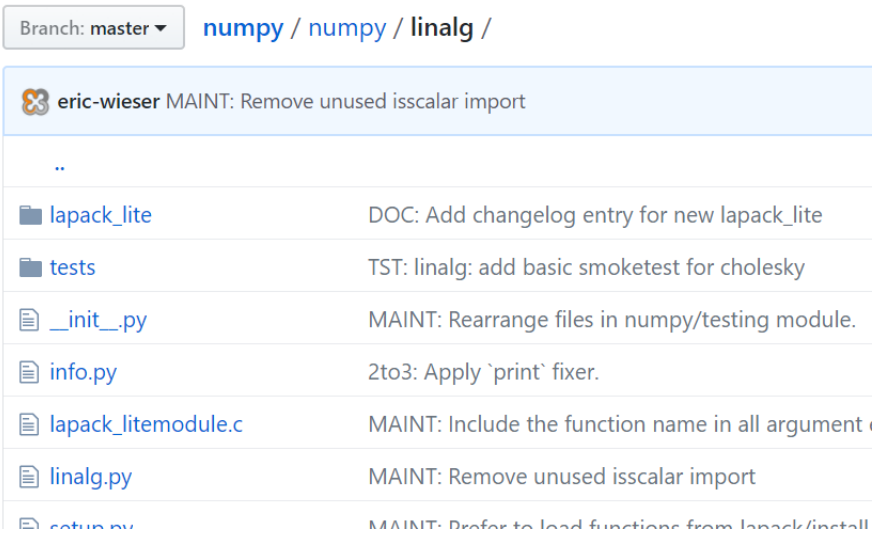
Večina paketov ima tako dodano testiranje pravilnosti; za primer si lahko pogledate izvorno kodo paketa *NumPy*, ki se nahaja na githubu[2]; če pogledate vsebino[3] podmodula `numpy.linalg` za linearno algebro:

opazimo podmapo `tests`. Slednja je v celoti namenjena testiranju in vsebuje množico Python datotek, ki preverjajo ustreznost podmodula.

Obstaja več možnosti testiranja kode, pogosto se uporabljajo sledeče:

- `unittest`[4] (je vgrajen v Python),

[1] https://docs.python.org/reference/simple_stmts.html#assert
[2] https://github.com/numpy/numpy/tree/master/numpy/
[3] https://github.com/numpy/numpy/tree/master/numpy/linalg
[4] https://docs.python.org/library/unittest.html



- `nose`[5],
- `pytest`[6].

Vsi trije pristopi imajo podobno funkcionalnost. V zadnjem obdobju pa se najpogosteje uporablja `pytest` (npr. NumPy/SciPy), katerega osnove si bomo tukaj pogledali.

14.1.1 pytest

Modul `pytest` je vključen v Anaconda distribucijo Pythona, sicer pa ga namestimo z ukazom `pip` ali `conda`.

Nekatere lastnosti:

- vrne opis testa, ki ni bil uspešen,
- samodejno iskanje testnih modulov in funkcij,
- lahko vključuje `unittest` in `nose` teste.

Če v ukazni vrstici poženete `pytest`, bo program sam poiskal trenutno mapo in vse podmape za datoteke oblike `test_*.py` ali `*_test.py` (napredno iskanje je navedeno v dokumentaciji[7]).

Če v ukazni vrstici poženemo ukaz:

```
pytest
```

bomo dobili tako poročilo:

```
============================= test session starts ==============================
platform win32 -- Python 3.6.2, pytest-3.2.1, py-1.4.34, pluggy-0.4.0
rootdir: c:\pypinm\moduli, inifile:
collected 4 items
```

[5] http://nose.readthedocs.io/en/latest/
[6] https://docs.pytest.org/
[7] https://docs.pytest.org/en/latest/goodpractices.html#test-discovery

```
test_orodja.py ...
test_prvi.py F

================================== FAILURES ===================================
_______________________________ test_kvadrat ________________________________

    def test_kvadrat():
>       assert kvadrat(2) == 4
E       assert 5 == 4
E        +  where 5 = kvadrat(2)

test_prvi.py:5: AssertionError
==================== 1 failed, 3 passed in 0.65 seconds =====================
```

Iz poročila vidimo, da je program našel dve datoteki (`test_orodja.py` in `test_prvi.py`) in da je prišlo do napake pri eni funkciji (`test_kvadrat`), tri funkcije pa so uspešno prestale test.

Za osnovno uporabo pri numeričnih izračunih je treba še izpostaviti modul `numpy.testing` (dokumentacija[8]), ki nudi podporo za testiranje numeričnih polj.

Izbrane funkcije so:

- `assert_allclose(dejansko, pričakovano[, rtol, ...])` (dokumentacija[9]) preveri enakost do zahtevane natančnosti,
- `assert_array_less(x, y[, err_msg, verbose])` (dokumentacija[10]) preveri, ali so elementi `x` manjši od elementov `y`,
- `assert_string_equal(dejansko, pričakovano)` (dokumentacija[11]) preveri enakost niza.

Zgled: `test_orodja.py`**:**

V datoteki `test_orodja.py` so pripravljene funkcije za testiranje modula `orodja.py`.

Tako so najprej pripravljeni podatki:

```
zacetna = np.asarray([[1, 2, 3],
                      [4, 5, 6],
                      [7, 8, 9]])
zamenjana_0_1_stolpca = np.asarray([[2, 1, 3],
                                    [5, 4, 6],
                                    [8, 7, 9]])
```

Potem je definirana funkcija, ki kliče funkcijo `orodja.zamenjaj_stolpca()` in rezultat primerja s pričakovanim `zamenjana_0_1_stolpca`:

```
def test_stolpec():
    a = zacetna.copy() # naredimo kopijo podatkov
    b = orodja.zamenjaj_stolpca(a, 0, 1) # b (in tudi a) imata zamenjane stolpce
    np.testing.assert_allclose(b, zamenjana_0_1_stolpca)
```

[8] http://docs.scipy.org/doc/numpy/reference/routines.testing.html#

[9] https://docs.scipy.org/doc/numpy/reference/generated/numpy.testing.assert_allclose.html#numpy.testing.assert_allclose

[10] https://docs.scipy.org/doc/numpy/reference/generated/numpy.testing.assert_array_less.html#numpy.testing.assert_array_less

[11] https://docs.scipy.org/doc/numpy/reference/generated/numpy.testing.assert_string_equal.html#numpy.testing.assert_string_equal

Če funkcija `orodja.zamenjaj_stolpca()` deluje pravilno, se klicanje `pytest` v ukazni vrstici uspešno zaključi. Za preostale teste poglejte datoteko `test_orodja.py` in dokumentacijo[12] paketa `pytest`.

14.2 Uporabniški vmesnik

Tudi za programiranje uporabniškega vmesnika obstaja veliko različnih načinov/modulov. Nekaj najpogosteje uporabljenih:

- PyQt[13] temelji na Qt[14], ki predstavlja najbolj razširjeno platforma za uporabniške vmesnike,
- PySide[15] podobno kakor *PyQt*, vendar s širšo licenco LGPL[16], a žal tudi slabšo podporo (tukaj je dobra knjiga: PySide GUI Application Development - SE[17]),
- Kivy[18] za hiter razvoj modernih uporabniških vmesnikov (ni tako zrel kakor npr. Qt),
- wxWidgets[19] široko uporabljena prosta platforma za izdelavo uporabniških vmesnikov.

Med vsemi naštetimi si bomo podrobneje pogledali *PyQt*, ki je verjetno najboljša in tudi najbolj dozorela izbira (za komercialno uporabo pa žal ni brezplačna).

Velja omeniti, da uporabniški vmesnik lahko:

- *rišemo* s QtDesignerjem[20] (glejte video[21]), ali
- *kodiramo*.

Tukaj bomo uporabniški vmesnik *kodirali*.

14.2.1 Zgled

Najprej uvozimo paket za interakcijo in z grafičnimi objekti:

```
In [1]: from PyQt5 import QtCore     # za interakcijo
        from PyQt5 import QtWidgets  # grafični objekti
```

Potrebovali bomo tudi modul `sys` za poganjanje programa:

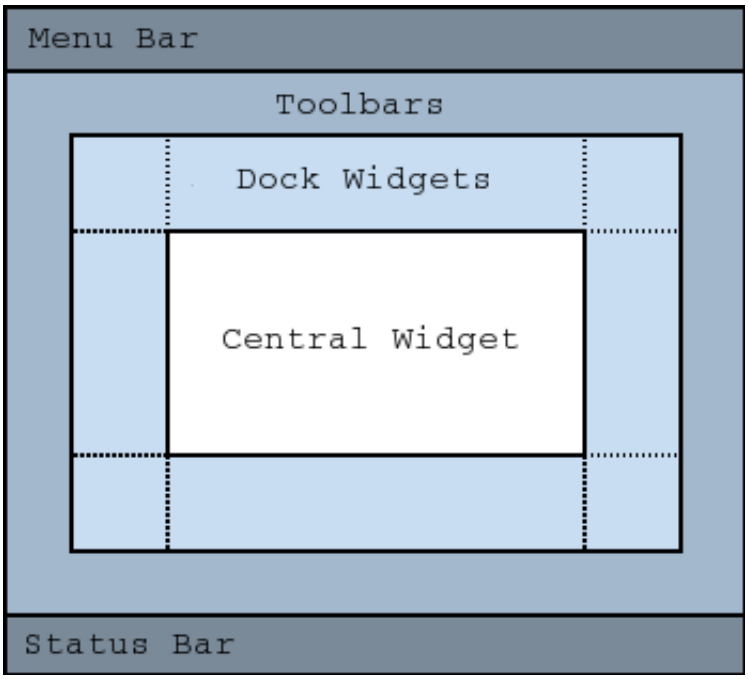
```
In [2]: import sys
```

Uporabniški vmesnik tipično gradimo na razredu `QtWidgets.QMainWindow` (dokumentacija[22]), ki ima spodaj prikazano strukturo:

Bistveni grafični elementi, ki jih pri takem uporabniškem vmesniku uporabimo, so:

- *Menu Bar*,

[12] https://docs.pytest.org/
[13] https://riverbankcomputing.com/software/pyqt/intro
[14] https://www.qt.io/
[15] https://wiki.qt.io/PySide
[16] http://qt-project.org/wiki/PySide
[17] https://www.packtpub.com/application-development/pyside-gui-application-development-second-edition
[18] http://kivy.org/
[19] https://www.wxwidgets.org/
[20] http://doc.qt.io/qt-5/qtdesigner-manual.html
[21] https://www.youtube.com/watch?v=GLqrzLIIW2E
[22] http://doc.qt.io/qt-5/qmainwindow.html



- *Toolbars*,
- *Dock Widgets*,
- *Central Widget*,
- *Status bar*.

Ni potrebno definirati vseh; spodaj si bomo pogledali primer, ko bomo definirali *Status bar*, *Central Widget* in *Menu Bar*. Ponavadi vse elemente definiramo pri inicializaciji instance razreda (metoda `__init__`).

Tukaj izpostavimo, da grafični vmesnik temelji na t. i. *Widgetih*, (za na primer gumb, tabelo, datum itd.). Prikaz nekaterih možnosti je prikazan v:

- Qt 5 Windows galeriji[23],
- Qt5 galeriji[24] in
- Qt5 fusion galeriji[25].

Zelo enostaven uporabniški vmesnik (z vrstičnimi komentarji kode) je prikazan spodaj.

```
In [3]: import sys
        from PyQt5 import QtCore
        from PyQt5 import QtGui
        from PyQt5 import QtWidgets

        class GlavnOkno(QtWidgets.QMainWindow):
            """ Glavno okno podeduje `QtWidgets.QMainWindow`
            """
```

[23] http://doc.qt.io/qt-5/gallery-windows.html
[24] http://doc.qt.io/qt-5/gallery.html
[25] http://doc.qt.io/qt-5/gallery-fusion.html

```
            def __init__(self):
                """ Konstruktor GlavnoOkno objekta
                """
                QtWidgets.QMainWindow.__init__(self) # konstruktor starša
                self.setWindowTitle('Naslovno okno programa')

                # status bar
                self.moj_status_bar = QtWidgets.QStatusBar() # novi widget
                self.moj_status_bar.showMessage('Ta tekst po 10s izgine', 10000)
                self.setStatusBar(self.moj_status_bar) # tukaj se self.moj_status_bar priredi predpripr

                # central widget
                self.gumb1 = QtWidgets.QPushButton('Gumb 1') # Gumb z napisom 'Gumb 1'
                self.gumb1.pressed.connect(self.akcija_pri_pritisku_gumba1) # povežemo pritisk s funkci
                self.setCentralWidget(self.gumb1) # dodamo gumb v `CentralWidget`

                # menuji
                self.moj_izhod_akcija = QtWidgets.QAction('&Izhod',
                                                          self, shortcut=QtGui.QKeySequence.Close,
                                                          statusTip="Izhod iz programa",
                                                          triggered=self.close)
                self.moj_datoteka_menu = self.menuBar().addMenu('&Datoteka') # menu 'Datoteka'
                self.moj_datoteka_menu.addAction(self.moj_izhod_akcija) # povezava do zgoraj definirane

            def akcija_pri_pritisku_gumba1(self):
                QtWidgets.QMessageBox.about(self,
                                            'Naslov :)',
                                            'Tole se sproži pri pritisku Datoteka.')
```

Program potem poženemo znotraj stavka `try`:

```
In [4]: try:
            app = QtWidgets.QApplication(sys.argv)
            mainWindow = GlavnOkno()
            mainWindow.show()
            app.exec_()
            sys.exit(0)
        except SystemExit:
            print('Zapiram okno.')
        except:
            print(sys.exc_info())
```

```
Zapiram okno.
```

Za naprednejši zgled glejte datoteki:

- preprosti_uporabniski_vmesnik.py[26],
- uporabniski_vmesnik.py[27] in

[26] ./moduli/preprosti_uporabniski_vmesnik.py
[27] ./moduli/uporabniski_vmesnik.py

- brskalnik.py[28].

Nekaj komentarjev na `uporabniski_vmesnik.py`:

1. Poglejte prepis dogodka `mouseDoubleClickEvent` in prepišite podedovan dogodek `keyPressEvent`, ki naj ob pritisku katerekoli tipke zapre program (če se nahajate v TextEdit polju, potem seveda pritisk tipke izpiše vrednost te tipke).
2. Dodajte še kakšen *Widget* s seznama[29].
3. Spremenite program, da se bo vedno izrisovala funkcija sinus, v vpisno polje `function_text` pa boste zapisali število diskretnih točk (sedaj je točk 100). Povežite polje z ustreznimi funkcijami.
4. Uredite lovljenje napak pri zgornji spremembi.

14.3 Nekaj vprašanj za razmislek!

1. V PyCharm-u pripravite modul, ki bo imel dve funkciji:
   - za množenje matrike in vektorja,
   - za množenje dveh matrik.
2. Za modul zgoraj pripravite skripto za testiranje (uporabite `numpy.testing`).
3. V `uporabniski_vmesnik_simple.py` inicializijski metodi `__init__` zakomentirajte vse klice na metode `self.init ...` razen na metodo: `self.init_status_bar()`. Poženite program v navadnem načinu. Nastavite *break* točko na `self.setGeometry(50, 50, 600, 400)` in poženite program v *debug* načinu.
4. Nadaljujte prejšnjo točko in poiščite bližnjico za pomikanje po vrsticah:
   - s preskokom vrstice,
   - z vstopom v vrstico.

   Vstopite v `init_status_bar(self)` in se ustavite pri vrstici `self.setStatusBar(self.status_bar)`. Odprite konzolo (*console*) in prek ukazne vrstice spremenite vrednost `self.status_bar.showMessage()`.
5. Odkomentirajte prej (zgoraj) zakomentirane vrstice. Dodajte tretji gumb, ki naj program zapre.
6. Dodajte še kakšen *Widget* s seznama[30].

[28] `./moduli/brskalnik.py`
[29] `http://doc.qt.io/qt-5/gallery-windows.html`
[30] `http://doc.qt.io/qt-5/gallery-windows.html`

Literatura

[1] J. Demšar, *Python za programerje, druga dopolnjena izdaja*, Fakulteta za računalništvo in informatiko, Univerza v Ljubljani, 2012.

[2] J. Petrišič, *Matlab za inženirje*, Fakulteta za strojništvo, Univerza v Ljubljani, 2013.

[3] J. Petrišič, *Reševanje enačb*, Fakulteta za strojništvo, Univerza v Ljubljani, 1996.

[4] J. Kiusalaas, *Numerical methods in engineering with Python 3*, Cambridge university press, 2013.

[5] R. L. Burden in J. D. Faires, *Numerical analysis*, Cengage Learning, 2011.

[6] A. Quarteroni, R. Sacco in F. Saleri, *Numerical mathematics*, **37**, Springer Science & Business Media, 2010.

[7] J. Slavič, *Dinamika, mehanska nihanja in mehanika tekočin*, Fakulteta za strojništvo, Univerza v Ljubljani, 2014.

Naslov dela:
Programiranje in numerične metode v ekosistemu Pythona,

Avtor:
dr. Janko Slavič

Obseg:
XI+321 strani

Izdala in založila:
Univerza v Ljubljani
Fakulteta za strojništvo

Ljubljana, 2017
ISBN 978-961-6980-45-6