

Numerične metode v ekosistemu Pythona

<https://jankoslavic.github.io/pynm/>

Janko Slavič

28. oktober 2026

Kazalo

0	Študijski in izpitni red: 2026/27	1
0.1	Predavanja / vaje	1
0.2	Sprotni študij	1
0.3	Izpit	7
0.4	Literatura	7
0.5	Pogoji za poslušanje v naprej?	7
1	Uvod v Python	8
1.1	Kaj je ekosistem Pythona?	8
1.2	Kdo se uči Python, zakaj ga uporabljati?	10
1.3	Uporaba Pythona?	11
1.4	Python vs Matlab	11
1.5	Namestitev Pythona	11
1.6	Jupyter notebook (Jupyter lab)	12
1.7	Prvi program in PEP	15
1.8	Osnove Pythona	17
1.9	Za konec: plonk list :)	30
2	Funkcije, izjeme, moduli	31
2.1	Funkcije	31
2.2	Obravnavanje izjem	39
2.3	Osnove modulov	40
2.4	Uporaba modulov iz spleta	43
2.5	Dodatno	44
2.6	Oblikovanje datuma in časa	44
3	Moduli, numpy, matplotlib	47
3.1	Osnove modula <code>numpy</code>	47
3.2	Osnove modula <code>matplotlib</code>	59
3.3	Dodatno:	65
3.4	<code>Numba</code>	65
3.5	<code>matplotlib</code> : animacije, povratni klic, XKCD stil	65
3.6	Za najbolj zagrete	65
4	Objektno programiranje, simbolno računanje	66
4.1	Objektno programiranje	66
4.2	Simbolno računanje s <code>SymPy</code>	70
4.3	Dodatno	89
4.4	<code>sympy.mechanics</code>	89

5	Uvod v numerične metode in sistemi linearnih enačb	90
5.1	Uvod v numerične metode	90
5.2	Uvod v sisteme linearnih enačb	91
5.3	Gaussova eliminacija	97
5.4	Dodatno	100
5.5	Primer simbolnega reševanja sistema linearnih enačb v okviru <code>sympy</code>	100
6	Sistemi linearnih enačb - nadaljevanje	101
6.1	Razcep LU	101
6.2	Pivotiranje	104
6.3	Modul <code>SciPy</code>	107
6.4	Računanje inverzne matrike	109
6.5	Reševanje predoločenih sistemov	110
6.6	Iterativne metode	112
7	Interpolacija	115
7.1	Uvod	115
7.2	Interpolacija s polinomom	116
7.3	Lagrangeva metoda	117
7.4	Kubični zlepci	122
7.5	Dodatno	126
7.6	Nekaj komentarjev modula <code>scipy.interpolate</code>	126
8	Aproksimacija	129
8.1	Uvod	129
8.2	Metoda najmanjših kvadratov za linearno funkcijo	130
8.3	Metoda najmanjših kvadratov za poljubni polinom	133
8.4	Aproksimacija s poljubno funkcijo	137
8.5	Dodatno	139
9	Reševanje enačb	142
9.1	Uvod	142
9.2	Inkrementalna metoda	144
9.3	Iterativna inkrementalna metoda	146
9.4	Bisekcijska metoda	148
9.5	Sekantna metoda	151
9.6	Newtonova metoda	154
9.7	Reševanje sistemov nelinearnih enačb	157
9.8	Dodatno	160
10	Numerično odvajanje	162
10.1	Uvod	162
10.2	Aproksimacija prvega odvoda po metodi končnih razlik	163
10.3	Centralna diferenčna shema	165
10.4	Necentralna diferenčna shema	172
10.5	Uporaba <code>numpy.gradient</code>	173
10.6	Zaokrožitvena napaka pri numeričnem odvajanju	175
10.7	Aproksimacija odvoda s kompleksnim korakom	179
10.8	Dodatno	181

11 Numerično integriranje	182
11.1 Uvod	182
11.2 Newton-Cotesov pristop	184
11.3 Gaussov integracijski pristop	199
11.4 <code>scipy.integrate</code>	203
12 Numerično reševanje diferencialnih enačb - začetni problem	207
12.1 Uvod	207
12.2 Eulerjeva metoda	207
12.3 Metoda Runge-Kutta drugega reda	213
12.4 Metoda Runge-Kutta četrtega reda	214
12.5 Uporaba <code>scipy</code> za reševanje navadnih diferencialnih enačb	217
12.6 Sistem navadnih diferencialnih enačb	220
12.7 Stabilnost reševanja diferencialnih enačb*	226
12.8 Dodatno: Primer Van der Polovega nihala	229
12.9 Dodatno: simbolno reševanje diferencialne enačbe drugega reda	230
13 Numerično reševanje diferencialnih enačb - robni problem	235
13.1 Reševanje dvotočkovnih robnih problemov	235
13.2 Streška metoda	235
13.3 Metoda končnih razlik	245
13.4 Dodatno: simbolna rešitev nosilca	251
14 Testiranje pravilnosti kode, uporabniški vmesnik	254
14.1 Testiranje pravilnosti kode	254
14.2 Grafični uporabniški vmesnik	256
14.3 Nekaj vprašanj za razmislek!	259
15 Literatura	260

Poglavje 0

Študijski in izpitni red: 2026/27

0.1 Predavanja / vaje

Predavanja:

- Ponedeljek 8:15 V/2: Prof. dr. Janko Slavič (DS-P3, 01 4771 226, janko.slavic@fs.uni-lj.si)

Vaje:

Ura	as. dr. Domen Gorjup	as. Marko Zupan	as. Gal Stanovnik
Četrtek			
8:00	Č1 (I/4b)	Č2 (RU SP-24)	
10:00	Č3 (I/4b)	Č4 (RU SP-24)	
Petek			
8:00		P1 (I/4b)	P2 (RU SP-24)
12:00	P3 (I/4b)		P4 (RU SP-24)

Govorilne ure:

- četrtek 11:30 - 12:00 v učilnici I/4b, po predhodni najavi asistentu.

Asistenti:

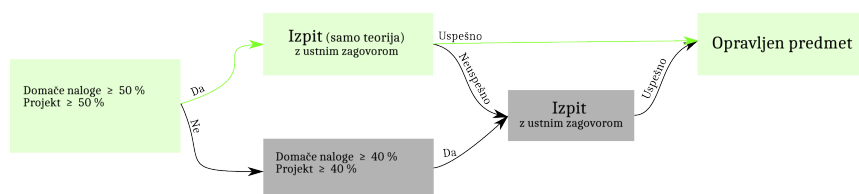
- as. dr. Domen Gorjup (DS-P5, 01 4771 228, domen.gorjup@fs.uni-lj.si)
- as. Marko Zupan (DS-P5, 01 4771 230, marko.zupan@fs.uni-lj.si)
- as. Gal Stanovnik (DS-P5, 01 4771 230, gal.stanovnik@fs.uni-lj.si)

Spremljanje sprotnega študija, obvestila itd

Sproten študij bomo izvajali s pomočjo: moj.ladisk.si. Za prijavo uporabite email, ki ga imate v VIS-u: [video prikaz prijave](#).

Vsebine, povezane s predmetom, najdete tudi v [e-učilnici](#).

0.2 Sprotni študij



0.2.1 Sestava ocene

- 5% sodelovanje pri predavanjih
- 10% tedenske domače naloge (vsaj 50%)
- 40% individualni projekt (vsaj 50%)
- 45% preizkus iz teorije (vsaj 50%): [glejte pretekle preizkuse](#)

Na predavanja in vaje morate priti pripravljeni; uporabili bomo nekatere principe obratnega učenja!

0.2.2 Predloge za vaje

Za vsako vajo je pripravljena **predloga** (Jupyter notebook) z definicijami nalog in pripravljeno kodo. Predlogo pred vajami prenesite in jo na vajah dopolnite z lastnimi zapiski in rešitvami.

Predloge vaj najdete v mapi [Predloge vaj](#).

0.2.3 Individualni projekt

Se navezuje na poljubno vsebino katerega-koli predmeta na FS in ga študent definira sam. Projekt mora vsebovati sledeče vsebine:

- *simbolno reševanje,*
- *sistemi linearnih enačb,*
- *interpolacija ali aproksimacija,*
- *iskanje ničel,*
- *integriranje ali odvajanje,*
- *reševanje diferencialnih enačb.*

0.2.4 Raba umetne inteligence pri projektu

Pri individualnem projektu **pričakujemo, da uporabljate orodja umetne inteligence** (UI; npr. Claude, ChatGPT, Copilot): za razlago metod, pisanje in pregled kode, iskanje napak in oblikovanje poročila. Z UI lahko v razpoložljivem času naredite bistveno več kot brez nje, zato so temu prilagojena tudi pričakovanja glede obsega projekta.

Pričakovan obseg projekta

Za oceno 10 pričakujemo projekt, pri katerem bi **vloženi čas brez UI bistveno presegel obseg tega predmeta**. To je merilo obsega in kakovosti izdelka, ne časa, ki ga projektu posvetite: z UI je tak projekt izvedljiv v okviru predvidene obremenitve pri predmetu.

Tak projekt ima običajno:

- **realen inženirski problem** (npr. iz drugega predmeta, s prakse ali iz laboratorija), ne šolske naloge, izmišljene zato, da se uporabijo predpisane metode,
- numerične vsebine, ki so **med seboj povezane** v celoto (npr. interpolirane meritve so vhod v diferencialno enačbo, rešitev se nato primerja z meritvami),
- **analizo vpliva parametrov** ali primerjavo več metod z oceno napake in konvergence,
- **preverjene rezultate** (analitična rešitev, druga metoda, meritve, literatura),
- uporaben programski izdelek: modul s testi, uporabniški vmesnik, animacijo, spletno aplikacijo, zajem podatkov ipd.

Okvirna ocena projekta glede na obseg:

Ocena	Pričakovan obseg
6–7	Vseh šest numeričnih vsebin je pravilno uporabljanih in pojasnjenih, a so med seboj slabo povezane; problem je preprost.
8	Smiselna celota, rezultati so preverjeni, nekaj analize vpliva parametrov.
9	Realen problem, povezane metode, primerjava metod in ocena napak, izdelan programski del (modul, testi).
10	Kot pri 9, v obsegu, pri katerem bi vloženi čas brez UI bistveno presežal obseg tega predmeta: npr. realni podatki ali meritve, več povezanih modelov, uporabniški vmesnik ali aplikacija, poglobljena kritična analiza.

Vaša odgovornost pri rabi UI

UI je orodje; avtor projekta in odgovorna oseba ste vi.

- **Vse, kar oddate, morate razumeti in znati pojasniti:** vsako metodo, vsak del kode in vsak rezultat. Na zagovoru vas lahko vprašamo o kateremkoli delu projekta.
- **Rezultate preverite.** UI pogosto predlaga neustrezno metodo, napačen robni pogoj ali kodo, ki se izvede, a računa narobe. Preverjanje rezultatov je del ocene.
- **Rabo UI opišite** v razdelku *Raba umetne inteligence* na koncu poročila (5–10 vrstic): katera orodja ste uporabili in za kaj, kje se je UI zmotila in kako ste to ugotovili ter kaj ste naredili sami.
- **Plagiatorstvo** preverjamo tudi naprej (glejte spodaj). Da je kodo napisala UI, ni opravičilo za ujemanje z drugim projektom; če izhajate iz lastnega problema, do ujemanja ne pride.
- Pri **izpitu** raba UI ni dovoljena. Zato je smiselno, da vam UI pri projektu pomaga tudi pri razumevanju, ne le pri pisanju kode.

0.2.5 Kaj ocenjujemo pri individualnem projektu

Namen projekta je prikazati **ustrezno izbiro in uporabo numeričnih metod, komentirati njihovo delovanje in ovrednotiti dobljene rezultate**. Pri ocenjevanju gledamo štiri področja:

- **Numerične metode: 45 %**
 - ustreznost in pravilnost uporabljenih metod: 30 %,
 - argumentacija in komentarji 15 %.
- **Poročilo in koda: 25 %.**
- **Lasten odnos in kreativni dodatek: 25 %.**
- **Testi kode in/ali uporabniški vmesnik: 5 %.**

Kaj pri posameznem področju gledamo, je podrobneje opisano spodaj.

Kriteriji ocenjevanja po področjih

1. Numerične metode (vsako od šestih predpisanih poglavij ocenjujemo ločeno)

- *Pravilnost in ustreznost metode:*
 - problem je rešen **numerično**; zgolj simbolna rešitev (*sympy*) ali zapis enačb ne zadoštuje,
 - metoda ustreza problemu (npr. predoločen sistem rešite po metodi najmanjših kvadratov, periodičnega signala ne aproksimirate s polinomom, diferencialno enačbo višjega reda pretvorite v sistem enačb 1. reda itd.),
 - če je to mogoče je rezultat **preverjen**: s primerjavo z analitično rešitvijo, drugo numerično metodo ali obstoječo funkcijo iz *numpy/scipy* (to velja tudi za rešitve, ki jih predlaga UI),

- uporaba obstoječih funkcij (**numpy**, **scipy**) je zaželen; lastna implementacija mora biti utemeljena in primerjana z obstoječo funkcijo.
- *Argumentacija in komentarji*, pri vsaki uporabljeni metodi pisno pojasnite:
 - **zakaj** ste izbrali to metodo in ne katere druge (npr. Newtonova metoda ali bisekcija? zakaj LU razcep? zakaj zlepki in ne polinom?, itd.),
 - **na kratko**: kakšne so predpostavke metode in kakšne so njene **omejitve** (natančnost, konvergenca, red metode, izbira koraka, robne točke pri odvajanju ...),
 - kaj rezultat pomeni za vaš fizikalni problem (**interpretacija**).
 - **Pomembno**: v seminarju **ne ponavljate izpeljav, teoretičnega ozadja metod** itd., če to ni bistveno za razumevanje ali izvajanje vaše kode. Teorija je predmet pisnega dela izpita.

2. Poročilo in koda

- *Poročilo*:
 - osnovni podatki (ime in priimek, vpisna številka, datum) in izjava o avtorstvu,
 - definicija naloge z opisom in **sliko** problema ter navedbo virov,
 - poglavja po **predlogi** (glejte tudi **primer projekta**); iz naslovov je razvidno, kje je katera numerična vsebina,
 - razlaga v **markdown** celicah, enačbe zapisane z LaTeX-om,
 - rezultati prikazani grafično (**matplotlib**), z označenimi osmi in enotami,
 - kratek opis rabe UI v razdelku *Raba umetne inteligence*.
- *Koda*:
 - poročilo se po **Restart** → **Run All** izvede **brez napak**, poti do datotek so relativne (vse potrebne datoteke so priložene, koda deluje kjerkoli, ne le na vašem računalniku),
 - koda je berljiva in v smiselnem vrstnem redu,
 - večkrat uporabljene funkcije so zbrane v **lastnem modulu** (.py), ki ga v poročilo uvozite,
 - funkcije imajo **docstringe** (opis, parametri, vrnjena vrednost) in smiselne komentarje,
 - dodatni paketi so navedeni z navodili za namestitev (npr. **requirements.txt**).

3. Lasten odnos in kreativni dodatek

- *Obseg in ambicioznost*: glejte *Pričakovan obseg projekta* zgoraj; za oceno 10 pričakujemo obseg, pri katerem bi vloženi čas brez UI bistveno presegel obseg tega predmeta.
- *Lasten odnos*: problem je smiselna celota in ne nabor nepovezanih nalog; numerične vsebine izhajajo iz problema; analiza je poglobljena (npr. vpliv parametrov, primerjava metod, ocena napake); rezultati so kritično ovrednoteni (ali so fizikalno smiselni?).
- *Kreativni dodatek*: vsebina, ki na predavanjih in vajah ni bila obravnavana (glejte seznam spodaj). Ocenjujemo kakovost izvedbe, ne le prisotnost; pri ocenah 9 in 10 je kreativni dodatek pričakovan.

4. Testi kode in/ali uporabniški vmesnik

Vsaj en test lastnih funkcij (npr. z **numpy.testing** ali **pytest**) ali preprost uporabniški vmesnik. Test mora biti v poročilu izveden in uspešen.

Najpogostejše pomanjkljivosti preteklih projektov, zaradi katerih so bile ocene nižje:

- metoda je uporabljena, ni pa pojasnjeno, zakaj ta, kako deluje in kako natančen je rezultat,
- poglavje je rešeno le simbolno,
- lastna implementacija metode brez utemeljitve in primerjave z obstoječo funkcijo,
- rezultati brez interpretacije,
- koda se zaradi napačnih poti ali manjkajočih paketov ne požene,
- koda, ki jo je napisala UI, študent pa je na zagovoru ne zna pojasniti,
- rezultati, ki jih ni nihče preveril.

Kreativni dodatek: primeri vsebin

Kot **kreativni dodatek** se tretira programska vsebina, ki na predavanjih in vajah ni bila posebej obravnavana. Z orodji UI so taki dodatki precej bolj dostopni, zato jih pri ocenah 9 in 10 pričakujemo. Na primer:

- animacija rezultatov z uporabo matplotlib-a,
- druge oblike naprednih prikazov (Plotly, Bokeh, vispy, pyqtgraph, VTK...),
- avtomatsko branje podatkov iz spleta, pošiljanje rezultatov po elektronski pošti,
- strojno generiranje poročila,
- uporaba knjižnice Pandas, zapis rezultatov v bazo podatkov, MS Excel,
- uporaba platform Raspberry Pi, Arduino, zajem podatkov,
- izdelava spletne aplikacije (Flask, Django),
- uporaba modulov in knjižnic, kot so scikit-learn, TensorFlow, PyOpenCV.

Navodila za oddajo projekta

- Rok za oddajo projekta je naveden spodaj, med pomembnimi datumi.
- Za izdelavo projekta uporabite **predlogo**; za zgled si oglejte **primer projekta**.
- Projekt oddajte v obliki zip datoteke z imenom: **Ime in priimek, vpisna številka.zip**.
Primer: Janez Novak, 23201111.zip.
- Velikost zip datoteke naj ne presega **20 MB**.
- **ZIP oddate v e-učilnici.**

Pred oddajo:

1. v projektu zakomentirajte (dodajte `#`) morebitno namestitev dodatnih paketov in funkcije, kot je na primer `%%timeit`:

```
# !pip install ime_paketa
# %%timeit
```

2. Preverite, da imate poti do slik in drugih datotek podane v *relativni* obliki (npr. 'slika.jpg', ne pa 'D:\Numerične metode\projekt\slika.jpg').
3. naredite Kernel → Restart in nato Cell → Run All. Prepričajte, da se vse celice izvedejo pravilno.

Plagiatorstvo pri individualnem projektu

Vsi oddani individualni projekti so pregledani s sistemom za ugotavljanje plagiatorstva.

Sistem preverja podobnost oddanih dokumentov z vsemi ostalimi projekti iz trenutnega in preteklih študijskih let. Kot plagiat je označeno vsako ujemanje teksta in postopkov oddane kode (tudi v primeru, da je ta uporabljena v drugačnem kontekstu, drugem vrstnem redu, z zamenjanimi imeni spremenljivk itd.). Iz pregleda je izvzeta vsebina predlog za predavanja, objavljenih v tem repozitoriju.

Če se znatni del vsebine vašega projekta ujema z drugim projektom, tega ne moremo oceniti kot vaše samostojno delo. To pomeni, da **niste opravili vaj in nimate pogoja za pristop k izpitu**.

0.2.6 Potek ocenjevanja projekta

Projekt ocenjujemo v dveh korakih: strojni pregled in ustni zagovor

1. **Strojni pregled oddanih poročil.**

Vsa oddana poročila najprej pregledamo strojno: preverimo, ali se poročilo izvede brez napak, ali vsebuje vse predpisane vsebine, in ga primerjamo z ostalimi projekti (glejte *Plagiatorstvo* zgoraj).

2. **Ustni zagovor** pri izvajalcu predmeta, približno **10 minut na študenta**.

Študent na kratko predstavi svoj projekt (problem, izbrane metode, ključni rezultati), nato izvajalec zastavi nekaj vprašanj o podrobnostih, npr.: zakaj ste izbrali določeno metodo, kako deluje vaša koda, kako ste preverili rezultat, kako ste uporabili UI. Lahko vas prosimo tudi, da v kodi naredite manjšo spremembo (npr. spremenite parameter ali robni pogoj) in pokažete njen vpliv.

Končna ocena projekta se oblikuje na podlagi obeh korakov. Projekta, ki ga študent na zagovoru ne zna pojasniti, ne moremo oceniti kot samostojno delo.

Zagovor individualnih projektov bo približno 1 teden po oddaji.

0.2.7 Domače naloge (moj.ladisk.si)

Tedenske domače naloge vam dodelimo v sistemu moj.ladisk.si. Domače naloge so dokumenti Jupyter Notebook (.ipynb), rešujete jih v okolju Visual Studio Code.

Raba UI pri domačih nalogah je dovoljena, vendar vam svetujemo, da jih rešujete sami: na izpitu UI ni na voljo, domače naloge pa so glavna priprava nanj.

Splošna navodila za reševanje

- Datoteke, ki jih prenesete s spleta, shranite v obliki .zip arhiva (če vaš brskalnik datoteki ne doda končnice .zip, to storite sami).
- Arhiv z datotekami, ki ga prenesete s spleta, **razpakirate v celoti**.
- V mapi z razpakiranimi datotekami odprete ukazno okno in poženete ukaz `jupyter notebook`.
- Celice v .ipynb datoteki z domačo nalogo poganjate po vrsti (tudi celice povsem na vrhu datoteke!).
- Rezultati so pri reševanju posredovani v spletni sistem moj.ladisk.si in se samodejno sprotno preverjajo. Pri reševanju je **nujna povezava s spletom**.

Pripravili smo tudi [video navodila za reševanje](#).

Več podrobnosti glede reševanja domačih nalog v sistemu moj.ladisk.si: moj.ladisk.si/help.

Časovna omejitev reševanja domačih nalog

Domače naloge imajo definiran čas *okna za reševanje*:

- Do odprtja okna za reševanje je možno **predčasno reševanje naloge brez časovne omejitve**.
- V oknu za reševanje je dovoljen čas reševanja naloge omejen. Časovna omejitev teče od trenutka, ko *znotraj okna za reševanje* nalogo prvič odprete.
- Po zaprtju okna se naloga zapre, reševanje **ni več možno**.

Svetujemo vam, da *čim večji del naloge rešite v času predčasnega reševanja, pred odprtjem okna za reševanje*.

Uporaba Jokerja za premik obveznosti

Vsakemu študentu sta pri predmetu na voljo dva Jokerja, ki jih lahko uporabite za premik obveznosti v prihodnost.

- Z uporabo Jokerja lahko okno za reševanje domače naloge premaknete za ≤ 14 dni naprej.
- Uporaba jokerja **ni možna, če ste nalogo že odprli**.
- Pred pričetkom okna za reševanje je za premik naloge potreben en Joker, v času okna za reševanje pa dva.

0.2.8 Predlog ocene na podlagi sprotne študija

Predlog ocene:

- 50 do 60%: 6
- 60 do 70%: 7
- 70 do 80%: 8
- 80 do 90%: 9
- 90% in več: 10

(Nedefinirana področja so vmesne ocene.)

Individualni projekt in predlog ocene se zagovarja na ustnem zagovoru (glejte *Potek ocenjevanja projekta*).

Pri zagovoru upoštevamo tudi kvalitativno oceno sodelovanja na vajah in predavanjih.

Če izpolnjujete pogoje sprotne dela (DN, projekt) imate **eno** možnost opravljanja preizkusa le iz teorije na zimskih izpitnih rokih po koncu semestra.

0.2.9 Pomembni datumi

1. Oddaja individualnega projekta: **12. jan 2027 do 12h** *
2. Preizkus iz teorije se piše na kateremkoli zimskem izpitnem roku (imate eno možnost).
3. Zagovori: po izpitnem roku.

* V primeru oddaje vsaj 1 teden **pred rokom**, se uspeh množi z 1.1; v primeru **zamude** pa se upeh množi z 0.9^n , kjer je n število začetih dni zamude.

0.3 Izpit

Izpolnjevanje pogojev za pristop k izpitu: vsak posamični dogodek (razen teorije in sodelovanja pri predvanjih), ki sestavlja oceno mora biti večji od 40%.

Izpit se opravlja v dveh delih: 40 min za tri teoretična vprašanja ([glejte primer izpita](#)) in 35 min preverjanje uporabe numeričnih metod za računalnikom (tipično dve nalogi, s sabo imate lahko poljubne pisne vire, dostopa do interneta ni).

Pri obeh delih izpita raba orodij UI **ni dovoljena**.

Za pozitivno oceno morata biti oba dela pozitivna.

0.4 Literatura

1. Slavič J.: Programiranje in numerične metode s Pythonom, [izvršljiva knjiga](#) ali [spletna knjiga](#), 2015-
2. Petrišič J.: Uvod v Matlab za inženirje, 2011
3. Demšar J. Python za programerje, 2012 ([glejte spletno učilnico FRI](#))
4. Nicolas P. Rougier: From Python to Numpy, 2017 [spletna knjiga](#)
5. Kiusalaas J: Numerical Methods in Engineering with Python 3, 2013
6. Fangohr H. Python for Computational Science and Engineering, 2014 ([dosegljiva na spletu](#))
7. Bucky R. Python Programming Tutorials ([videi na YouTube](#))
8. [Projekt TOMO](#) (več vsebin, glejte predvsem predmete na FMF in FRI)
9. [Scientific Python Lectures](#) (prej *Scipy Lecture Notes*)
10. Uporaba paketov Numpy, Scipy itd za inženirje [Mr. P Solver \(YouTube\)](#)

0.5 Pogoji za poslušanje v naprej?

Predmeta ni mogoče opravljati v naprej.

Poglavje 1

Uvod v Python

1.1 Kaj je ekosistem Pythona?

Marsikaterega bralca beseda *ekosistem* zmoti/zmede. [Slovar slovenskega knjižnega jezika](#) 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 različnih paketov (samo [pypi.org](#) trenutno hrani več kot 685 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 nivojem abstrakcije. **Nizkonivojski** programski jeziki, v nasprotju, imajo nizek nivo abstrakcije (npr. strojna koda).

Poglejmo si izračun [Fibonaccijevega števila](#), ki sledi definiciji:

$$F_1 = F_2 = 1 \quad \text{in} \quad F_n = F_{n-1} + F_{n-2}. \quad (1.1)$$

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

Izračun F_n v [strojni kodi](#): _____

```

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

```

MASM koda
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

```

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

števila = [1., 2, 3, 4, 5] # seznam števil
[število**2 for število in števila if število>3]

[16, 25]

```

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

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

```
[1, 3, 5]
```

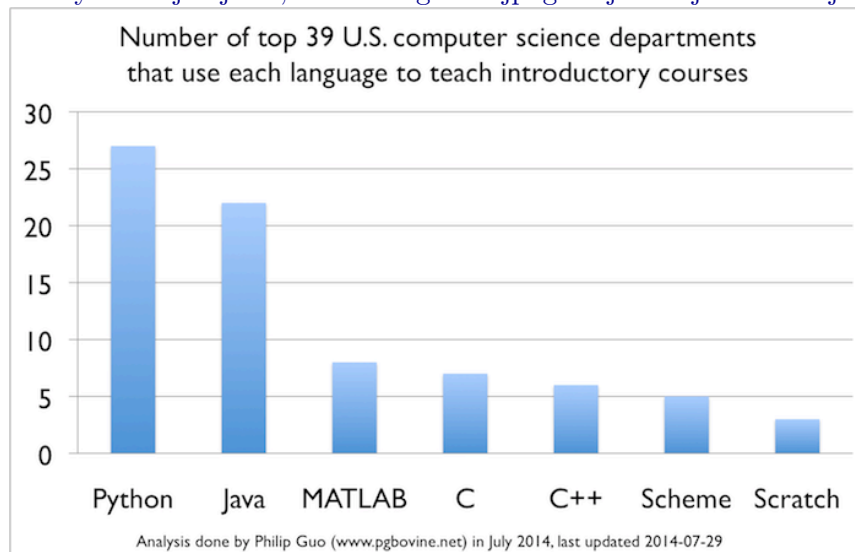
Kratek pregled začetkov nekaterih programskih jezikov:

- 1957 Fortran, * 1970 Pascal, * 1972 C
- 1980 C++, * 1984 Matlab, * 1986 Objective-C
- **1991 Python**, * 1995 Java, * 1995 Delphi
- 2006 Rust, 2009 Go, 2012 Julia

1.2 Kdo se uči Python, zakaj ga uporabljati?

ZDA

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



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

```
%%html
<blockquote class="twitter -tweet" data -lang="en"><p lang="en" dir="ltr">First day
of class in Foundations of <a
href="https://twitter.com/hashtag/Datascience?src=hash">#Datascience</a> for <a
href="https://twitter.com/BerkeleyDataSci">@BerkeleyDataSci</a> - all materials at <a
href="https://t.co/yYoJbaKlnL">https://t.co/yYoJbaKlnL</a> <a
href="https://t.co/J6DLd0VQT1">pic.twitter.com/J6DLd0VQT1</a></p>&mdash; Cathryn
Carson (@CathrynCarson) <a
href="https://twitter.com/CathrynCarson/status/900408152348270592">August 23,
2017</a></blockquote>
<script async src="//platform.twitter.com/widgets.js" charset="utf -8"></script>
```

First day of class in Foundations of #Datascience for @BerkeleyDataSci - all materials at <https://t.co/yyoJbaKlnL> [pic.twitter.com/J6DLd0VQTl](https://t.co/yyoJbaKlnL)
 — Cathryn Carson (@CathrynCarson) August 23, 2017

Python v zadnjih 10 letih pospešeno pridobiva na popularnosti in je danes eden od najboljših (IEEE Spectrum.) in tudi najpopularnejših programskih jezikov (Why is Python Growing so Quickly? in Python's explosive growth).

1.3 Uporaba Pythona?

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

Nekatera podjetja v Sloveniji: *Gorenje, Kolektor, Iskra Mehanizmi, Hidria, Mahle, Domel, ebm-papst Slovenia, Efaflex...*

Čedalje več programov, ki imajo vgrajeno skriptno podporo za Python: *Abaqus, Ansys Workbench, MSC (Adams, SimXpert), MS Excel, ParaView, LMS Siemens, AVL.*

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

Obširni seznam je [tukaj](#).

1.4 Python vs Matlab

Matlab je bil v preteklosti pogosto poučevan na univerzah in imel tudi veliko uporabnikov v industriji. Že vrsto let je ekosistem Pythona za večino opravil bistveno močnejše orodje. Na spletu boste našli ogromno primerjav, npr.: [Python vs Matlab](#).

Kratka primerjave glede na [vir](#):

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
o	=	so-so	
-	=	poor/no	

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

Če ste mlajši od 25-30 let, preostanek te celice verjetno lahko preskočite.

Če niste, ste se verjetno učili Matlab in se sprašujete zakaj je v vašem klubu čedalje manj članov. Verjetno je čas za prehod na Python; tukaj prispevek, ki vam lahko pomaga: [MATLAB vs Python: Why and How to Make the Switch](#).

1.5 Namestitev Pythona

Pojdite na www.python.org ter namestite Python; izberite najnovejšo različico (trenutno 3.14) (64 bit). Pri tem sledite pripravljenim [video navodilom](#).

1.5.1 Uporaba githuba repozitorija in posodobitve

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

- v izvorni obliki na naslovu github.com/jankoslav/pym,
- v spletni obliki na naslovu <https://jankoslav.github.io/pym>.

Najbolje je za sinhronizacijo s spletnim repozitorijem uporabljati kakšnega od Git klientov, npr. [GitHub Desktop](#) ali znotraj Visual Studio Code (predstavljen v nadaljevanju). Git je sicer zelo močno orodje in skupini razvojniki omogoča relativno enostavno delo na isti kodi in kontrolo nad različnimi verzijami datotek.

1.5.2 Visual Studio Code



Pri tem predmetu bomo v glavnem programirali v Jupyter Lab/Notebooku znotraj [Visual Studio Code - VSC](#), kar nam omogoča zelo učinkovito programiranje ob uporabi polno formatiranega besedila in interaktivnih prikazov. Za namestitev VSC sledite [pripravljenim video navodilom](#).

Kadar želimo programirati obsežnejše in bolj splošno uporabne pakete ter namenske programe, pa se ponavadi poslužimo še drugih funkcij, ki nam jih razvojno okolje (angl. *integrated development environment* - IDE) VSC ponuja. V zadnjem obdobju VSC, zaradi hitrosti, enostavnosti uporabe in razširljivosti, pridobiva na priljubljenosti. Kako nam lahko VSC pomaga pri programiranju odlično prikaže video posnetek na temo [produktivnosti z VSC](#).

1.5.3 Nameščanje dodatkov in posodobitev

Na pypi.org se nahaja veliko različnih paketov! Namestimo jih iz ukazne vrstice* (angl. *command prompt*) z ukazom **pip**.

* Ukazno vrstico priključite v operacijskem sistemu Windows tako, da kliknete "Start" in napišite "command prompt" ter pritisnete "enter" ali pa odprete terminal znotraj VSC ([video](#)).

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

1. Odprite ukazno vrstico,
2. sprožite ukaz za posodobitev paketov, ki jih bomo potrebovali tekom predmeta:

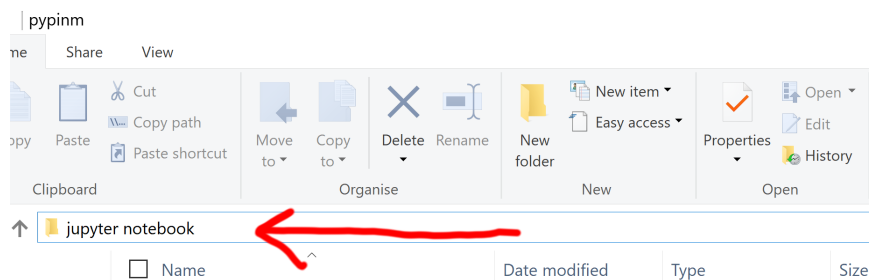
```
| pip install numpy scipy matplotlib sympy requests jupyter - --upgrade
```

1.6 Jupyter notebook (Jupyter lab)

[Jupyter notebook](#) 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 ali colab.research.google.com.

Jupyter notebook zaženemo (v poljubni mapi) iz *ukazne vrstice* ali iz *Raziskovalca/Explorerja* z ukazom: `jupyter notebook`,



ali znotraj VSC kot je predstavljeno v [posnetku](#).

Opombe:

- s sprožitvijo **jupyter lab** pridemo do nadgradnje **jupyter notebook** okolja (rahlo bolj zahtevno okolje, vendar bistveno bolj napredno okolje),
- tukaj predpostavimo, da je pot do datoteke `jupyter[.exe]` dodana v spremenljivke okolja (in jo lahko prožite iz ukazne vrstice v poljubni mapi),
- če želite ukazno vrstico sprožiti v poljubni mapi, potem v *explorerju* pridržite **shift** in kliknite mapo z desnim gumbom; nato izberite *Odpri ukazno vrstico 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]).

1.6.1 Markdown

Jupyter notebook omogoča zapis v načinu *markdown*, ki ga je v letih 2003-2004 predstavil [John Gruber](#). 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 uporabo na Githubu) so [podana tukaj](#); spodaj jih bomo na kratko ponovili.

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](#) 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 `:a3=\sqrt{365}$`,
- blokovni način (angl. *block mode*): se začne in konča z znakoma `\`, primer: `$$\sum F=m\,\ddot{x}$$`, kar se izriše kot:

$$\sum F = m \ddot{x} \quad (1.2)$$

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**.

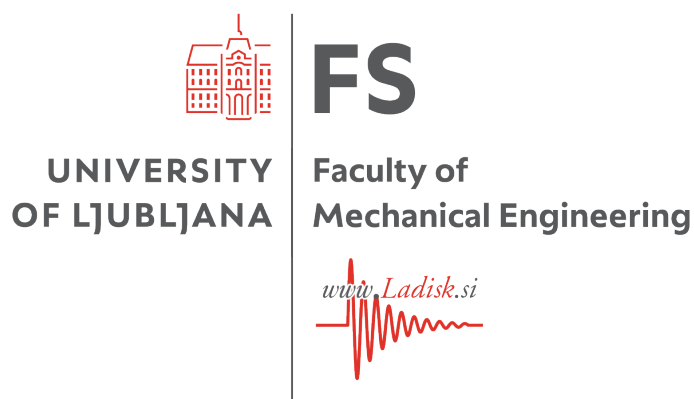
Če želimo oštevilčene sezname, začnemo z `“1.”` nato pa nadaljujemo z `“1.”`:

1. prvi element
2. drugi
3. naslednji ...

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

Uporabimo:

```
| ![Ladisk (to je opcijski opis slike)](./fig/ladisk.png)
```



Za prikaz:

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):

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

Magičnih ukazov je v notebooku še veliko (glejte [dokumentacijo](#)) 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] (<https://www.ladisk.si>) se oblikuje kot: [Spletna stran laboratorija Ladisk](#).

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:

```
| Ime | Masa [kg] | Višina [cm] |
|: -| -:|: -:|
| Lakotnik | 180 | 140 |
| Trdonja | 30 | 70 |
| Zvitorepec | 40 | 100 |
```

Rezultira v:

Ime	Masa [kg]	Višina [cm]
Lakotnik	180	140
Trdonja	30	70
Zvitorepec	40	100

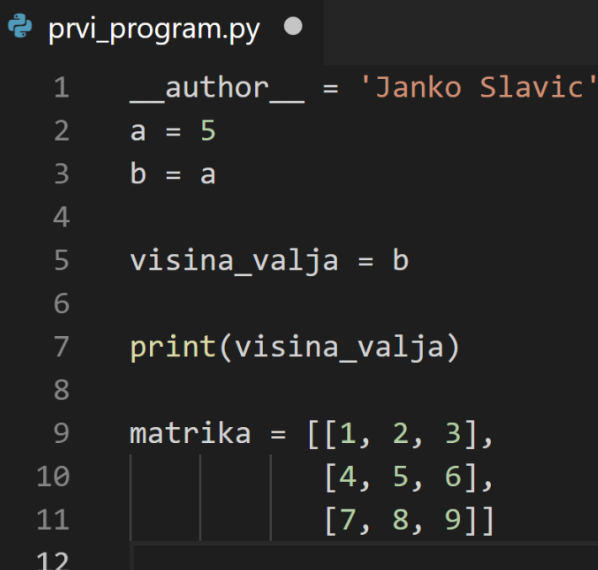
Opazimo, da dvopičje definira poravnavo teksta v stolpcu (levo :-, desno -: ali sredinsko :-:).

1.7 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`.



```

1  __author__ = 'Janko Slavic'
2  a = 5
3  b = a
4
5  visina_valja = b
6
7  print(visina_valja)
8
9  matrika = [[1, 2, 3],
10             [4, 5, 6],
11             [7, 8, 9]]
12

```

Primer [kode](#) v okolju *Visual Studio Code*:

To datoteko nato poženemo v ukazni vrstici (predpostavimo, da je pot do datoteke python.exe dodana v spremenljivke okolja, sicer glejte tale [vir](#)):

```
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*:

```
a = 5.0 # to je komentar
print(a) # izpiše vrednost a

5.0
```

1.7.1 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](#), 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: `širina_valja = 5`,
- za razrede uporabljamo t. i. format CamelCase.
- pri aritmetičnih operatorjih 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.

Opomba: moderna razvojna orodja nam pomagajo pri oblikovanju kode, ki ustreza PEP8 (v VSC pritisnite `ctrl+shift+P` ter napišite “*Format document*” in pritisnite `enter`)!

PEP20

PEP20 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.

1.8 Osnove Pythona

1.8.1 Osnovni podatkovni tipi in operatorji

Dinamično tipiziranje

Python je dinamično tipiziran jezik. Tipi spremenljivk se preverjajo šele med izvajanjem, ne ob zapisu kode. Posledično nekatere operacije niso mogoče na nekaterih tipih oz. so prilagojene tipu podatka (pozneje bomo na primer pogledali množenje niza črk). Tip se določi pri dodelitvi vrednosti, poglejmo primer:

```
| a = 1 # to je komentar: a je celo število (integer)
| a
|
| 1
```

Tip prikažemo z ukazom `type`:

```
| type(a)
|
| int
```

Vrednost `a` lahko prepisemo z novo vrednostjo:

```
| a = 1.0 # to je sedaj število s plavajočo vejico (float)
| a      # če `a` tako zapišemo v zadnji vrstici, se bo izpisala vrednost
|
| 1.0
|
| type(a)
|
| float
```

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

```
| 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 dvojnimi 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 izvirno kodo, če je ta na voljo).

Primer:

```
| ?print
| ??print
```

Logični operatorji

Logični operatorji so ([vir](#)):

- `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)`.

Primerjalni operatorji

Primerjalni operatorji ([vir](#)):

- `<` 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:

```
| 5 < 6
```

```
| True
```

Lahko naredimo niz logičnih operatorjev:

```
| 1 < 3 < 6 < 7
```

```
| True
```

preverimo istost (ali gre za isti objekt):

```
| a = 1
| a == 1
```

```
| True
```

Uporabimo še logični operator:

```
| 1 < 3 and 6 < 7
```

```
| True
```

```
| 1 < 3 and not 6 < 7
```

```
| 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:

```
| 1 and True
```

```
| True
```

```
| 0 and True
```

```
| 0
```

```
| (1,) and True
```

```
| True
```

Podatkovni tipi za numerične vrednosti

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

- `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:

```
| celo_število = -2**3000          # `**` predstavlja potenčni operator
| realno_število = 3.141592
| kompleksno_število = 1 + 3j     # `3j` predstavlja imaginarni del
```

Operatorji numeričnih vrednosti

Operatorji, razvrščeni po naraščajoči prioriteti ([vir](#)):

1. `x + y` vsota,
2. `x - y` razlika,
3. `x * y` produkt,
4. `x / y` deljenje,
5. `x // y` celoštevilsko deljenje (rezultat je celo število zaokroženo navzdol),
6. `x % y` ostanek pri celoštevilskem deljenju,
7. `-x` negiranje,
8. `+x` nespremenjen,
9. `abs(x)` absolutna vrednost,
10. `int(x)` celo število tipa `int`,
11. `float(x)` realno število tipa `float`,
12. `complex(re, im)` kompleksno število tipa `complex`,
13. `c.conjugate()` kompleksno konjugirano število,
14. `divmod(x, y)` vrne par `(x // y, x % y)`,
15. `pow(x, y)` vrne `x` na potenco `y`,
16. `x ** y` vrne `x` na potenco `y`.

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

Poglejmo si primer. Namesto:

```
a = 1
a = a + 1 # prištejemo vrednost 1
a
```

```
2
```

lahko zapišemo:

```
a = 1
a += 1
a
```

```
2
```

Operatorji na nivoju bitov

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

- $x \mid y$ *ali* na nivoju bitov x in y ,
- $x \hat{\wedge} y$ *ekskluzivni ali* (samo en, ne oba) na nivoju bitov x in y ,
- $x \& y$ bitni *in* na nivoju bitov x in y ,
- $x \ll n$ premik bitov x za n bitov levo,
- $x \gg n$ premik bitov x za n bitov desno,
- $\sim x$ negiranje bitov x .

Poglejmo si primer ([vir](#)):

```
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
```

```
1
```

Prikažimo število v bitni obliki:

```
bin(c)
```

```
'0b1'
```

1.8.2 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](#)) in da lahko poljubne znake po standardu *Unicode* uporabimo tako v tekstu, kakor tudi v programskih datotekah .py.

Niz se začne in zaključa z dvojnimi " ali enojnim ' narekovajem.

```
b = 'tekst' # b je niz (string)
b
```

```
'tekst'
```

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

```
\pi = 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:

```
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 s plavajočo vejico

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](#)) ali po slovensko terke so sezname poljubnih **objektov**, ki jih ločimo z vejico in pišemo znotraj okroglih oklepajev:

```
terka_1 = (1, 'programiranje', 5.0)

terka_1

(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:

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

'p8og8ami8anje'
```

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

```
terka_2 = (1,)
type(terka_2)

tuple
```

Terke lahko zapišemo tudi brez okroglih oklepajev:

```
terka_3 = 1, 'brez oklepajev', 5.

terka_3

(1, 'brez oklepajev', 5.0)
```

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

```
terka_1[0]

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](#)) se zapišejo podobno kot terke, vendar se uporabijo oglati oklepaji:

```
| seznam = [1., 2, 'd']  
  
| seznam  
  
| [1.0, 2, 'd']
```

Za razliko od terk, se sezname lahko spreminja:

```
| seznam[0] = 10  
| seznam  
  
| [10, 2, 'd']
```

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

```
| b = seznam  
| b  
  
| [10, 2, 'd']
```

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

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

```
| b[0] = 9  
| b  
  
| [9, 2, 'd']  
  
| seznam  
  
| [9, 2, 'd']
```

Če želimo narediti kopijo podatkov, potem moramo narediti tako:

```
| b = seznam.copy()  
| b[0] = 99  
| b  
  
| [99, 2, 'd']  
  
| seznam  
  
| [9, 2, 'd']
```

Primer seznama seznamov:

```
| matrika = [[1, 2, 3],  
|             [4, 5, 6],  
|             [7, 8, 9]]  
  
| matrika  
  
| [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Več o matrikah bomo izvedeli pozneje.

Izbrane operacije nad seznamami

Tukaj si bomo pogledali nekatere najbolj pogoste operacije, za več, glejte [dokumentacijo](#).

Najprej si pogledajmo dodajanje elementa:

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

[0, 1, 'test', 'dodajanje elementa']
```

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

```
seznam.insert(1, 'na drugo mesto')
seznam

[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:

```
odstranjeni_element = seznam.pop(2) # odstrani 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`:

```
seznam

[0, 'na drugo mesto', 'test', 'dodajanje elementa']

seznam.index('test')

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 **hitrejšje** od seznamov; poleg tega jih ne moremo po nerodnosti **spremeniti**. Za širši odgovor glejte [vir](#).

Množice (Sets)

Množice so v principu podobne *množicam* kot jih poznamo iz matematike. Množice v Pythonu ([dokumentacija](#)) 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()`:

```
A = {'a', 4, 'a', 2, 'ž', 3, 3}
A

{2, 3, 4, 'a', 'ž'}
```

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

```
B = set([6, 4, 6, 6])
B
{4, 6}
```

Uporabimo lahko tipične matematične operacije nad množicami:

```
B - A    # elementi v B brez elementov, ki so tudi v A
{6}

A | B    # elementi v A ali B
{2, 3, 4, 6, 'a', 'ž'}

A & B    # elementi v A in B
{4}

A ^ B    # elementi v A ali B, vendar ne v obeh
{2, 3, 6, 'a', 'ž'}
```

Slovar (dictionary)

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

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

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

```
parametri['višina']
5.2
```

Pogosteje uporabljamo metode:

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

Primer:

```
parametri.items()
dict_items([('višina', 5.2), ('širina', 43), ('g', 9.81)])
```

Slovarje lahko spreminjamo; npr. dodamo element:

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

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

```
parametri.pop('lega_težišča')
99999
parametri
{'višina': 5.2, 'širina': 43, 'g': 9.81}
```

Operacije nad sestavljenimi podatkovnimi strukturami

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

- `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 pojavitev elementa `x`.

Nekateri primeri:

```
| 'a' in 'abc' # in primerja ali je objekt v nizu, seznamu,...
```

```
| True
```

Zgoraj smo imeli slovar `parametri`:

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

Preverimo, ali vsebuje element s ključem `širina`:

```
| 'širina' in parametri
```

```
| True
```

Štetje in rezanje sestavljenih podatkovnih struktur

Predpostavimo spodnji niz:

```
| abeceda = 'abcčdefghijklmnoprsštuvzž'
```

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](#), stran 28) ali pa tudi prof. dr. Edsger W. Dijkstra v zapisu na [spletu](#).

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 privzeta vrednost:

- `i` je privzeto 0, kar pomeni prvi element,
- `j` je privzeto `len(s)` (dolžina zaporedja), zato je vključen tudi zadnji element,
- `k` je privzeto 1.

Prvi znak je torej:

```
| abeceda[0]
```

```
| 'a'
```

Prvi trije znaki so:

```
| abeceda[:3]
| 'abc'
```

Od prvih 15 znakov, vsak tretji je:

```
| abeceda[:15:3]
| 'ačfil'
```

Zadnjih petnajst znakov, vsak tretji:

```
| abeceda[-15::3]
| '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):

```
| abeceda[::-1]
| 'žzvutšsrponmlkjihgfedčcba'
```

1.8.3 Kontrola toka programa

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

Stavek if

Tipična oblika stavka if ([dokumentacija](#)):

```
| 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:

```
| 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 > 5`. 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:

```
| if a > 5: print('taka oblika je odsvetovana'); print(40*' -'); print('uporaba
| podpičja namesto nove vrstice')
|
| taka oblika je odsvetovana
| - - - - -
| uporaba podpičja namesto nove vrstice
```

Izraz if

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

Sintaksa je:

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

Preprost primer:

```
'Je res' if 5==3 else 'Ni res'
```

```
'Ni res'
```

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

Zanka while

Sintaksa zanke while ([dokumentacija](#)) 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:

```
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](#)). 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:

```
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:

```
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
```

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:

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

[1, 4, 9]
```

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

Zgornji primer bi bil:

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

[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](#)); spodaj bomo uporabili blokovni magični ukaz (označuje ga `%`):

```
%timeit -n1000
```

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

Najprej daljši način:

```
seznam = range(1000)
```

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

52.3 \mus \pm 1.1 \mus per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)

Nato izpeljevanje seznamov (bolj pregledno in malenkost hitreje):

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

48.3 \mus \pm 1.33 \mus per loop (mean \pm std. dev. of 7 runs, 1,000 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`:

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

[11, 4, 9]

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

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

[4, 9]

Funkcija `zip`

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

```
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](#)). 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:

```
for i in range(3):
    print(i)
```

```
0
1
2
```

Funkcija `enumerate`

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

Poglejmo primer:

```
for i, ime in enumerate(imena):  
    print(i, ime)
```

```
0 Jaka  
1 Miki  
2 Luka  
3 Anja
```

1.9 Za konec: plonk list :)

Veliko jih je; tukaj je [povezava](#) do enega.

Še nekaj branja: [automatetheboringstuff.com](#).

In en twit:

```
%%html  
<blockquote class="twitter -tweet" data -lang="en"><p lang="en" dir="ltr">The most  
important slide from my <a href="https://twitter.com/PyData">@PyData</a> talk. <a  
href="https://t.co/Hc8j4ZpHV4">pic.twitter.com/Hc8j4ZpHV4</a></p>&mdash; Almar Klein  
(@almarklein) <a  
href="https://twitter.com/almarklein/status/708909917684568064">March 13,  
2016</a></blockquote>  
<script async src="//platform.twitter.com/widgets.js" charset="utf -8"></script>
```

The most important slide from my @PyData talk. [pic.twitter.com/Hc8j4ZpHV4](#)
— Almar Klein (@almarklein) [March 13, 2016](#)

Poglavje 2

Funkcije, izjeme, moduli

2.1 Funkcije

2.1.1 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:

```
| print('prvi argument', 'drugi', '=', 5.)  
  
| prvi argument drugi = 5.0
```

Za bolj splošno uporabo glejmo [dokumentacijo](#):

```
| 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. *separator*),
- `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 zaključí (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*,
- `\b` izbriše zadnji znak *backspace*,
- `\'` izpiše enojni narekovaj `'`,
- `\"` izpiše dvojni narekovaj `"`,
- `\\` prikaže *izhodni znak* (angl. *backslash*),
- `\` na koncu vrstice: niz se nadaljuje v naslednji vrstici (brez znaka za novo vrstico).

Primer (kako ponavadi ne programiramo):

```
| print('pet','et', sep=' -', end='')  
| print('p:')  
  
| pet -et=p:)
```

In še primer z izhodnimi znaki:

```
print('Nizi v Pythonu (po PEP8) naj ne bi bili daljši od 79 znakov. \
Razlog je v tem, da nam ni treba pomikati okna levo in desno. \
Da pa vseeno lahko napišemo nize, ki so daljši od 79 znakov \
Uporabimo enojni delilni znak \\ (bodite pozorni na to, \
kako se izpiše: \\)',)
```

```
Nizi v Pythonu (po PEP8) naj ne bi bili daljši od 79 znakov. Razlog je v tem, da nam
ni treba pomikati okna levo in desno. Da pa vseeno lahko napišemo nize, ki so daljši
od 79 znakov Uporabimo enojni delilni znak \ (bodite pozorni na to, kako se izpiše:
\)
```

2.1.2 Oblikovanje nizov

Pri oblikovanju/formatiranju nizov imamo na voljo več orodij:

- **f-niz** (angl. *f-string*, *f* zaradi “formatirani”), [dokumentacija](#),
- metoda `.format()`, [dokumentacija](#),
- `%` oblikovanje, [dokumentacija](#).

Poglejmo si primere:

```
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](#)) 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](#)).

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

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

V zavitih oklepajih je:

- pred : podamo ime vrednosti, ki jo želimo oblikovati
- za : oblikujemo izpis spremenljivke; najpogosteje bomo uporabili *fmt* oblike:
 - **w.df** - predstavitev s plavajočo vejico (**float**)
 - **w.de** - predstavitev z eksponentom (**exponent**)
 - **w.dg** - splošni format (**general**)
 - **ws** - niz znakov (**string**).

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

Poglejmo si primer:

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

'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:

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

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

Primer z uporabo slovarja:

```
parametri = {'visina': 5., 'gostota':100.111, 'ime uporabnika': 'Janko'}
f'višina = {parametri["visina"]:7.3f}, gostota = {parametri["gostota"]:7.3e}'
'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](#)):

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

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

- *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 števč po decimalnem ločilu,
- *type* definira, kako naj bodo podatki prikazani, možnosti: "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" | "n" | "o" | "s" | "x" | "X" | "%"

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

```
f'{višina:*^20}' # pred ^ je znak s katerim zapolnimo levo in desno stran
'*****1.84*****'

f'{parametri["visina"]:_.>20}'
'_____5.0'

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

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

Oblikovanje datuma, ure si lahko pogledate v dodatku spodaj.

2.1.3 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](#)):

```
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' tekstovna oblika zapisa (privzeto),
- '+' datoteka se odpre za branje in pisanje.

Zadnji trije znaki ('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](#).

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'`):

```
| 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`:

```
| datoteka.write('test prve vrstice\n')
```

```
| 18
```

Metoda `write` vrne število zapisanih znakov. 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:

```
| datoteka.write('druga vrstica\n', )
```

```
| 14
```

Vpišimo sedaj še nekaj številčnih vrednosti:

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

Datoteko zapremo:

```
| datoteka.close()
```

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

```
| 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:

```
| 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:

```
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](#)). Celovitost stavka `with` presega namen tega predmeta, je pa zelo preprosta uporaba pri delu z datotekami, zato si pogledajmo primer:

```
with open('data/prikaz vpisa.txt') as datoteka:
    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
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.

2.1.4 Funkcije v splošnem

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

Generični primer funkcije je ([dokumentacija](#)):

```
def ime_funkcije(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:

```
def površina(dolžina, širina):
    S = dolžina * širina
    return S
```

```
površina(1, 20)
```

```
20
```

2.1.5 Posredovanje argumentov v funkcije

Python pozna dva načina posredovanja argumentov:

- glede na **mesto** (*positional*)
- glede na **ime**.

Poglejmo si oba načina na primeru:

```
| površina(3, 6) # pozicijsko
| 18
| površina(širina = 3, dolžina = 6) # glede na ime
| 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:

```
| površina(3, širina=4)
| 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 *dolžina* prvi argument, bi klicanje take kode:

```
| površina(3, dolžina=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:

```
| def neka_funkcija(student, izpit = True):
|     ''' Prva verzija funkcije '''
|     print(f'Student {student} se je prijavil na izpit: {izpit}.')
|
| 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):

```
| 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'Student {student} se je prijavil na izpit: {izpit}.')
|     if izpisi_sliko:
|         print('slika:')
|
| neka_funkcija('Janez', izpit=True)
|
| Študent Janez se je prijavil na izpit: True.
|
| 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:

```
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:

```
| izpis(oblika_a, 3)
|
| Prikaz vrednosti: 3
|
| izpis(oblika_b, 3)
|
| Prikaz vrednosti: 3.00
```

2.1.6 Docstring funkcije

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

[Dokumentacija](#) 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.

Primer dokumentirane funkcije:

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

    Funkcija vrne vrednost površine.

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

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

```
| površina()
|
| 10
```

2.1.7 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:

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

```
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**.

2.1.8 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:

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

```
vrnem_terko()
```

```
(1, 2, 3)
```

Rezultat funkcije lahko ujamemo tako:

```
rezultat = vrnem_terko()
rezultat
```

```
(1, 2, 3)
```

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

```
i1, i2, i3 = vrnem_terko()
i1 # izpis samo ene vrednosti
```

```
1
```

2.1.9 Anonimna funkcija/izraz

Anonimna funkcija (glejte [dokumentacijo](#)) 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:

```
seznam = [ -4, 3, -8, 6]
```

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

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

Metoda ima dva opcijska argumenta:

- `key` ključ, po katerem razvrstimo elemente
- `reverse`, 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](#)) pa vrne seznam urejenih vrednosti.

Poglejmo primer:

```
| seznam = [ -4, 3, -8, 6]
| seznam.sort()
| seznam
|
| [ -8, -4, 3, 6]
```

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

```
| seznam.sort(key = lambda a: a**2)
| seznam
|
| [3, -4, 6, -8]
```

2.2 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 kod 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:

```
try:
    1/0
except:
    pass
```

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

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

```
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:

```
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.

2.2.1 Proženje izjem

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

Primer proženja napake je:

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

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

```
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`.

2.3 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](#).

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

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

Pomembno: pogosto module hierarhično razporejamo po direktorijih; direktorij z moduli imenujemo *paket* (angl. *package*). Če direktorij vsebuje datoteko (lahko tudi prazno) `__init__.py`, gre za običajni paket; od Pythona 3.3 naprej pa se direktorij obnaša kot paket tudi brez nje (t. i. *namespace package*). V zgornjem primeru se tako mapa `moduli` obnaša kot *paket*, v njem pa je *modul* `prvi_modul` (datoteka `prvi_modul.py`).

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 `kvadrat()` kličemo tako: `prvi_modul.kvadrat()`.

Drugi način uvažanja modula:

```
| import moduli.prvi_modul
```

Uvozimo *imenski prostor* `moduli.prvi_modul`; to pomeni, da funkcijo `kvadrat()` kličemo tako: `moduli.prvi_modul.kvadrat()`. (Sam ukaz `import moduli` podmodulov **ne** uvozi samodejno.)

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

```
| from moduli import prvi_modul
```

Sedaj lahko funkcijo `kvadrat` kličemo tako:

```
| prvi_modul.kvadrat(5)
```

```
| 25
```

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

```
| from moduli.prvi_modul import kvadrat
```

Sedaj kličemo neposredno funkcijo `kvadrat`:

```
| kvadrat(6)
```

```
| 36
```

Funkcije ali module lahko tudi preimenujemo. Primer:

```
| from moduli.prvi_modul import kvadrat as sqr  
| sqr(7)
```

```
| 49
```

Kje Python išče module?

1. V trenutni mapi (to je tista, v kateri ste prožili `jupyter notebook`),
2. v mapah, definiranih v `PYTHONPATH`,
3. v mapah, kjer je nameščen Python (poddirektorij `site-packages`).

2.3.1 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!**

2.3.2 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 realnih 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](#).

`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="strict")` za branje datoteke `file`.

Uporabo si bomo pogledali na primeru. Najprej uvozimo modul:

```
| import pickle
```

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

```
| 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 zadnjo verzijo
```

Opazimo, da so datoteko odprli za pisanje v binarni obliki `wb`. Pomembno je tudi, da uporabimo zadnjo verzijo protokola (parameter `protocol`); privzeta je izbrana zaradi združljivosti s starejšimi različicami Pythona, zadnja pa je praviloma najhitrejša.

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

```
| with open('data/stanje1.pkl', 'rb') as datoteka:
|     b = pickle.load(datoteka)
|
| b
|
| {'število': 1, 'velikost': 10}
```

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

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

```
| data = list(range(50000))
```

Pisanje v tekstovni obliki

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

```
| 25.1 ms \pm 1.46 ms per loop (mean \pm std. dev. of 7 runs, 10 loops each)
```

Pisanje s pomočjo `pickle`:

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

```
| 1.12 ms \pm 55.6 \mus per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)
```

Pohitritev je več desetkratna!

2.4 Uporaba modulov iz spleta

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.

Modul je tehnično gledano ena datoteka, kadar nek večji modul vsebuje več modulov, pa lahko začnemo govoriti o *paketi*. Najbolj pomembne pakete na področju inženirskih ved bomo spoznali v okviru te knjige.

Module oz. pakete lahko *posodabljamo* ali nameščamo *nove*, pri tem nam pomagajo t. i. *upravljalniki paketov* (angl. *package manager*). Najbolj pogost upravljalnik paketov je `pip` ([dokumentacija](#)).

2.4.1 Upravljalnik paketov pip

`pip` je upravljalnik paketov z dolgo zgodovino in podpira veliko število paketov ([pypi.org](#)).

Že nameščene pakete najdemo z ukazom v ukazni vrstici ([dokumentacija](#)):

```
| pip 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 `%sx` - shell execute, glejte [dokumentacijo](#)). Opomba: dva klicaja bi vrnila celoten rezultat neposredno v notebook (ker je izpis zelo dolg, smo se temu izognili).

Poglejmo primer:

```
| rezultat = !pip list
| rezultat[:5]

| ['Package          Version',
|  '-----',
|  '-----',
|  'attrs            26.1.0',
|  'btrfsutil        7.1',
|  'certifi           2026.7.22']
```

Pakete namestimo z ukazom (v ukazni vrstici):

```
| pip install [ime_paketa]
```

posodobimo z:

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

in odstranimo z:

```
| pip uninstall [ime_paketa]
```

Za več glejte [dokumentacijo](#).

Primer namestitve paketa

Sicer pa pakete najpogosteje najdemo na spletu ([pypi.org](#)). Pojdite na omenjeni portal in poiščite pakete na temo snemanja posnetkov iz portala [www.youtube.com](#). Z iskanjem “youtube download” najdemo obetaven in dobro vzdrževan paket `yt-dlp`, ki ga namestimo (v ukazni vrstici):

```
| pip install yt-dlp
```

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

```
import yt_dlp
nastavitve = {'format': 'best[ext=mp4]'} # najboljša kakovost v formatu mp4
with yt_dlp.YoutubeDL(nastavitve) as yt: # kreiramo instanco objekta (kaj to točno
    pomeni, spoznamo pozneje)
    yt.download(['https://www.youtube.com/watch?v=P4IrAXbpeyI']) # prenesemo video
```

2.5 Dodatno

2.6 Oblikovanje datuma in časa

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

```
import datetime
sedaj = datetime.datetime.now()
sedaj

datetime.datetime(2026, 9, 25, 15, 0, 17, 342722)
```

Podatek o času oblikujemo:

```
f'{sedaj:%Y -%m -%d %H:%M:%S}'

'2026 -09 -25 15:00:17'
```

Oblikovanje datuma in ure sledi metodam:

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

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

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

'2026 -09 -25 15:00:17 Fri'
```

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

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

datetime.datetime(2017, 9, 28, 5, 31, 45)
```

2.6.1 Nekateri moduli

1. `openpyxl` za pisanje in branje Excel `xlsx/xlsm` datotek: [vir](#),
- Regularni izrazi (zelo močno orodje za iskanje po nizih): [vir](#).

2.6.2 Modul sys

Python ima veliko število vgrajenih modulov (in še veliko se jih lahko prenese s spleta, npr. [tukaj](#)), ki dodajo različne funkcionalnosti.

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

Poglejmo si primer; najprej uvozimo modul:

```
| import sys
```

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

```
| sys.path
```

```
| ['c:\\_j\\Work Janko\\Pedagosko delo\\_predavanja\\Programiranje in numericne
metode\\pynm\\notebooks',
| 'C:\\Users\\janko\\AppData\\Local\\Programs\\Python\\Python311\\python311.zip',
| 'C:\\Users\\janko\\AppData\\Local\\Programs\\Python\\Python311\\DLLs',
| 'C:\\Users\\janko\\AppData\\Local\\Programs\\Python\\Python311\\Lib',
| 'C:\\Users\\janko\\AppData\\Local\\Programs\\Python\\Python311',
| '',
| 'C:\\Users\\janko\\AppData\\Local\\Programs\\Python\\Python311\\Lib\\site
-packages',
| 'C:\\Users\\janko\\AppData\\Local\\Programs\\Python\\Python311\\Lib\\site
-packages\\win32',
| 'C:\\Users\\janko\\AppData\\Local\\Programs\\Python\\Python311\\Lib\\site
-packages\\win32\\lib',
| 'C:\\Users\\janko\\AppData\\Local\\Programs\\Python\\Python311\\Lib\\site
-packages\\Pythonwin']
```

2.6.3 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:

```
| import os
```

Poglejmo trenutno mapo:

```
| os.path.curdir
```

```
| '.'
```

ali trenutno mapo v absolutni obliki:

```
| os.path.abspath(os.path.curdir)
```

```
| 'c:\\_j\\Work Janko\\Pedagosko delo\\_predavanja\\Programiranje in numericne
metode\\pynm\\notebooks'
```

Kako najdemo vse datoteke in mape v direktoriju?

```
| seznam = os.listdir()
```

Prikažimo prve štiri s končnico `ipynb`:

```
| 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: Predavanje 03 - Moduli, numpy, matplotlib.ipynb
Našel sem datoteko: Predavanje 07 - Interpolacija.ipynb
Našel sem datoteko: Untitled.ipynb
Našel sem datoteko: Predavanje 02 - Print, delo z datotekami, funkcije, moduli.ipynb
```

Za več glejte [dokumentacijo](#).

Poglavje 3

Moduli, numpy, matplotlib

3.1 Osnove modula numpy

Modul `numpy` namestimo z upravljalnikom paketov `pip` z naslednjim ukazom (v kolikor ga še nimamo nameščenega):

```
| pip install numpy
```

oz. paket posodobimo na najnovešo različico:

```
| pip install numpy - -upgrade
```

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

```
| import numpy as np
```

Gre za enega najbolj pomembnih modulov. Na kratko: gre za visoko optimiran modul za numerične izračune; zelo dobri tutoriali so zbrani na naslovu: github.com/numpy/numpy-tutorials. Za odličen uvod v Numpy si lahko ogledate [YouTube posnetek: NumPy Tutorial \(2021\): For Physicists, Engineers, and Mathematicians](#)

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

```
| 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:

```
| np.zeros((3,4))
```

```
| array([[0., 0., 0., 0.],  
        [0., 0., 0., 0.],  
        [0., 0., 0., 0.]])
```

ali pa:

```
| np.zeros((3,5))
```

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

```
| np.ones(4, dtype=int)
|
| array([1, 1, 1, 1])
```

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

```
| 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 določi iz ostalih argumentov).

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

```
| np.arange(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:

```
| np.arange(7, 12, 2, dtype=float)
|
| array([ 7.,  9., 11.])
```

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

```
| 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 določi iz ostalih argumentov).

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

```
| np.linspace(-np.pi, np.pi, 10)
|
| 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](#)).

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

Opomba: funkcije `numpy.random.seed()`, `rand()`, `random()`, `normal()` itd. spadajo v starejši (*legacy*) vmesnik, ki še vedno deluje. Sodobni pristop je, da si ustvarimo lasten generator in kličemo njegove metode ([dokumentacija](#)):

```
rng = np.random.default_rng(0) # seme 0
rng.random(3)                 # enakomerna porazdelitev na [0, 1)
rng.normal(size=3)             # normalna porazdelitev
```

Spodnja vrstica:

```
np.random.seed(0)
```

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

```
np.random.rand(3)
array([0.5488135 , 0.71518937, 0.60276338])
```

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

```
np.random.seed(0)
np.random.random(3)
array([0.5488135 , 0.71518937, 0.60276338])
```

3.1.1 Zapis matrik in vektorjev

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

```
a = np.zeros((3, 2))
a
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

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

```
b = np.zeros(3) # (1 x 3)
b
array([0., 0., 0.])
```

ali kot **stolpični vektor**:

```
c = np.zeros((3, 1)) # 3 x 1
c
array([[0.],
       [0.],
       [0.]])
```

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

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

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](#)). Pristopa `numpy.matrix` tukaj ne bomo obravnavali.

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

3.1.2 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](#)) je:

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

pri tem velja:

- indeksiranje se začne z 0 (kot sicer pri Pythonu),
- **od** pomeni $\geq$,
- **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:

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

```
| array([3, 5, 7])
```

Primer od elementa 3 naprej:

```
| b[3:]
| array([3, 4, 5, 6, 7, 8, 9])
```

Primer od elementa 3 naprej, vendar vsak tretji:

```
| b[3::3]
| array([3, 6, 9])
```

Primer zadnjih 5 elementov, vendar vsak drugi:

```
| b[-5::2]
| array([5, 7, 9])
```

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

```
| b[-5: -2:2]
| array([5, 7])
```

3.1.3 Rezanje večdimenzijskega numeričnega polja

V 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](#)) spremenimo obliko v matriko 3 x 5:

```
a = np.arange(15)
a = a.reshape((3,5))
a

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](#)).

```
a.shape

(3, 5)
```

Atribute bomo sicer podrobno spoznali pri obravnavi razredov.

Prikažimo vrstice z indeksom 0:

```
a[0]

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

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

```
a[0,:]

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

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

```
a[:,1]

array([ 1,  6, 11])
```

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

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

array([[3, 4],
       [8, 9]])
```

Podobna logika se uporabi pri dimenzijah višjih od 2.

3.1.4 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:

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

[1, 2, 3, 4, 5, 6, 7]
```

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

```
b = np.array(a)
b

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

Ko izpišemo `b`, smo opozorjeni, da gre za `array(...)`.
Poglejmo, kako se obnaša Pythonov seznam pri množenju:

```
| 2*a
| [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`:

```
| 2*b
| 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!

3.1.5 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:

```
| b + 3*b - b**2
| array([ 3,  4,  3,  0, -5, -12, -21])
```

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

```
| np.power(b, 2)
| array([ 1,  4,  9, 16, 25, 36, 49])
```

Mimogrede opazimo, da je rezultat tipa `int64` (integer). Ko smo ustvarili ime `b`, smo namreč ustvarili numerično polje z elementi tipa `int64`.

3.1.6 Matematične funkcije

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

- [trigonometrične funkcije](#),
- [hiperbolične](#),
- [funkcije za zaokroževanje](#),
- [funkcije za vsoto, produkt in odvod](#),
- [eksponenti in logaritmi](#),
- [posebne ter preostale funkcije](#).

Poglejmo primer funkcije `sin()`:

```
a = np.linspace(0, 2*np.pi, 5)
np.sin(a)

array([ 0.0000000e+00,  1.0000000e+00,  1.2246468e -16, -1.0000000e+00,
        -2.4492936e -16])
```

Poglejmo še hitrost izvajanja:

```
%timeit -n100 np.sin(a)

437 ns \pm 38.9 ns per loop (mean \pm std. dev. of 7 runs, 100 loops each)
```

3.1.7 Podatkovni tipi

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

Osredotočili se bomo predvsem na sledeče tipe:

- `int` - celo število (fiksne širine, običajno 64 bitov; za razliko od Pythonovega `int` lahko prekorači obseg)
- `float` - število s plavajočo vejico ([dokumentacija](#))
- `complex` - kompleksno število s plavajočo vejico
- `object` - Python objekt.

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

```
np.arange(5, dtype=int)

array([0, 1, 2, 3, 4])

np.arange(5, dtype=float)

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

np.arange(5, dtype=complex)

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

3.1.8 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]:

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

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:

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

array([[ 1.,  1.,  0.,  0.],
       [ 1.,  1.,  0.,  0.],
       [ 0.,  0.,  0., 100.]])
```

Z 2 pomnožimo stolpec z indeksom 1:

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

```
pogled = a[:, 2]
pogled
array([0., 0., 0.] )
```

Sedaj spremenimo izbrane vrednosti numeričnega polja **a**:

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

```
pogled
array([5., 5., 5.] )
```

Če želimo kopijo, moramo narediti tako:

```
kopija = a[:, 2].copy()
kopija
array([5., 5., 5.] )
```

in rezultat **kopija** ostane nespremenjen:

```
a[:, 2] = 2
kopija
array([5., 5., 5.] )
```

3.1.9 Osnove matričnega računanja

Če želite ponoviti matematične osnove matričnega računanja, potem sledite tej [povezavi](#) (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**:

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

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

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

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

```
| 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 [dokumentacija](#).

Poglejmo primer množenja dveh vektorjev, to lahko izvedemo tako:

```
| np.dot(x, y)
| np.int64(11)
```

ali tudi tako:

```
| x.dot(y)
| np.int64(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):

```
| np.dot(A, x)
| array([5, 7])
```

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

```
| A.dot(np.transpose([x]))
| array([[5],
|         [7]])
```

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

```
| A.T
| array([[1, 3],
|         [2, 2]])
```

Poglejmo si primer množenja dveh matrik:

```
| np.dot(A, B)
| array([[5, 5],
|         [7, 7]])
```

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

Zgornji primeri zapisani z operatorjem `@`:

```
| x @ y
```

```
| np.int64(11)
|
| A @ x
|
| array([5, 7])
|
| A @ B
|
| array([[5, 5],
|        [7, 7]])
```

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

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

kjer `a` in `b` definirata komponente vektorjev. Vektorja morata imeti tri komponente, rezultat je vektor s tremi komponentami. Ravninske vektorje (samo komponenti x in y) NumPy od različice 2.0 ne sprejme več; dopolnimo jih s komponento $z = 0$, rezultat pa ima neničelno le komponento z . Uporaba funkcije je možna tudi na večdimenzijskih numeričnih poljih in temu so namenjeni preostali argumenti (glejte dokumentacijo).

Primer vektorskega produkta ravninskih vektorjev:

```
| x = np.array([1, 0])
| y = np.array([0, 1])
| np.cross(np.append(x, 0), np.append(y, 0))[2] # dopolnimo z z=0 in vzamemo komponento
| z
|
| np.int64(1)
```

Primer vektorskega produkta prostorskih vektorjev:

```
| x = np.array([1, 0, 0])
| y = np.array([0, 1, 0])
| np.cross(x, y)
|
| array([0, 0, 1])
```

3.1.10 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](#)):

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

kjer argument `n` definira število vrstic in stolpcev kvadratne matrike. Tip `dtype` je privzeto `float`.

Primer enotske matrike:

```
| A = np.identity(3)
| A
|
| array([[1., 0., 0.],
|        [0., 1., 0.],
|        [0., 0., 1.]])
```

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

```
| 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:

```
| np.diagonal(A)
```

```
| array([1., 1., 1.])
```

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

```
| A[0, 1] = 10
```

```
| np.diagonal(A, offset=1)
```

```
| array([10., 0.])
```

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

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

Primer sledi diagonale:

```
| np.trace(A)
```

```
| np.float64(3.0)
```

in potem sosednje diagonale:

```
| np.trace(A, offset=1)
```

```
| np.float64(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](#)):

```
| 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 pogledjmo `A`):

```
| A
```

```
| array([[ 1., 10.,  0.],  
|       [ 0.,  1.,  0.],  
|       [ 0.,  0.,  1.]])
```

```
| np.max(A)
```

```
| np.float64(10.0)
```

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

```
| np.max(A, axis=0)
```

```
| array([ 1., 10.,  1.])
```

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

```
| np.max(A, axis=1)
| array([10.,  1.,  1.]
```

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

```
| A
| array([[ 1., 10.,  0.],
|        [ 0.,  1.,  0.],
|        [ 0.,  0.,  1.]])
| np.argmax(A, axis=0)
| array([0, 0, 2])
```

3.1.11 Linearna algebra z numpy

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

Za primer najprej definirajmo matriko in vektor:

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

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

```
| numpy.linalg.inv(a)
| Primer:
| np.linalg.inv(A)
| 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](#)):

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

Primer:

```
| rešitev = np.linalg.solve(A, b)
| rešitev
| array([0.66666667, 0.83333333])
```

Enakost elementov numeričnega polja `a` in `b` (znotraj določene tolerance) preverimo s funkcijo `numpy.isclose()` ([dokumentacija](#)):

```
| numpy.isclose(a, b, rtol=1e-05, atol=1e-08, equal_nan=False)
```

Primer:

```
| np.isclose(np.dot(A, rešitev), b)
| array([ True,  True])
```

3.1.12 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)

```
N = 1000
a = np.arange(N)
b = np.arange(N)
```

Izračunajmo skalarni produkt vektorjev z uporabo zanke `for`:

```
c = 0
for i in range(N):
    c += a[i] * b[i]
c
```

```
np.int64(332833500)
```

Izmerimo hitrost:

```
%%timeit -n100
c = 0
for i in range(N):
    c += a[i] * b[i]
```

```
183 \mus \pm 1.75 \mus per loop (mean \pm std. dev. of 7 runs, 100 loops each)
```

Isti rezultat pridobimo še v vektorski obliki:

```
%%timeit -n100
c = a @ b
```

```
1.44 \mus \pm 131 ns per loop (mean \pm std. dev. of 7 runs, 100 loops each)
```

Vidimo, da je vektorski način bistveno hitrejši (za še hitrejši način glejte `numba` v dodatku)!

3.2 Osnove modula `matplotlib`

V Pythonu imamo več možnosti za prikaz rezultatov v grafični obliki. Najbolj uporabni paketi so:

- `matplotlib` za visoko kakovostne, visoko prilagodljive slike (relativno počasno),
- `pyqtgraph` za kakovostne in prilagodljive uporabniške vmesnike (zelo hitro),
- `bokeh` za interaktiven prikaz v brskalniku (relativno hitro).

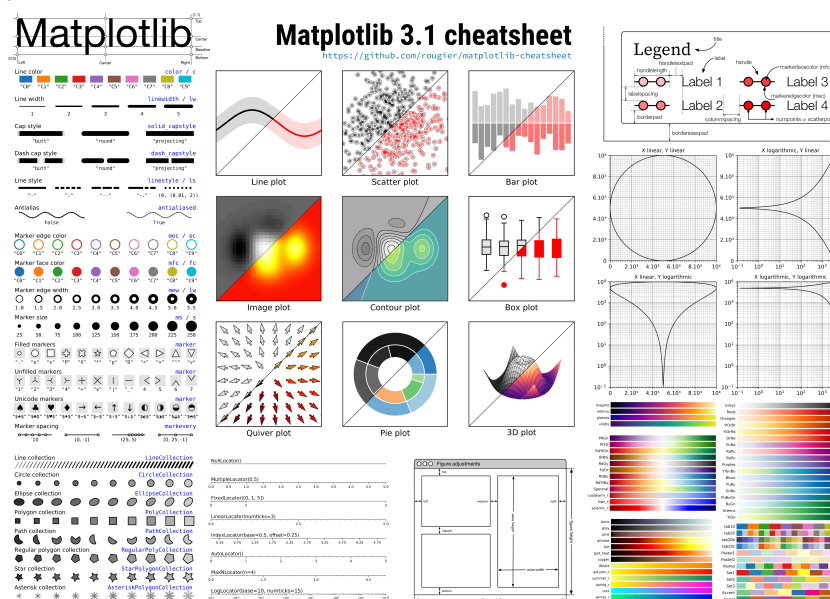
Obstaja še veliko drugih; dober pregled je naredil Jake VanderPlas na konferenci `PyCon 2017` (sicer avtor paketa za deklarativno vizualizacijo: *Altair*).

3.2.1 Osnovna uporaba

Najbolj razširjen in najbolj splošno uporabljen je paket `matplotlib`:



Sposobnosti paketa najbolje prikazuje [galerija](#). Gre za zelo sofisticiran paket in tukaj si bomo na podlagi primerov pogledali nekatere osnove.



Pri uporabi vam lahko koristi [plonk listek](#):

Tipično uvozimo matplotlib.pyplot kot plt:

```
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 widget] : slike so interaktivno vključene v notebook (**medvrstični interaktivni** način), zahteva namestitve paketa **ipyml**,
3. [%matplotlib notebook] : slike so interaktivno vključene v notebook (**medvrstični interaktivni** način).

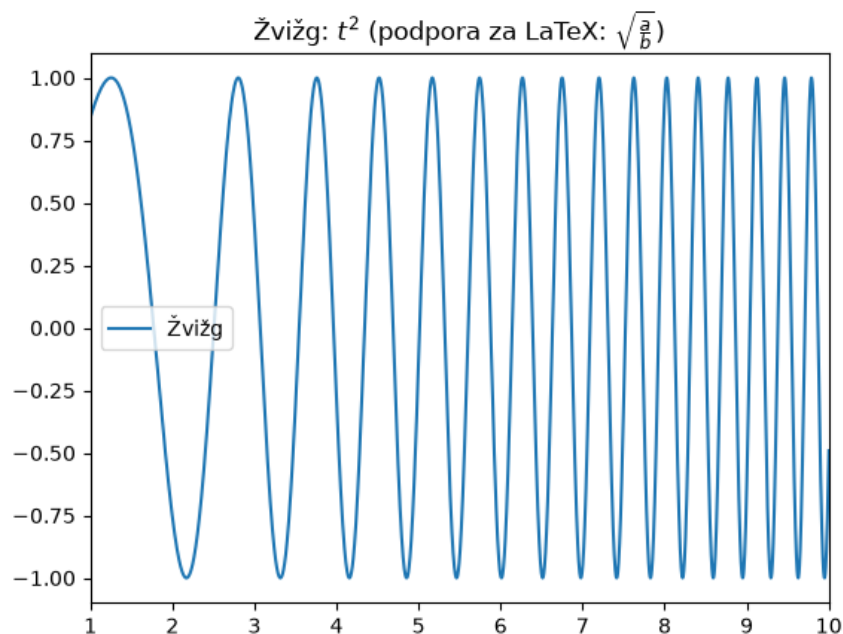
Opomba: *interaktivni način se v pasivni, spletni/pdf verziji te knjige ne prikaže pravilno.*

Tukaj bomo najpogosteje uporabljali *medvrstični* način:

```
%matplotlib inline
```

Kratek primer:

```
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();
plt.show()
```



Mimogrede, zakaj to imenujemo žvižg (oz. kvadraten žvižg)? Da dobimo odgovor, podatke predvajamo na zvočnik:

```
from IPython.display import Audio, display
display(Audio(data=žvižg, rate=44000))
```

```
<IPython.lib.display.Audio object>
```

Aktivirajmo sedaj interaktivni način (glejte tudi `%matplotlib`):

```
%matplotlib notebook
%matplotlib notebook
```

```
plt.plot([1,2,3], [2,4,5]);
```

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

Zgornjo sliko lahko sedaj interaktivno klikamo in tudi dopolnjujemo s kodo:

```
plt.title('Pozneje dodani naslov!')
plt.xlabel('Čas [t]');
```

3.2.2 Interaktivna uporaba

Pri tem predmetu bomo večkrat uporabljali interaktivnost pri delu z grafičnimi prikazi. V ta namen najprej uvozimo `interact` iz paketa `ipywidgets`:

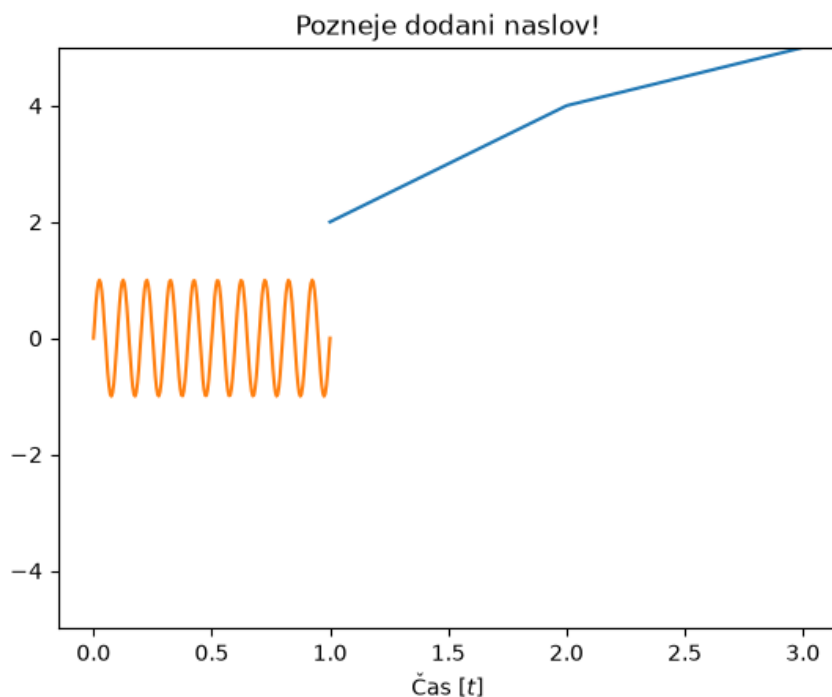
```
from ipywidgets import interact
```

Potem definiramo sliko kot funkcijo z argumenti `amplituda`, `fr`, `faza` in `dušenje`:

```
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):

```
%matplotlib inline
slika()
```



Če funkcijo za izris slike pošljemo v funkcijo `interact`, slednja poskrbi za interaktivne gumbe, s katerimi lahko spreminjamo parametre klicanja funkcije `slika`:

```
interact(slika);

interactive(children=(IntSlider(value=1, description='amplituda', max=3, min= -1),
IntSlider(value=10, descript...
```

Razpon parametrov lahko tudi sami definiramo:

```
interact(slika, amplituda=(1, 5, 1), dušenje=(0, 1, 0.05), fr=(10, 100, 1), faza=(0,
2*np.pi, np.pi/180));

interactive(children=(IntSlider(value=1, description='amplituda', max=5, min=1),
IntSlider(value=10, descripti...
```

3.2.3 Napredna uporaba

Prikaz več funkcij

Poglejmo si preprost primer prikaza več funkcij (najprej, opsijsko, definirajmo fonte za izpis v pdf):

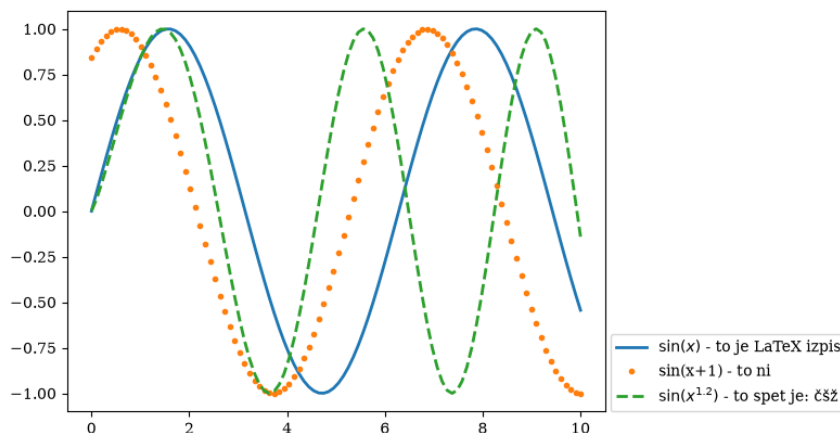
```
from matplotlib import rc
rc('font',**{'family':'serif'})
plt.rcParams['pdf.fonttype'] = 42

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=r'$\sin(x)$ - to je LaTeX izpis', linewidth = 2);
```

```
plt.plot(x, y2, '.', label='sin(x+1) - to ni', linewidth = 2);
plt.plot(x, y3, '- -', label=r'$\sin(x^{1.2})$ - to spet je: čšž', linewidth = 2);
plt.legend(loc=(1.01,0));
plt.savefig('data/prvi plot.pdf')
```



Prikaz več slik

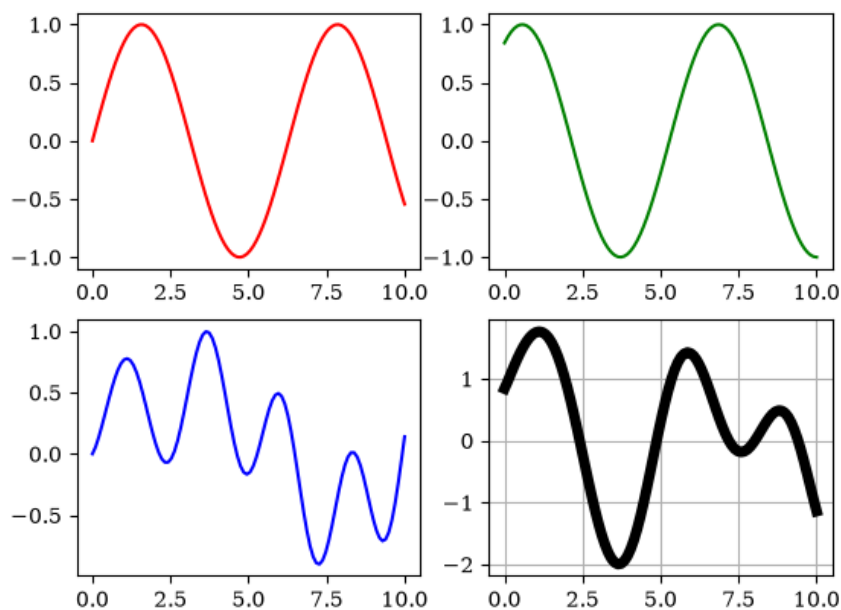
Več slik prikažemo s pomočjo metode `subplot`, ki definira mrežo in lego naslednjega izrisala.

```
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 v Pythonu, 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 (kjer se indeks začne z 1).

Delujoči primer je:

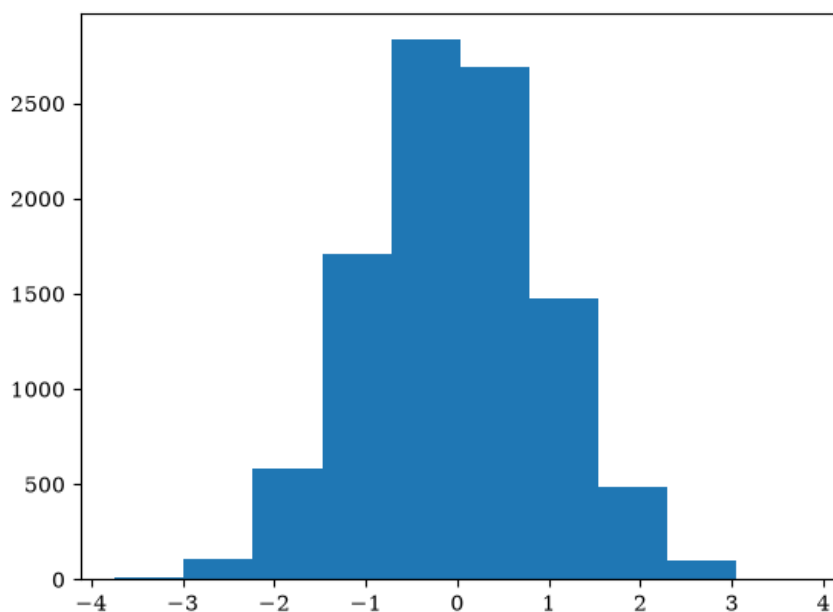
```
plt.subplot(2, 2, 1)
plt.plot(x, y1, 'r')
plt.subplot(2, 2, 2)
plt.plot(x, y2, 'g')
plt.subplot(2, 2, 3)
plt.plot(x, y2*y3, 'b')
plt.subplot(2, 2, 4)
plt.plot(x, y2+y3, 'k', linewidth=5);
plt.grid()
```



Histogram

Generirajmo 10000 normalno porazdeljenih vzorcev in jih prikažimo v obliki histograma:

```
np.random.seed(0)
x = np.random.normal(size=10000)
plt.hist(x);
```



3.2.4 Uporaba primerov iz matplotlib.org

Primere iz [galerije](http://matplotlib.org/examples/) lahko uvozimo s pomočjo magične funkcije `%load`.

Poskusite:

- `%load http://matplotlib.org/mpl_examples/lines_bars_and_markers/line_demo.py`
- `%load http://matplotlib.org/examples/widgets/slider_demo.py`

3.3 Dodatno:

3.4 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](#)) iz paketa `numba`:

```
import numpy as np
from numba import jit
```

`jit` uporabimo kot [dekorator](#) funkcije, ki jo želimo pohitriti:

```
@jit
def skalarni_produkt(a, b):
    c = 0
    for i in range(N):
        c += a[i] * b[i]
    return c
```

Sedaj definirajmo vektorja:

```
N = 1000
a = np.arange(N)
b = np.arange(N)
```

Preverimo hitrost `numpy` skalarnega produkta:

```
%%timeit -n1000
a @ b
```

```
1.48 \mus \pm 43.1 ns per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)
```

Preverimo še hitrost `numba` pohitrene verzije. Prvi klic izvedemo, da se izvede kompilacija, potem merimo čas:

```
skalarni_produkt(a, b)
332833500
```

```
%%timeit -n1000
skalarni_produkt(a, b)
```

```
808 ns \pm 114 ns per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)
```

Vidimo, da smo še izboljšali hitrost!

3.5 matplotlib: animacije, povratni klic, XKCD stil

Z `matplotlib` lahko pripravimo tudi **animacije**. Dva primera lahko najdete tukaj:

- [Preprosta animacija](#),
- [Dvojno nihalo](#).

Več v [dokumentaciji](#).

Slika s **povratnim klicem** (angl. *call back*):

- [Risanje črte](#).

Pripravite lahko tudi *na roko* narisane slike (**XKCD stil**):

- [XKCD stil](#).

3.6 Za najbolj zagrete

1. Naučite se še kaj novega na chrisalbon.com.

Poglavje 4

Objektno programiranje, simbolno računanje

4.1 Objektno programiranje

Pri programiranju poznamo različne pristope, dokumentacija Python-a (docs.python.org) omenja npr:

1. **proceduralni**: seznam navodil, kaj je treba izvesti (npr.: `C`, `Pascal`)
2. **deklarativni**: opišemo kaj želimo, programski jezik pa izvede (npr., `SQL`)
3. **funkcijski**: programiranje temelji na funkcijah (npr.: `Haskell`)
4. **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 eksplisitemu 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](#)):

```
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:

```
class Pravokotnik:
    """Razred za objekt pravokotnik"""

    def __init__(self, širina=1, višina=1): # to je konstruktor objekta. Se izvede,
    ko kličemo Pravokotnik(sirina=1, visina=4)
        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):

```
moj_pravokotnik = Pravokotnik()
```

Funkcije definirane znotraj razreda poimenujemo **metode**, ko jih kličemo na objektih.

V zgornjem primeru metodo **površina** uporabimo tako:

```
moj_pravokotnik.površina()
1
```

Ime **self** je referenca na instanco razreda (objekt, ki bo ustvarjen). Imena znotraj razreda postanejo **atributi** objekta.

Primer atributa **višina**:

```
moj_pravokotnik.višina
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:

```
tvoj_pravokotnik = Pravokotnik(višina=5, širina=3)
tvoj_pravokotnik.površina()
15
```

Pripravili smo tudi metodo, ki spremeni atribut **širina**:

```
moj_pravokotnik.set_širina(širina=100)
moj_pravokotnik.površina()
100
```

Atribute lahko spreminjamo tudi neposredno, vendar se temu (zaradi možnosti napake in napačne uporabe) ponavadi izogibamo.

Primer:

```
moj_pravokotnik.višina = 3
moj_pravokotnik.površina()
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**). 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 `object`.
Primer, ko novi razred `Kvadrat` podeduje obstoječega (`Pravokotnik`):

```
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:

```
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:

```
moj_kvadrat.površina()  
  
16
```

V kolikor spremenimo širino, se ustrezno spremeni površina:

```
moj_kvadrat.set_širina(5)  
moj_kvadrat.površina()  
  
25
```

Primer dedovanja razreda `list` (seznam)

Najprej pripravimo seznam:

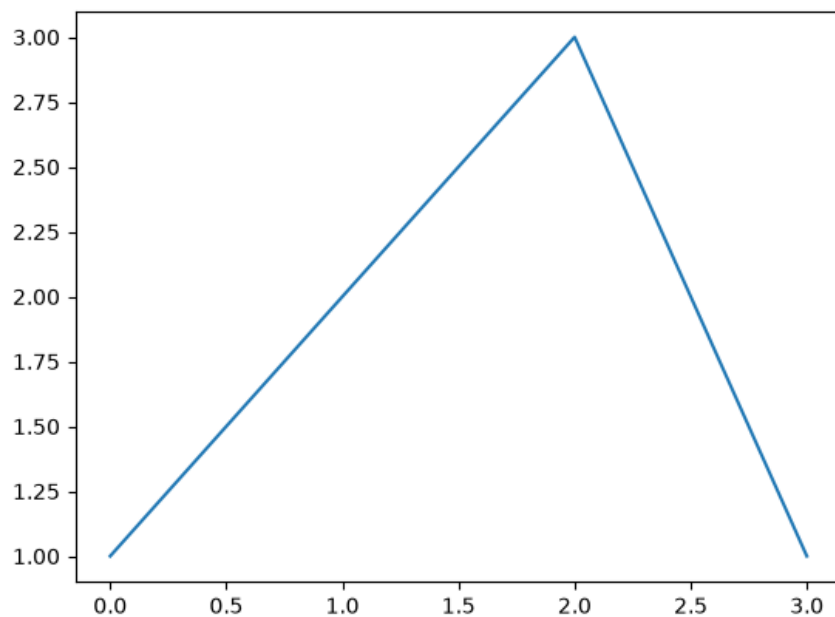
```
seznam = list([1,2,3])  
seznam  
  
[1, 2, 3]
```

Če želimo seznamu dodati vrednost, uporabimo metodo `append` (to je metoda, ki jo imajo objekti tipa `list`):

```
seznam.append(1)
```

Nato seznam prikažemo (najprej uvozimo `matplotlib`):

```
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**:

```
class Seznam(list):
    def nariši(self):
        plt.plot(self, 'r.', label='Dolgo besedilo')
        plt.legend()
        plt.ylim(-5, 5)
        plt.show()
```

Instanca objekta je:

```
moj_seznam = Seznam([1, 2, 3])
type(moj_seznam)
```

```
__main__.Seznam
```

```
moj_seznam
```

```
[1, 2, 3]
```

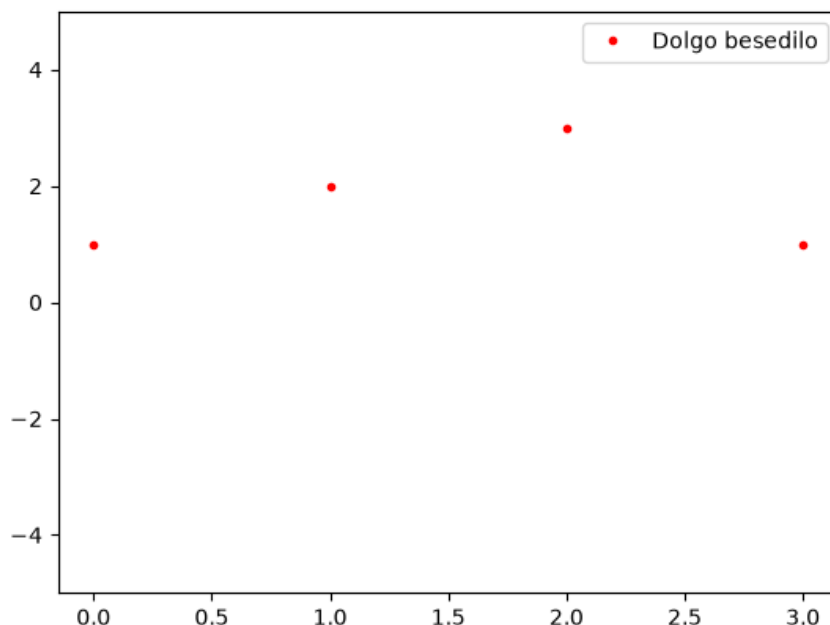
Čeprav nismo definirali metode **append**, jo je nov razred podedoval po razredu **list**:

```
moj_seznam.append(1)
moj_seznam
```

```
[1, 2, 3, 1]
```

Ima tudi metodo za izris:

```
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](#),
- [SymPy](#) - uradna dokumentacija modula,
- Odličen članek nekaterih avtorjev SymPy: [Meurer et. al, 2017](#).

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](#) na primer ni v celoti napisan v Pythonu).

Nekatera namenska komercialna orodja:

- [Mathematica](#)
- [Maple](#)

Najprej uvozimo modul SymPy; tipično paket uvozimo kot `sym`:

```
| 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 "povezijo".

Zato da dobimo lepo oblikovan LaTeX izpis, uporabimo:

```
| sym.init_printing()
```

4.2.1 Definiranje spremenljivk in numerični izračun

Spremenljivke definiramo takole:

```
| x, y, k = sym.symbols('x, y, k')
```

Preverimo lahko tip:

```
| type(x)
```

```
| sympy.core.symbol.Symbol
```

Opazimo, da je spremenljivka `x` sedaj `Symbol` iz paketa `sympy`. Sedaj lahko naredimo preprost izračun:

```
| x**y + k
```

$$k + x^y \quad (4.1)$$

Funkcijo lahko tudi poimenujemo; tukaj je primer, kjer uporabimo funkcijo sinus in konstanto π :

```
| f = sym.sin(1.2*sym.pi + x)**2
| f
```

$$\sin^2(x + 1.2\pi) \quad (4.2)$$

Bralec se morebiti sprašuje, zakaj potrebujemo *ново* 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:

```
| enačba = sym.Eq(sym.sin(k*x), 0.5)
| enačba
```

$$\sin(kx) = 0.5 \quad (4.3)$$

Nedefinirane matematične funkcije zapišemo kot ([dokumentacija](#)):

```
| g = sym.Function('g')
```

Sedaj lahko, na primer, definiramo diferencialno enačbo:

```
| sym.Eq(g(x).diff(x), x)
```

$$\frac{d}{dx}g(x) = x \quad (4.4)$$

Pri definiranju spremenljivk lahko dodajamo predpostavke:

```
| x = sym.Symbol('x', positive=True)
```

```
| x.is_positive
```

```
| True
```

Predpostavke 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:

```
| sym.sqrt(x**2)
```

$$x \tag{4.5}$$

Z metodo `assumptions0` pogledamo predpostavke objekta:

```
| x.assumptions0
| {
|   'commutative': True,
|   'complex': True,
|   'extended_negative': False,
|   'extended_nonnegative': True,
|   'extended_nonpositive': False,
|   'extended_nonzero': True,
|   'extended_positive': True,
|   'extended_real': True,
|   'finite': True,
|   'hermitian': True,
|   'imaginary': False,
|   'infinite': False,
|   'negative': False,
|   'nonnegative': True,
|   'nonpositive': False,
|   'nonzero': True,
|   'positive': True,
|   'real': True,
|   'zero': False
| }
```

`SymPy` pozna tipe števil ([dokumentacija](#)):

- `Float` realna števila (s plavajočo vejico, poljubne natančnosti),
- `Rational` racionalna števila,
- `Integer` cela števila,
- kompleksna števila zapišemo z imaginarno enoto `I` (npr. `2 + 3*sym.I`).

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*:

```
| r1 = sym.Rational(4, 5)
| r2 = sym.Rational(5, 4)
```

```
| r1
```

$$\frac{4}{5} \tag{4.6}$$

Nekateri izračuni:

```
| r1+r2
```

$$\frac{41}{20} \tag{4.7}$$

```
| r1/r2
```

$$\frac{16}{25} \tag{4.8}$$

Kompleksna števila

Imaginarno število se zapiše z I (to je drugače kot pri `numpy`, kjer je imaginarni del definiran z j):

```
| 1+1*sym.I
```

$$1 + i \quad (4.9)$$

```
| sym.I**2
```

$$-1 \quad (4.10)$$

Numerični izračun

Pri simbolnem izračunu najprej analitične izraze rešimo, poenostavimo itd., nato pa pogosto želimo tudi izračunati konkreten rezultat.

Poglejmo primer:

```
| x = sym.symbols('x')
| f = sym.exp(x**x**x + sym.pi)
| f
```

$$e^{x^{x^x} + \pi} \quad (4.11)$$

Če želimo sedaj namesto x uporabiti vrednost, npr `0.5`, to naredimo z metodo `subs()` ([dokumentacija](#)):

```
| f.subs(x, 0.5)
```

$$1.84512555475729e^\pi \quad (4.12)$$

Zgoraj smo uporabili konstanto π ([dokumentacija](#)); 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 obliki realnega števila. To moramo eksplicitno zahtevati z metodo:

- `evalf` (angl. **evaluate function*, [dokumentacija](#)) ali
- `N`,

kateri imata obe argument `n` (število števk).

Poglejmo primer:

```
| f.subs(x, 2).evalf(n=80)
```

$$205630752.2559788837336514598572458946969835445645441306426707100142289480369288 \quad (4.13)$$

Podobno je z `N`:

```
| sym.N(f.subs(x, 2), n=80)
```

$$205630752.2559788837336514598572458946969835445645441306426707100142289480369288 \quad (4.14)$$

Mimogrede smo pokazali, da pod pogojem, da v izrazu nimamo števil s plavajočo vejico, lahko rezultat prikažemo poljubno natančno ([dokumentacija](#)).

V `subs` funkciji lahko uporabimo tudi slovar. Primer:

```
| x, y = sym.symbols('x, y')
| parametri = {x: 4, y: 10}
| (x + y).subs(parametri)
```

$$14 \quad (4.15)$$

ali seznam terk:

```
| (x + y).subs([(x, 4), (y, 10)])
```

$$14 \quad (4.16)$$

Podobno ima metoda `sympy.evalf()` argument `subs`, ki sprejme slovar zamenjav, primer:

```
| (y**x).evalf(subs=parametri)
```

$$10000.0 \quad (4.17)$$

Metoda `sympy.subs` pa lahko zamenja simbol (ali izraz) tudi z drugim izrazom:

```
| (x + y).subs(x, y + sym.oo)
```

$$2y + \infty \quad (4.18)$$

4.2.2 SymPy in NumPy

Pogosto `sympy` povežemo z `numpy`. Za primer si pogledjmo, kako bi izraz:

```
| x = sym.symbols('x')
| f = sym.sin(x) + sym.exp(x)
| f
```

$$e^x + \sin(x) \quad (4.19)$$

numerično učinkovito izračunali pri tisoč vrednostih x .

Najprej uvozimo paket `numpy`:

```
| import numpy as np

| Pripravimo numerično polje vrednosti:
| x_vec = np.linspace(0, 10, 1000, endpoint=False)
| x_vec[:10]
| 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:
| y_vec = np.array([f.evalf(subs={x: vrednost}) for vrednost in x_vec])

| Opazimo, da je to dolgotrajno, zato izmerimo potreben čas:
| %timeit -n1
| y_vec = np.array([f.evalf(subs={x: vrednost}) for vrednost in x_vec])
| 87.3 ms \pm 850 \mu s per loop (mean \pm 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](#)):

```
| sympy.lambdify(simboli, funkcija, modules=None)
```

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:

```
| f_hitra = sym.lambdify(x, f, modules='numpy')
| y_vec_hitra = f_hitra(x_vec)
```

Preverimo hitrost:

```
| %%timeit -n100
| f_hitra(x_vec)
```

```
| 17.3 \mus \pm 1.36 \mus per loop (mean \pm std. dev. of 7 runs, 100 loops each)
```

Opazimo približno 10.000-kratno pohitritev!

Poglejmo še primer uporabe funkcije več spremenljivk:

```
| f_hitra2 = sym.lambdify((x, y), (x + y + sym.pi)**2, 'numpy')
| x = np.linspace(0, 10, 10)
| y = x
| f_hitra2(x, y)
```

```
| 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](#)).

Pogledali si bomo preproste primere, ki se navezujejo na funkcijo `sympy.plotting.plot`; najprej uvozimo funkcijo:

```
| from sympy.plotting import plot
```

Sintaksa uporabe funkcije `sympy.plotting.plot()` ([dokumentacija](#)) je:

```
| plot(izraz, razpon, **kwargs)
```

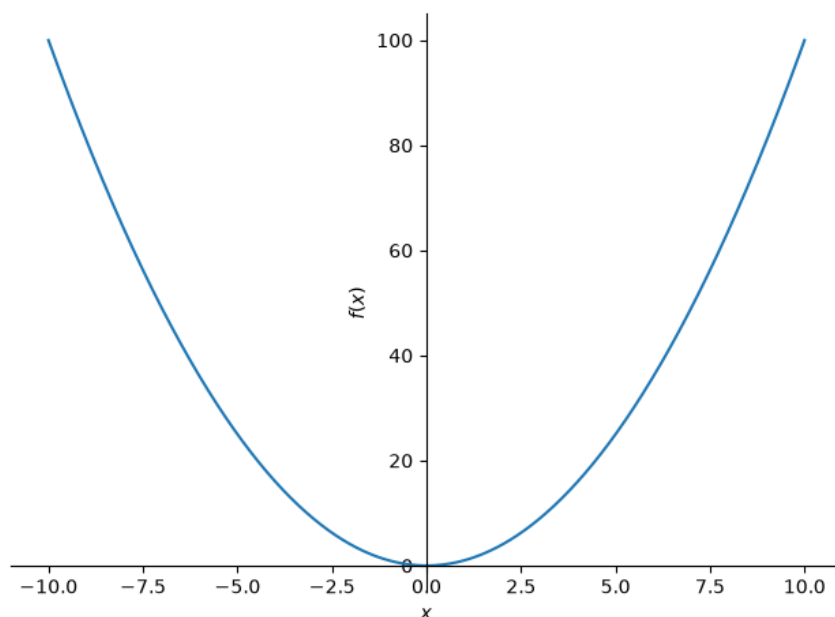
kjer so argumenti:

- `izraz` je matematični izraz ali več izrazov,
- `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:

```
| x = sym.symbols('x')
| plot(x**2)
```



```
<sympy.plotting.backends.matplotlibbackend.matplotlib.MatplotlibBackend at
0x74e7d3718470>
```

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:

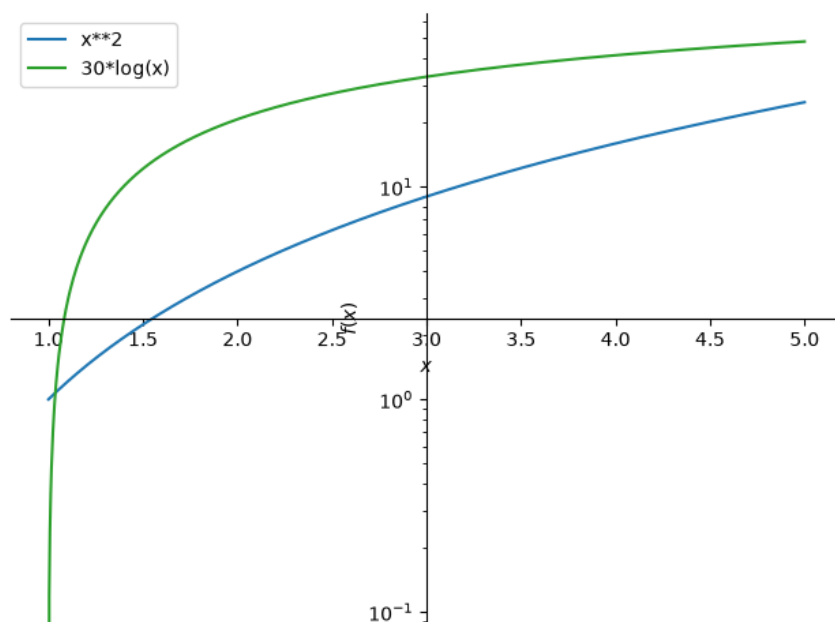
- `show` prikaže sliko (privzeto `True`),
- `line_color` barva izrisa,
- `yscale` in `yscale` način prikaza (možnosti: `linear` ali `log`),
- `xlim` in `ylim` omejitev prikaza za osi (terka dveh (`min`, `max`) vrednosti).

Pripravimo dve sliki, kjer bo y os logaritemska in bo razpon izrisa od 1 do 5:

```
| 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:

```
| izris1.extend(izris2)
| izris1.show()
```



Parametrični izris

Podobno uporabljamo funkcijo `sympy.plotting.plot_parametric` za parametrični izris ([dokumentacija](#)):

```
| plot_parametric(izraz_x, izraz_y, range, **kwargs)
```

kjer sta nova argumenta:

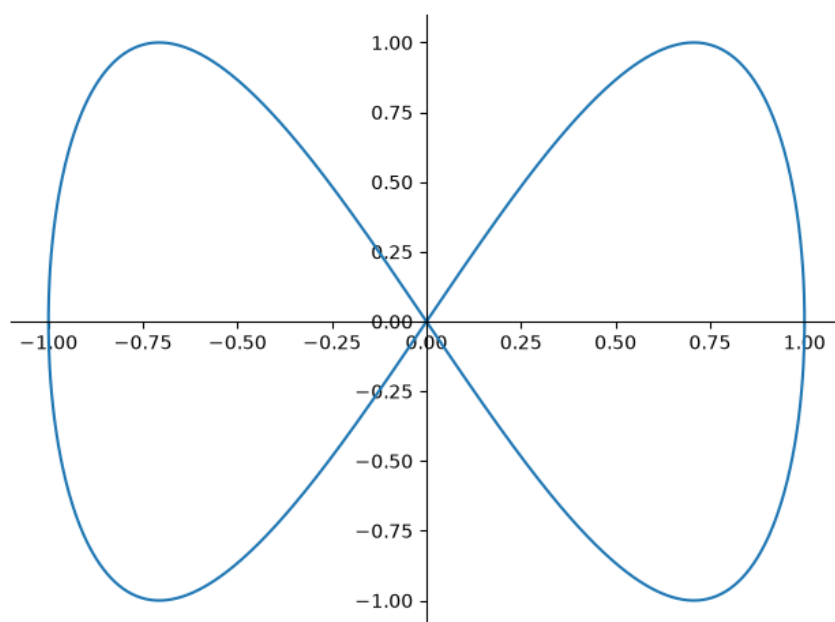
- `izraz_x` in `izraz_y` definicije lege koordinate x in y ,
- `**kwargs` je slovar možnosti.

Uvozimo funkcijo:

```
| from sympy.plotting import plot_parametric
```

Prikažimo uporabo na primeru:

```
| plot_parametric(sym.sin(x), sym.sin(2*x), (x, 0, 2*sym.pi));
```



Izris v prostoru

Funkcija `sympy.plotting.plot3d` ([dokumentacija](#)) za izris v prostoru ima sintakso:

```
| plot3d(izraz, razpon_x, razpon_y, **kwargs)
```

kjer so argumenti:

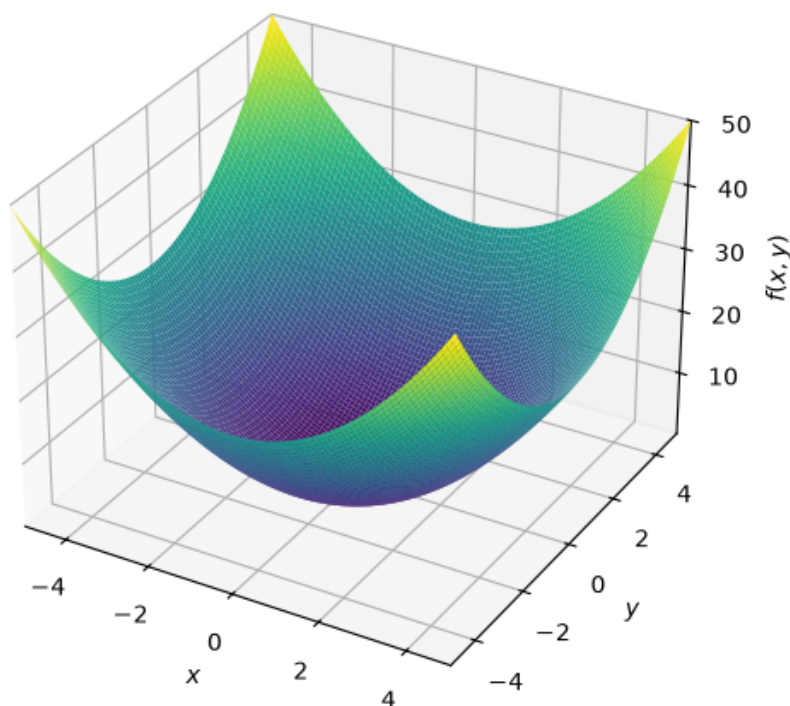
- `izraz` definicija površine,
- `razpon_x` in `razpon_y` razpon koordinate `x` in `y`,
- `**kwargs` slovar možnosti.

Uvozimo funkcijo:

```
| from sympy.plotting import plot3d
```

Prikažimo uporabo na primeru:

```
| x, y = sym.symbols('x y')
| plot3d(x**2 + y**2, (x, -5, 5), (y, -5, 5));
```



Za ostale prikaze glejte [dokumentacijo](#).

4.2.4 Algebra

V tem poglavju si bomo pogledali nekatere osnove uporabe SymPy za algebrajske operacije.

Uporaba `expand in factor`

Definirajmo matematični izraz:

```
| x = sym.symbols('x')
| f = (x+1)*(x+2)*(x+3)
| f
```

$$(x + 1)(x + 2)(x + 3) \quad (4.20)$$

in ga sedaj **razčlenimo** (angl. *expand*, glejte [dokumentacijo](#)):

```
aa = sym.expand(f)
aa
```

$$x^3 + 6x^2 + 11x + 6 \quad (4.21)$$

Če želimo pogledati koeficiente pred x , to naredimo z metodo `coeff()`:

```
aa.coeff(x)
```

$$11 \quad (4.22)$$

Argumenti funkcije definirajo, kakšno razširitev želimo ([dokumentacija](#)). Če želimo npr. trigonometrično razširitev, potem uporabimo `trig=True`:

```
a, b = sym.symbols('a, b')
sym.expand(sym.sin(a+b))
```

$$\sin(a + b) \quad (4.23)$$

```
sym.expand(sym.sin(a+b), trig=True)
```

$$\sin(a) \cos(b) + \sin(b) \cos(a) \quad (4.24)$$

Obratna operacija od razčlenitve je **razcepitev** ali **razstavljanje** ali **faktorizacija** (angl. *factor*, [dokumentacija](#)):

```
sym.factor(x**3 + 6 * x**2 + 11*x + 6)
```

$$(x + 1)(x + 2)(x + 3) \quad (4.25)$$

Če nas zanimajo posamezni členi, potem to naredimo s funkcijo `sympy.factor_list`:

```
sym.factor_list(x**3 + 6 * x**2 + 11*x + 6)
```

$$(1, [(x + 1, 1), (x + 2, 1), (x + 3, 1)]) \quad (4.26)$$

Poenostavljanje izrazov s `simplify`

Funkcija `sympy.simplify()` ([dokumentacija](#)) poskuša poenostaviti izraze v bolj preproste (npr. s krajšanjem spremenljivk).

Za posebne namene lahko poenostavimo tudi z:

- `sympy.trigsimp`,
- `sympy.powsimp`,
- `sympy.logcombine`.

Za več glejte [dokumentacijo](#).

Primeri poenostavljanja:

```
sym.simplify((x+1)*(x+1)*(x+3))
```

$$(x + 1)^2 (x + 3) \quad (4.27)$$

```
| sym.simplify(sym.sin(a)**2 + sym.cos(a)**2)
```

$$1 \quad (4.28)$$

```
| sym.simplify(sym.cos(x)/sym.sin(x))
```

$$\frac{1}{\tan(x)} \quad (4.29)$$

4.2.5 Uporaba apart in together

Funkciji uporabljamo za delo z ulomki:

```
| f1 = 1/((1 + x) * (5 + x))
| f1
```

$$\frac{1}{(x + 1)(x + 5)} \quad (4.30)$$

Razcep na parcialne ulomke (angl. *partial fraction decomposition*) izvedemo s funkcijo `sympy.apart()` ([dokumentacija](#)):

```
| f2 = sym.apart(f1, x)
| f2
```

$$-\frac{1}{4(x + 5)} + \frac{1}{4(x + 1)} \quad (4.31)$$

in potem ponovno v obratni smeri s funkcijo `sympy.together()`:

```
| sym.together(f2)
```

$$\frac{1}{(x + 1)(x + 5)} \quad (4.32)$$

V slednjem primeru pridemo do podobnega rezultata s `sympy.simplify()`:

```
| sym.simplify(f2)
```

$$\frac{1}{(x + 1)(x + 5)} \quad (4.33)$$

4.2.6 Odvajanje

Odvajanje je načeloma relativno preprosta matematična operacija, ki jo izvedemo s funkcijo `sympy.diff()` ([dokumentacija](#)):

Pripravimo primer:

```
| x, y, z = sym.symbols('x, y, z')
| f = sym.sin(x*y) + sym.cos(y*z)
| f
```

$$\sin(xy) + \cos(yz) \quad (4.34)$$

Odvajajmo ga po x :

```
| sym.diff(f, x)
```

$$y \cos(xy) \quad (4.35)$$

ali tudi

```
| f.diff(x)
```

$$y \cos(xy) \quad (4.36)$$

Odvide višjega reda definiramo tako:

```
| sym.diff(f, x, x, x)
```

$$-y^3 \cos(xy) \quad (4.37)$$

ali (isti rezultat malo drugače):

```
| sym.diff(f, x, 3)
```

$$-y^3 \cos(xy) \quad (4.38)$$

Odvod po več spremenljivkah $\frac{d^3 f}{dx dy^2}$ izvedemo takole:

```
| sym.diff(f, x, 1, y, 2)
```

$$-x(xy \cos(xy) + 2 \sin(xy)) \quad (4.39)$$

4.2.7 Integriranje

Funkcijo `integrate` lahko uporabimo za nedoločeno integriranje ([dokumentacija](#)):

```
| 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:

```
| x = sym.symbols('x')
| f = sym.sin(x*y) + sym.cos(y*z)
| sym.integrate(f, x)
```

$$x \cos(yz) + \begin{cases} -\frac{\cos(xy)}{y} & \text{for } y \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad (4.40)$$

Opazimo, da `sympy` pravilno upošteva možnost, da je $y = 0$.

Še primer določenega integriranja:

```
| sym.integrate(f, (x, -1, 1))
```

$$2 \cos(yz) \quad (4.41)$$

Primer, ko so meje v neskončnosti (uporabimo konstanto za neskončnost `sympy.oo`):

```
| sym.integrate(sym.exp(-x**2), (x, -sym.oo, sym.oo))
```

$$\sqrt{\pi} \quad (4.42)$$

4.2.8 Vsota in produkt vrste

Vsoto vrste definiramo s pomočju funkcije `sympy.Sum()` ([dokumentacija](#)):

```
| sympy.Sum(izraz, (spr, start, end))
```

kjer so argumenti:

- `izraz` izraz, katerega seštevamo,
- `spr, start` in `end` spremenljivka, ki narašča od `start` do `end` (`end` je vključen).

Primer vsote vrste:

```
| n = sym.Symbol('n')
| f = sym.Sum(1/x**n, (n, 1, sym.oo))
| f
```

$$\sum_{n=1}^{\infty} x^{-n} \quad (4.43)$$

Šele ko uporabimo metodo `doit()`, se izračun izvede:

```
| f.doit()
```

$$\begin{cases} \frac{1}{x(1-\frac{1}{x})} & \text{for } \frac{1}{|x|} < 1 \\ \sum_{n=1}^{\infty} x^{-n} & \text{otherwise} \end{cases} \quad (4.44)$$

Poglejmo še številčni rezultat:

```
| f.subs({x: 5}).evalf()
```

$$0.25 \quad (4.45)$$

Produkt vrste definiramo podobno s funkcijo `sympy.Product` ([dokumentacija](#)):

```
| 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:

```
| f = sym.Product(1/n, (n, 1, 5))
| f
```

$$\prod_{n=1}^5 \frac{1}{n} \quad (4.46)$$

```
| f.doit()
```

$$\frac{1}{120} \quad (4.47)$$

4.2.9 Limitni račun

Limite računamo s pomočjo funkcije `sympy.limit()` ([dokumentacija](#)):

```
| sympy.limit(f, x, x0)
```

kjer so argumenti:

- `f` izraz, katerega limito iščemo,
- `x` spremenljivka, ki limitira proti `x0`,
- `x0` limita.

Primer:

```
| x = sym.symbols('x')
| f = sym.sin(x)/x
| f
```

$$\frac{\sin(x)}{x} \quad (4.48)$$

```
| sym.limit(f, x, 0)
```

$$1 \quad (4.49)$$

Za primer si pogledjmo uporabo limite na definiciji odvoda:

$$\frac{df}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h, y) - f(x, y)}{h}. \quad (4.50)$$

Pripravimo funkcijo `f` in njen odvod:

```
| x, y, z, h = sym.symbols('x, y, z, h')
| f = sym.sin(x*y) + sym.cos(y*z)
```

Odvod funkcije je:

```
| sym.diff(f, x)
```

$$y \cos(xy) \quad (4.51)$$

Enak rezultat izračunamo tudi z uporabo limite:

```
| sym.limit((f.subs(x, x+h) - f)/h, h, 0)
```

$$y \cos(xy) \quad (4.52)$$

4.2.10 Taylorjeve vrste

Taylorjeve vrste izračunamo s pomočjo funkcije `sympy.series()` ([dokumentacija](#)):

```
| 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:

```
| x = sym.symbols('x')
| sym.series(sym.exp(x), x) # privzete vrednosti x0=0, in n=6
```

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + O(x^6) \quad (4.53)$$

Če želimo definirati drugo izhodišče (`x0=2`) in z več členi (`n=8`), to izvedemo takole:

```
| s1 = sym.series(sym.exp(x), x, x0=2, n=8)
| s1
```

$$e^2 + (x-2)e^2 + \frac{(x-2)^2 e^2}{2} + \frac{(x-2)^3 e^2}{6} + \frac{(x-2)^4 e^2}{24} + \frac{(x-2)^5 e^2}{120} + \frac{(x-2)^6 e^2}{720} + \frac{(x-2)^7 e^2}{5040} + O((x-2)^8; x \rightarrow 2) \quad (4.54)$$

Rezultat vključuje tudi red veljavnosti; na ta način lahko kontroliramo veljavnosti izvajanja ($\mathcal{O}$).

Primer:

```
| s1 = sym.cos(x).series(x, 0, 5)
| s1
```

$$1 - \frac{x^2}{2} + \frac{x^4}{24} + O(x^5) \quad (4.55)$$

```
| s2 = sym.sin(x).series(x, 0, 2)
| s2
```

$$x + O(x^2) \quad (4.56)$$

Izračuna `s1` in `s2` imata različna reda veljavnosti, posledično je produkt:

```
| s1 * s2
```

$$(x + O(x^2)) \left(1 - \frac{x^2}{2} + \frac{x^4}{24} + O(x^5)\right) \quad (4.57)$$

natančen samo do reda $O(x^2)$, kar `sympy` ustrezno obravnava:

```
| s3 = sym.simplify(s1 * s2)
| s3
```

$$x + O(x^2) \quad (4.58)$$

Podatek o stopnji veljavnosti lahko odstranimo:

```
| s3.removeO()
```

$$x \quad (4.59)$$

4.2.11 Linearna algebra

Matrike in vektorji

Matrike in vektorje definiramo s funkcijo `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:

```
| m11, m12, m21, m22 = sym.symbols('m11, m12, m21, m22')
| b1, b2 = sym.symbols('b1, b2')
```

Nato matriko in stolpični vektor:

```
| A = sym.Matrix([[m11, m12], [m21, m22]])
| A
```

$$\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \quad (4.60)$$

```
| b = sym.Matrix([[b1], [b2]])
| b
```

$$\begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \quad (4.61)$$

Sedaj si pogledjmo nekatere tipične operacije; najprej množenje matrike in vektorja:

```
| A * b
```

$$\begin{bmatrix} b_1 m_{11} + b_2 m_{12} \\ b_1 m_{21} + b_2 m_{22} \end{bmatrix} \quad (4.62)$$

Nato skalarni produkt dveh vektorjev (paziti moramo na transponiranje enega od vektorjev):

| `b.T*b`

$$[b_1^2 + b_2^2] \quad (4.63)$$

Determinanta in inverzna matrika:

| `A.det()`

$$m_{11}m_{22} - m_{12}m_{21} \quad (4.64)$$

| `A.inv()`

$$\begin{bmatrix} \frac{m_{22}}{m_{11}m_{22}-m_{12}m_{21}} & -\frac{m_{12}}{m_{11}m_{22}-m_{12}m_{21}} \\ -\frac{m_{21}}{m_{11}m_{22}-m_{12}m_{21}} & \frac{m_{11}}{m_{11}m_{22}-m_{12}m_{21}} \end{bmatrix} \quad (4.65)$$

Množenje in potenca matrike:

| `A*A`

$$\begin{bmatrix} m_{11}^2 + m_{12}m_{21} & m_{11}m_{12} + m_{12}m_{22} \\ m_{11}m_{21} + m_{21}m_{22} & m_{12}m_{21} + m_{22}^2 \end{bmatrix} \quad (4.66)$$

| `A**2`

$$\begin{bmatrix} m_{11}^2 + m_{12}m_{21} & m_{11}m_{12} + m_{12}m_{22} \\ m_{11}m_{21} + m_{21}m_{22} & m_{12}m_{21} + m_{22}^2 \end{bmatrix} \quad (4.67)$$

4.2.12 Reševanje enačb

Enačbe in sistem enačb rešujemo s funkcijo `sympy.solve()` ([dokumentacija](#)). 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.

Sintaksa 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:

```
| x = sym.symbols('x')
| f = sym.sin(x)
| en = sym.Eq(f, 1/2)
| en
```

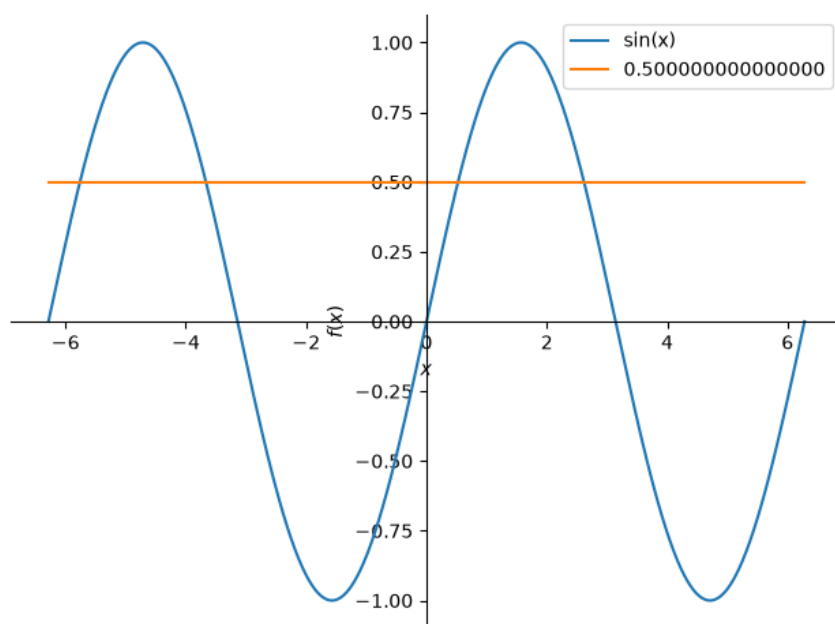
$$\sin(x) = 0.5 \quad (4.68)$$

```
| sym.solve(en, x)
```

$$[0.523598775598299, 2.61799387799149] \quad (4.69)$$

Prikažimo rešitev (opazimo, da smo našli samo dve od neskončno rešitev):

```
| p1 = sym.plotting.plot(sym.sin(x), (x, -2*sym.pi, 2*sym.pi), line_color='C0',
| show=False, legend=True)
| 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()
```



Kvadratna enačba:

```
| a, b, c, x = sym.symbols('a, b, c, x')
| sym.solve(a*x**2 + b*x + c, x)
```

$$\left[\frac{-b - \sqrt{-4ac + b^2}}{2a}, \frac{-b + \sqrt{-4ac + b^2}}{2a} \right] \quad (4.70)$$

Sistem enačb:

```
| x, y = sym.symbols('x y')
| sym.solve([x + y - 1, x - y - 1], [x, y])
```

$$\{x : 1, y : 0\} \quad (4.71)$$

Za nelinearne sisteme pa lahko uporabimo tudi numerično reševanje s funkcijo `sympy.nsolve()` ([dokumentacija](#)):

```
| sympy.nsolve(f, [args,] x0, modules=['mpmath'], **kwargs)
```

kjer so argumenti:

- `f` enačba ali sistem enačb, ki ga rešujemo,
- `args` spremenljivke (opcijsko),

- `x0` začetni približek (skalar ali vektor),
- `modules` paket, ki se uporabi za izračun numerične vrednosti (enaka logika kot pri funkciji `lambdify`, privzet je paket `mpmath`),
- `**kwargs` slovar opcij.

Poglejmo primer od zgoraj:

```
x = sym.symbols('x')
eq = sym.Eq(sym.sin(x), 0.5)
sol = sym.nsolve(eq, x, 3)
sol
```

$$2.61799387799149 \quad (4.72)$$

4.2.13 Reševanje diferencialnih enačb

Nedefinirane funkcije

Preden si ogledamo diferencialne enačbe, moramo spoznati nedefinirane funkcije. Take funkcije definiramo `sympy.Function` [dokumentacija](#).

Kot primer si pogledajmo kako definiramo enačbo $\ddot{x}(t) = g$. Najprej definirajmo nedoločeno funkcijo:

```
| x = sym.Function('x')
```

Nato simbole:

```
| t, g = sym.symbols('t, g')
```

In še enačbo:

```
| sym.Eq(x(t).diff(t,2), g)
```

$$\frac{d^2}{dt^2}x(t) = g \quad (4.73)$$

Diferencialne enačbe

Diferencialne enačbe in sisteme diferencialnih enačb rešujemo s funkcijo `sympy.dsolve()` ([dokumentacija](#)):

```
| 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` diferencialna enačba ali sistem diferencialnih enačb,
- `func` rešitev, ki jo iščemo,
- `ics` začetni in robni pogoji diferencialne enačbe.

Poglejmo si primer mase m , ki drsi po površini s koeficientom trenja μ ; začetna hitrost je v_0 , pomik $x_0 = 0$.

Definirajmo simbole:

```
| x = sym.Function('x')
| t, m, mu, g, v0, x0 = sym.symbols('t m mu g v0 x0', real=True, positive=True)
```

Definirajmo diferencialno enačbo:

```
eq = sym.Eq(m*x(t).diff(t,2), -mu*g*m)
eq
```

$$m \frac{d^2}{dt^2} x(t) = -gm\mu \quad (4.74)$$

Poglejmo lastnosti diferencialne enačbe:

```
sym.ode_order(eq, x(t))
```

$$2 \quad (4.75)$$

```
sym.classify_ode(eq)
```

```
('factorable',
 'nth_algebraic',
 'nth_linear_constant_coeff_undetermined_coefficients',
 'nth_linear_euler_eq_nonhomogeneous_undetermined_coefficients',
 'nth_linear_constant_coeff_variation_of_parameters',
 'nth_linear_euler_eq_nonhomogeneous_variation_of_parameters',
 'nth_algebraic_Integral',
 'nth_linear_constant_coeff_variation_of_parameters_Integral',
 'nth_linear_euler_eq_nonhomogeneous_variation_of_parameters_Integral',
 '2nd_nonlinear_autonomous_conserved',
 '2nd_nonlinear_autonomous_conserved_Integral')
```

Rešimo jo:

```
rešitev = sym.dsolve(eq, x(t),
                    ics={
                        x(0):0, #pri času 0s je pomik nič
                        x(t).diff(t).subs(t, 0): v0 #pri času 0s je hitrost enaka v0
                    })
rešitev
```

$$x(t) = -\frac{g\mu t^2}{2} + tv_0 \quad (4.76)$$

Desno stran enačbe prikličemo takole (`rhs` - right-hand side):

```
rešitev.rhs
```

$$-\frac{g\mu t^2}{2} + tv_0 \quad (4.77)$$

4.3 Dodatno

4.4 sympy.mechanics

`sympy` ima vgrajeno podporo za klasično mehaniko ([dokumentacija](#)). Celovit tutorial je bil prikazan na znanstveni konferenci [SciPy 2016](#).

Poglavje 5

Uvod v numerične metode in sistemi linearnih enačb

5.1 Uvod v numerične metode

Kadar želimo simulirati izbrani fizikalni proces, ponavadi postopamo takole:

1. postavimo matematični model*,
2. izberemo numerično metodo in njene parametre,
3. pripravimo program (pomagamo si z vgrajenimi funkcijami),
4. 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 števki.

Število z dvojno natančnostjo se v Pythonu imenuje **float64** in je zapisano v spomin v binarni obliki v 64 bitih (1 bit za predznak, 11 bitov za eksponent in 52 bitov za mantiso). 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](#)):

```
import sys
sys.float_info.epsilon
1e16+2
#sys.float_info #preverite tudi širši izpis!

1.0000000000000002e+16
```

Poleg števila z dvojno natančnostjo se uporabljajo drugi tipi podatkov; dober pregled različnih tipov je prikazan v okviru **numpy** in **python** dokumentacije.

Tukaj si pogledjmo 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:

```
import numpy as np
število = np.int8(1) # poskušate še števila: -128 in nato 127, 128, 129. Kaj se
dogaja?
f'Število {število} tipa {type(število)} zapisano v binarni obliki:{število:8b}'

"Število 1 tipa <class 'numpy.int8'> zapisano v binarni obliki:      1"
```

5.1.2 Napaka metode

Poleg zaokrožitvene napake pa se pogosto srečamo tudi z **napako metode** ali **metodično napako**, 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 $|e_1| \gg |e_2|$ 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 ($E_i, i = 0, 1, \dots, m-1$) z n neznankami ($x_j, j = 0, 1, \dots, n-1$):

$$\begin{array}{ccccccc} E_0 : & A_{0,0} x_0 & + & A_{0,1} x_1 & + & \dots & + & A_{0,n-1} x_{n-1} & = & b_0 \\ E_1 : & A_{1,0} x_0 & + & A_{1,1} x_1 & + & \dots & + & A_{1,n-1} x_{n-1} & = & b_1 \\ & \vdots & & & & & & \vdots & & \\ E_{m-1} : & A_{m-1,0} x_0 & + & A_{m-1,1} x_1 & + & \dots & + & A_{m-1,n-1} x_{n-1} & = & b_{m-1}. \end{array} \quad (5.1)$$

Koeficienti $A_{i,j}$ in b_i so znana števila.

V kolikor je desna stran enaka nič, torej $b_i = 0$, imenujemo sistem **homogen**, sicer je sistem **nehomogen**.

Sistem enačb lahko zapišemo tudi v matrični obliki:

$$\mathbf{A} \mathbf{x} = \mathbf{b} \quad (5.2)$$

kjer sta $\mathbf{A}$ in $\mathbf{b}$ znana matrika in vektor, vektor $\mathbf{x}$ pa ni znan. Matriko $\mathbf{A}$ imenujemo **matrika koeficientov**, vektor $\mathbf{b}$ **vektor konstant** (tudi: vektor prostih členov ali vektor stolpec desnih strani) in $\mathbf{x}$ **vektor neznank**. Če matriki $\mathbf{A}$ dodamo kot stolpec vektor $\mathbf{b}$, 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.: $\mathbf{a}$,
- matrične spremenljivke pišemo z veliko črko poudarjeno, npr.: $\mathbf{A}$.

5.2.1 O rešitvi sistema linearnih enačb

```
from IPython.display import YouTubeVideo
YouTubeVideo('7YbyijGUbYw', width=800, height=300)
```



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 $\mathbf{A}$ jo lahko preoblikujemo v t. i. **vrstično kanonično obliko**:

1. če obstajajo ničelne vrstice, so te na dnu matrike,
2. prvi neničelni element se nahaja desno od prvih neničelnih elementov predhodnih vrstic,
3. prvi neničelni element v *vrstici* imenujemo **pivot** in je enak 1,
4. pivot je edini neničelni element v *stolpcu*.

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**. Rang matrike je torej enak številu linearno neodvisnih vrstic matrike. Transponiranje matrike njenega ranga ne spremeni, zato je rang matrike enak tudi številu linearno neodvisnih *stolpcev* matrike.

Primer preoblikovanja matrike $\mathbf{A}$:

```
import numpy as np #uvozimo numpy
A = np.arange(9).reshape((3,3))+1
A
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 $A[1,:]-4A[0,:]$:

```
| A[1,:] -= A[1,0]*A[0,:]
| A
```

```
| array([[ 1,  2,  3],
|        [ 0, -3, -6],
|        [ 7,  8,  9]])
```

Podobno naredimo za tretjo vrstico:

```
| A[2,:] -= A[2,0]*A[0,:]
| A
```

```
| array([[ 1,  2,  3],
|        [ 0, -3, -6],
|        [ 0, -6, -12]])
```

Drugo vrstico sedaj delimo z $A[1,1]$, da dobimo pivot:

```
| A[1,:] = A[1,:]/A[1,1]
| A
```

```
| 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:

```
| A[0,:] -= A[0,1]*A[1,:] # odštevanje od prve vrstice
| A
```

```
| array([[ 1,  0, -1],
|        [ 0,  1,  2],
|        [ 0, -6, -12]])
```

```
| A[2,:] -= A[2,1]*A[1,:] # odštevanje od zadnje vrstice
| A
```

```
| 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.

```
| A
```

```
| 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](#)):

```
| 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.

```
| np.linalg.matrix_rank(A)
```

```
| np.int64(2)
```

Če velja $r = \text{rang}(\mathbf{A}) = \text{rang}([\mathbf{A}|\mathbf{b}])$, potem rešitev **obstaja** (rečemo tudi, da je sistem **konsistenten**).

Konsistenten sistem ima:

- natanko eno rešitev, ko je število neznank n enako rang r (rešitev je neodvisna) 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 $n = r$:

$$\mathbf{A} \mathbf{x} = \mathbf{b}. \quad (5.3)$$

Pod zgornjimi pogoji je matrika koeficientov $\mathbf{A}$ nesingularna ($|\mathbf{A}| \neq 0$) in sistem ima rešitev:

$$\mathbf{x} = \mathbf{A}^{-1} \mathbf{b}. \quad (5.4)$$

Poglejmo si primer sistema, ko so **enačbe linearno odvisne** ($r < n$):

```
| A = np.array([[1 , 2],
|               [2, 4]])
| b = np.array([1, 2])
| Ab = np.column_stack((A,b))
| Ab
|
| array([[1, 2, 1],
|        [2, 4, 2]])
```

S pomočjo numpy knjižnice pogledajmo sedaj rang matrike koeficientov in razširjene matrike ter determinanto z uporabo `numpy.linalg.det` ([dokumentacija](#)):

```
| det(a)
```

kjer je $\mathbf{a}$ matrika (ali seznam matrik), katere determinanto iščemo; funkcija `det` vrne determinanto (ali seznam determinant).

```
| f'rang(A)={np.linalg.matrix_rank(A)}, rang(Ab)={np.linalg.matrix_rank(Ab)}, \
| število neznank: {A.shape[1]}, det(A)={np.linalg.det(A)}'
|
| 'rang(A)=1, rang(Ab)=1, število neznank: 2, det(A)=0.0'
```

Poglejmo še primer, ko **rešitve sploh ni** (nekonsistenten sistem):

```
| A = np.array([[1 , 2],
|               [2, 4]])
| b = np.array([1, 1])
| Ab = np.column_stack((A,b))
| Ab
|
| array([[1, 2, 1],
|        [2, 4, 1]])
|
| f'rang(A)={np.linalg.matrix_rank(A)}, rang(Ab)={np.linalg.matrix_rank(Ab)}, \
| število neznank: {A.shape[1]}, det(A)={np.linalg.det(A)}'
|
| 'rang(A)=1, rang(Ab)=2, število neznank: 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):

$$\text{cond}(\mathbf{A}) = \|\mathbf{A}\| \|\mathbf{A}^{-1}\|. \quad (5.5)$$

Z $\|\mathbf{A}\|$ je označena **norma** matrike.

Obstaja več načinov računanja norme; navedimo dve:

- Evklidska norma (tudi Frobeniusova):

$$\|\mathbf{A}\|_e = \sqrt{\sum_{i=1}^n \sum_{j=1}^n A_{ij}^2} \quad (5.6)$$

- Norma vsote vrstic ali tudi neskončna norma:

$$\|\mathbf{A}\|_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^n |A_{ij}| \quad (5.7)$$

Pogojenost računamo z vgrajeno funkcijo `numpy.linalg.cond` ([dokumentacija](#)):

```
| cond(x, p=None)
```

ki sprejme dva parametra: matriko `x` in opsijski tip norme `p` (privzeti tip je `None`; v tem primeru se uporabi 2-norma (razmerje največje in najmanjše singularne vrednosti); za Frobeniusovo normo je treba podati `p='fro'`).

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.

Primer slabo pogojene matrike

Pogledali si bomo slabo pogojen sistem, kjer bomo z malenkostno spremembo na matriki koeficientov povzročili veliko spremembo rešitve.

Matrika koeficientov:

```
| A = np.array([[1, 1],
|               [1, 1.00001]])
| np.linalg.cond(A)
| np.float64(400002.0000028382)
```

Vektor konstant:

```
| b = np.array([3, -3])
| Ab = np.column_stack((A,b))
| Ab
| array([[ 1.      ,  1.      ,  3.      ],
|        [ 1.      , 1.00001, -3.      ]])
```

Preverimo rang in determinanto:

```
f'rang(A)={np.linalg.matrix_rank(A)}, rang(Ab)={np.linalg.matrix_rank(Ab)}, \
število neznank: {A.shape[1]}, det(A)={np.linalg.det(A)}'
```

```
'rang(A)=2, rang(Ab)=2, število neznank: 2, det(A)=1.000000000006551e -05'
```

Od druge enačbe odštejemo prvo:

```
Ab[1,:] -= Ab[0,:]
Ab
```

```
array([[ 1.e+00,  1.e+00,  3.e+00],
       [ 0.e+00,  1.e -05, -6.e+00]])
```

Določimo x_1 :

```
x1 = Ab[1,2]/Ab[1,1]
x1
```

```
np.float64( -599999.9999960692)
```

Preostane še določitev x_0 :

```
x0 = (Ab[0,2] - Ab[0,1]*x1)/Ab[0,0]
x0
```

```
np.float64(600002.9999960692)
```

Malenkostno spremenimo matriko koeficientov in ponovimo reševanje:

```
A = np.array([[1 , 1],
              [1, 1.0001]]) # <= tukaj smo spremenili 1.00001 na 1.0001
np.linalg.cond(A)
```

```
np.float64(40002.00007491187)
```

Ponovimo izračun:

```
Ab = np.column_stack((A,b))
f'rang(A)={np.linalg.matrix_rank(A)}, rang(Ab)={np.linalg.matrix_rank(Ab)}, \
število neznank: {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:

```
[x0, x1] # prva rešitev
```

```
[np.float64(600002.9999960692), np.float64( -599999.9999960692)]
```

```
[x0_, x1_] # druga rešitev
```

```
[np.float64(60003.00000000661), np.float64( -60000.00000000661)]
```

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:

$$[\mathbf{A}|\mathbf{b}] = \left[\begin{array}{cccc|c} A_{0,0} & A_{0,1} & \cdots & A_{0,n-1} & b_0 \\ A_{1,0} & A_{1,1} & \cdots & A_{1,n-1} & b_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ A_{n-1,0} & A_{n-1,1} & \cdots & A_{n-1,n-1} & b_{n-1} \end{array} \right] \quad (5.8)$$

Gaussovo eliminacijo si bomo pogledali na zgledu:

```
A = np.array([[ 8., -6, 3],
              [-6, 6, -6],
              [ 3, -6, 6]])
b = np.array([-14, 36, 6])
Ab = np.column_stack((A,b))

Ab

array([[ 8., -6.,  3., -14.],
       [-6.,  6., -6.,  36.],
       [ 3., -6.,  6.,  6.]])
```

Korak 0: prvo vrstico pomnožimo z $\text{Ab}[1,0]/\text{Ab}[0,0]=-6/8$ in odštejemo od druge:

```
Ab[1,:] -= Ab[1,0]/Ab[0,0] * Ab[0,:]
Ab

array([[ 8. , -6. ,  3. , -14. ],
       [ 0. ,  1.5, -3.75, 25.5 ],
       [ 3. , -6. ,  6. ,  6. ]])
```

Nato prvo vrstico pomnožimo z $\text{Ab}[2,0]/\text{Ab}[0,0]=3/8$ in odštejemo od tretje:

```
Ab[2,:] -= Ab[2,0]/Ab[0,0] * Ab[0,:]
Ab

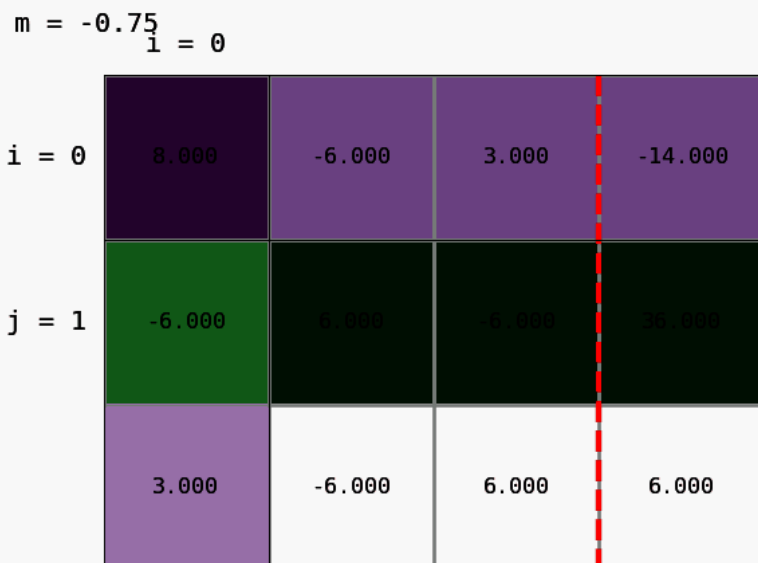
array([[ 8. , -6. ,  3. , -14. ],
       [ 0. ,  1.5, -3.75, 25.5 ],
       [ 0. , -3.75,  4.875, 11.25 ]])
```

Korak 1: drugo vrstico pomnožimo z $\text{Ab}[2,1]/\text{Ab}[1,1]=-3.75/1.5$ in odštejemo od tretje:

```
Ab[2,:] -= Ab[2,1]/Ab[1,1] * Ab[1,:]
Ab
```

```
array([[ 8. , -6. ,  3. , -14. ],
       [ 0. ,  1.5, -3.75, 25.5 ],
       [ 0. ,  0. , -4.5 , 75. ]])
```

S pomočjo pripravljenega modula `moduli/gauss_prikaz.py` lahko postopek tudi animiramo:



Dobili smo zgornje trikotno matriko in Gaussova eliminacija je končana. Lahko izračunamo rešitev, torej določimo vektor neznank x z **obratnim vstavljanjem**.

Iz zadnje vrstice zgornje trikotne matrike izračunamo x_2 :

```
Ab
array([[ 8. , -6. ,  3. , -14. ],
       [ 0. ,  1.5, -3.75, 25.5 ],
       [ 0. ,  0. , -4.5 , 75. ]])

x = np.zeros(3) #pripravimo prazen seznam
x[2] = Ab[2, -1]/Ab[2,2]
x
array([ 0. ,  0. , -16.66666667])
```

S pomočjo predzadnje vrstice izračunamo x_1 :

```
Ab
array([[ 8. , -6. ,  3. , -14. ],
       [ 0. ,  1.5, -3.75, 25.5 ],
       [ 0. ,  0. , -4.5 , 75. ]])

x[1] = (Ab[1, -1] - Ab[1,2]*x[2]) / Ab[1,1]
x
array([ 0. , -24.66666667, -16.66666667])
```

S pomočjo prve vrstice nato izračunamo x_0 :

```
Ab
array([[ 8. , -6. ,  3. , -14. ],
       [ 0. ,  1.5, -3.75, 25.5 ],
       [ 0. ,  0. , -4.5 , 75. ]])
```

```

x[0] = (Ab[0,3] - Ab[0,1:3]@x[1:]) / Ab[0,0]
x
array([ -14.          , -24.66666667, -16.66666667])

Preverimo rešitev:

A @ x - b
array([ -3.55271368e -15,  7.10542736e -15,  7.10542736e -15])

```

Povzetek Gaussove eliminacije

V modul `orodja.py` shranimo funkciji:

```

def gaussova_eliminacija(A, b, prikazi_korake = False):
    Ab = np.column_stack((A, b))
    for p, pivot_vrstica in enumerate(Ab[: -1]):
        for vrsta in Ab[p+1:]:
            if pivot_vrstica[p]:
                vrsta[p:] -= pivot_vrstica[p:] * vrsta[p] / pivot_vrstica[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_vrstica in enumerate(Ub[: -1]):
        x[v - p - 1] = (pivot_vrstica[ -1] - pivot_vrstica[v - p: -1] @ x[v - p: ] ) /
        (pivot_vrstica[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*).

```

Ub = gaussova_eliminacija(A, b, prikazi_korake=False)
Ub
array([[ 8.  , -6.  ,  3.  , -14.  ],
       [ 0.  ,  1.5 , -3.75, 25.5 ],
       [ 0.  ,  0.  , -4.5 , 75.  ]])

gaussova_el_resitev(Ub)
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 skupaj približno $n^3/2$ operacij, torej dodatnih približno $n^3/6$ (kar se šteje kot glavna slabost te metode).

5.3.2 Uporaba knjižnice numpy

Reševanje sistema linearnih enačb z `numpy.linalg.solve` ([dokumentacija](#)):

```
| 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.

```
| np.linalg.solve(A, b)
```

```
| array([ -14.          , -24.66666667, -16.66666667])
```

5.4 Dodatno

5.5 Primer simbolnega reševanja sistema linearnih enačb v okviru sympy

```
| import sympy as sym
| sym.init_printing()

| 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]])

| eq = sym.Eq(A*x, b)
| eq
```

$$\begin{bmatrix} A_{11}x_1 + A_{12}x_2 \\ A_{21}x_1 + A_{22}x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \quad (5.9)$$

```
| %%timeit
| resitev = sym.solve(eq, [x1, x2])
| resitev
```

$$\left\{ x_1 : \frac{-A_{12}b_2 + A_{22}b_1}{A_{11}A_{22} - A_{12}A_{21}}, x_2 : \frac{A_{11}b_2 - A_{21}b_1}{A_{11}A_{22} - A_{12}A_{21}} \right\} \quad (5.10)$$

```
| A.det()
```

$$A_{11}A_{22} - A_{12}A_{21} \quad (5.11)$$

Poglavje 6

Sistemi linearnih enačb - nadaljevanje

6.1 Razcep LU

Za rešitev sistema linearnih enačb zahteva Gaussov eliminacijski postopek najmanjše število računskih operacij.

V primeru, ko se matrika koeficientov $\mathbf{A}$ ne spreminja in se spreminja zgolj vektor konstant $\mathbf{b}$, se je mogoče izogniti ponovni Gaussovi eliminaciji matrike koeficientov. Z razcepom matrike $\mathbf{A}$ lahko pridemo do rešitve z manj računskimi operacijami. V ta namen si bomo pogledali razcep LU!

Poljubno matriko lahko zapišemo kot produkt dveh matrik:

$$\mathbf{A} = \mathbf{B} \mathbf{C}. \quad (6.1)$$

Pri tem je možnosti za zapis matrik $\mathbf{B}$ in $\mathbf{C}$ neskončno veliko.

Pri razcepu LU zahtevamo, da je matrika $\mathbf{B}$ spodnje trikotna in matrika $\mathbf{C}$ zgornje trikotna:

$$\mathbf{A} = \mathbf{L} \mathbf{U}. \quad (6.2)$$

Vsaka od matrik $\mathbf{L}$ in $\mathbf{U}$ ima $(n+1)n/2$ neničelnih elementov; skupaj torej $n^2 + n$ neznank. Znana matrika $\mathbf{A}$ definira n^2 vrednosti. Za enolično določitev matrik $\mathbf{L}$ in $\mathbf{U}$ torej manjka n enačb. Tukaj bomo uporabili razcep LU, ki dodatne enačbe pridobi s pogojem $L_{ii} = 1, i = 0, 1, \dots, n-1$.

Sistem linearnih enačb:

$$\mathbf{A} \mathbf{x} = \mathbf{b} \quad (6.3)$$

torej zapišemo z razcepom matrike $\mathbf{A}$:

$$\mathbf{L} \underbrace{\mathbf{U} \mathbf{x}}_{\mathbf{y}} = \mathbf{b}. \quad (6.4)$$

Do rešitve sistema $\mathbf{A} \mathbf{x} = \mathbf{b}$ sedaj pridemo tako, da rešimo dva trikotna sistema enačb.

Najprej izračunamo vektor $\mathbf{y}$:

$$\mathbf{L} \mathbf{y} = \mathbf{b}. \quad (\text{direktno vstavljanje}) \quad (6.5)$$

Ko je $\mathbf{y}$ izračunan, lahko iz:

$$\mathbf{U} \mathbf{x} = \mathbf{y} \quad (\text{obratno vstavljanje}) \quad (6.6)$$

določimo $\mathbf{x}$.

6.1.1 Razcep LU matrike koeficientov $\mathbf{A}$

V nadaljevanju bomo pokazali, da Gaussova eliminacija dejansko predstavlja razcep LU matrike koeficientov $\mathbf{A}$. Pri tem si bomo pomagali s simbolnim izračunom, zato uvozimo paket `sympy`:

```
import sympy as sym # uvozimo sympy
sym.init_printing() # za lep prikaz izrazov
```

Prikaz začnimo na primeru simbolno zapisanih matrik $\mathbf{L}$ in $\mathbf{U}$ dimenzije 3×3 :

```
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 $\mathbf{A}$ zapisana z elementi matrik $\mathbf{L}$ in $\mathbf{U}$ torej je:

```
A = L*U
A
```

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ L_{21}U_{11} & L_{21}U_{12} + U_{22} & L_{21}U_{13} + U_{23} \\ L_{31}U_{11} & L_{31}U_{12} + L_{32}U_{22} & L_{31}U_{13} + L_{32}U_{23} + U_{33} \end{bmatrix} \quad (6.7)$$

Izvedimo sedaj Gaussovo eliminacijo nad matriko koeficientov $\mathbf{A}$.

S pomočjo prve vrstice izvedemo Gaussovo eliminacijo v prvem stolpcu:

```
A[1,:] -= L21 * A[0,:]
A[2,:] -= L31 * A[0,:]
A
```

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & L_{32}U_{22} & L_{32}U_{23} + U_{33} \end{bmatrix} \quad (6.8)$$

Nadaljujemo v drugem stolpcu:

```
A[2,:] -= L32 * A[1,:]
A
```

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix} \quad (6.9)$$

Iz zgornje eliminacije ugotovimo:

1. matrika $\mathbf{U}$ je enaka matriki, ki jo dobimo, če izvedemo Gaussovo eliminacijo nad matriko koeficientov $\mathbf{A}$.
- izven diagonalni členi $\mathbf{L}$ 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:

```
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 , shranimo v matriko L na mesto z indeksi, kot jih ima v matriki A eliminirani element.

```
(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: {}'.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 ]]
```

L

```
array([[ 0.  ,  0.  ,  0.  ],
       [-0.75,  0.  ,  0.  ],
       [ 0.375, -2.5 ,  0.  ]])
```

Dopolnimo diagonalo L :

```
for i in range(v):
    L[i, i] = 1.
```

L

```
array([[ 1.  ,  0.  ,  0.  ],
       [-0.75,  1.  ,  0.  ],
       [ 0.375, -2.5 ,  1.  ]])
```

Sedaj rešimo spodnje trikotni sistem enačb $Ly = b$:

```
## direktno vstavljanje
y = np.zeros_like(b)
for i, b_ in enumerate(b):
    y[i] = (b_ - np.dot(L[i, :i], y[:i]))
```

y

```
| array([ -14. ,  25.5,  75. ])
```

Nadaljujemo z reševanjem zgornje trikotnega sistema $\mathbf{U} \mathbf{x} = \mathbf{y}$:

```
| U
```

```
| array([[ 8. , -6. ,  3. ],
|        [ 0. ,  1.5, -3.75],
|        [ 0. ,  0. , -4.5 ]])
```

```
| y
```

```
| array([ -14. ,  25.5,  75. ])
```

```
| ## 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
```

```
| array([ -14.          , -24.66666667, -16.66666667])
```

Kakor smo navedli zgoraj, ob spremembi vektorja konstant $\mathbf{b}$ ponovna Gaussova eliminacija ni potrebna. Izvesti je treba samo direktno in nato obratno vstavljanje. Poglejmo primer:

```
| 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
```

```
| array([ -4.33333333, -7.88888889, -4.55555556])
```

```
| A.dot(x)
```

```
| array([ -1.,  6.,  7.])
```

6.2 Pivotiranje

Poglejmo si spodnji sistem enačb:

```
| A = np.array([[0, -6, 6],
|               [-6, 6, -6],
|               [8, -6, 3]], dtype=float) # poskusite tukaj izpustiti dtype=float in
|               preverite rezultat
| b = np.array([6, 36, -14], dtype=float)
```

Če bi izvedli Gaussovo eliminacijo v prvem stolpcu matrike $\mathbf{A}$:

```
| A[1,:] - A[1,0]/A[0,0] * A[0,:]
```

```
| /tmp/ipykernel_342475/3999418912.py:1: RuntimeWarning: divide by zero encountered in
| scalar divide
```

```
| A[1,:] - A[1,0]/A[0,0] * A[0,:]
```

```
| /tmp/ipykernel_342475/3999418912.py:1: RuntimeWarning: invalid value encountered in
| multiply
```

```
| A[1,:] - A[1,0]/A[0,0] * A[0,:]
```

```
| 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 $\mathbf{A}$ diagonalno dominantna, pivotiranje po vrsticah ni potrebno. Reševanje je brez pivotiranja numerično stabilno.

Kvadratna matrika $\mathbf{A}$ dimenzije n je **diagonalno dominantna**, če je absolutna vrednost diagonalnega elementa vsake vrstice večja od vsote absolutnih vrednosti ostalih elementov v vrstici:

$$|A_{ii}| > \sum_{j=1, j \neq i}^n |A_{ij}| \quad (6.10)$$

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 kolikor 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:

```
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_vrstica = Ab[p, :]
        for vrstica in Ab[p + 1:]:
            if pivot_vrstica[p]:
                vrstica[p:] -= pivot_vrstica[p] * vrstica[p] / pivot_vrstica[p]
        if prikazi_korake:
            print('Korak: {}'.format(p))
            print('Pivot vrstica:', pivot_vrstica)
            print(Ab)
    return Ab
```

```

A

array([[ 0., -6.,  6.],
       [-6.,  6., -6.],
       [ 8., -6.,  3.]])

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.

```

def LU_razcep_pivotiranje(A, prikazi_korake=False):
    """ Vrne razcep LU matriko in vektor zamenjanih vrstic pivotiranja,
    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: {}'.format(p))
            print('Pivot vrsta:', pivot_vrsta)
            print(LU)
    return LU, pivotiranje

```

Poglejmo si primer:

```

A = np.array([[0, -6, 6],
              [-6, 6, -6],
              [8, -6, 3]], dtype=float)
b = np.array([-14, 36, 6], dtype=float)

```

```
| 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 kompakten 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 $\mathbf{lu}$ tako vsebuje $3 \times 3 = 9$ elementov:

```
| lu

| array([[ 8.  , -6.  ,  3.  ],
|        [ 0.  , -6.  ,  6.  ],
|        [-0.75, -0.25, -2.25]])
```

Na diagonali in nad diagonalo so vrednosti zgornje trikotne matrike $\mathbf{U}$, pod diagonalo pa so poddiagonalni elementi matrike $\mathbf{L}$. Pri izračunu rešitve bomo upoštevali, da so diagonalne vrednosti $\mathbf{L}$ enake 1.

Numerični seznam `piv` nam pove, kako so bile zamenjane vrstice, kar je treba upoštevati pri izračunu rešitve:

```
| piv

| array([2, 0, 1])

| Določimo sedaj še rešitev (koda za računanje rešitve je v modulu orodja.py)

| from moduli import orodja

| r = orodja.LU_resitev_pivotiranje(lu, b, piv)
| r

| array([-3.66666667, -14.11111111, -16.44444444])
```

Preverjanje rešitve:

```
| A@r

| 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](#). Za odličen uvod v SciPy si lahko ogledate [YouTube posnetek: SciPy Tutorial \(2022\): For Physicists, Engineers, and Mathematicians](#).

Kratek pregled hierarhije modula:

- Linearna algebra ([scipy.linalg](#))
- Integracija ([scipy.integrate](#))

- Optimizacija ([scipy.optimize](#))
- Interpolacija ([scipy.interpolate](#))
- Fourierjeva transformacija ([scipy.fft](#))
- Redke matrike ([scipy.sparse](#))
- Statistika ([scipy.stats](#))
- Procesiranje signalov ([scipy.signal](#))
- Posebne funkcije ([scipy.special](#))
- Večdimenzijsko procesiranje slik ([scipy.ndimage](#))
- Delo z datotekami ([scipy.io](#))

V sledečih predavanjih si bomo nekatere podmodule podrobneje pogledali.

Poglejmo si, kako je znotraj SciPy implementiran *razcep LU*:

```
| from scipy.linalg import lu_factor, lu_solve
|
| Funkcija scipy.linalg.lu_factor (dokumentacija):
|
| 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 prepíše vrednost *a* (to je lahko pomembno, da se prihrani spomin in poveča hitrost). Funkcija *lu_factor* vrne terko (*lu*, *piv*):

- *lu* - $\mathbf{L} \setminus \mathbf{U}$ matrika (matrika, ki je enake dimenzije kot *a*, vendar pod diagonalo vsebuje elemente $\mathbf{L}$, preostali elementi pa definirajo $\mathbf{U}$; diagonalni elementi $\mathbf{L}$ 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](#)), 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:

```
| A = np.array([[0, -6, 6],
|               [-6, 6, -6],
|               [8, -6, 3]], dtype=float)
| b = np.array([-14, 36, 6], dtype=float)
|
| lu, piv = lu_factor(A)
|
| lu
|
| array([[ 8., -6.,  3. ],
|        [ 0., -6.,  6. ],
|        [-0.75, -0.25, -2.25]])
|
| piv
|
| array([2, 2, 2], dtype=int32)
```

Pridobimo rešitev:

```
| r = lu_solve((lu, piv), b)
| r
|
| array([-3.66666667, -14.11111111, -16.44444444])
```

Preverimo ustreznost rešitve:

```
| A@r
|
| 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

$$\mathbf{A} \mathbf{A}^{-1} = \mathbf{A}^{-1} \mathbf{A} = \mathbf{I}, \quad (6.11)$$

kjer je $\mathbf{I}$ enotska matrika.

Najbolj učinkovit način za izračun inverzne matrike od matrike $\mathbf{A}$ je rešitev matrične enačbe:

$$\mathbf{A} \mathbf{X} = \mathbf{I}. \quad (6.12)$$

Matrika $\mathbf{X}$ je inverzna matriki $\mathbf{A}$: $\mathbf{A}^{-1} = \mathbf{X}$.

Izračun inverzne matrike je torej enak reševanju n sistemov n linearnih enačb:

$$\mathbf{A} \mathbf{x}_i = \mathbf{b}_i, \quad i = 0, 1, 2, \dots, n-1, \quad (6.13)$$

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 $\mathbf{x}_i$ ($2n^2$ računskih operacij za vsak i). Skupni računski obseg je torej še vedno reda n^3 . (Če bi n krat izvajali Gaussovo eliminacijo, bi bil obseg reda n^4 .)

```
lu_piv = lu_factor(A)
lu_piv

(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:

```
np.identity(len(A))

array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Nato rešimo sistem enačb za vsak stolpec enotske matrike:

```
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:

```
A_inv

array([[ -0.16666667, -0.16666667,  0.          ],
       [-0.27777778, -0.44444444, -0.33333333],
       [-0.11111111, -0.44444444, -0.33333333]])
```

Preverimo rešitev:

```
A@A_inv
```

```
array([[ 1.00000000e+00,  0.00000000e+00,  1.11022302e -16],
       [ 2.22044605e -16,  1.00000000e+00, -1.11022302e -16],
       [-1.66533454e -16,  0.00000000e+00,  1.00000000e+00]])
```

Rešitev z uporabo Numpy:

```
#%%timeit
np.linalg.inv(A)
```

```
array([[ -0.16666667, -0.16666667,  0.          ],
       [-0.27777778, -0.44444444, -0.33333333],
       [-0.11111111, -0.44444444, -0.33333333]])
```

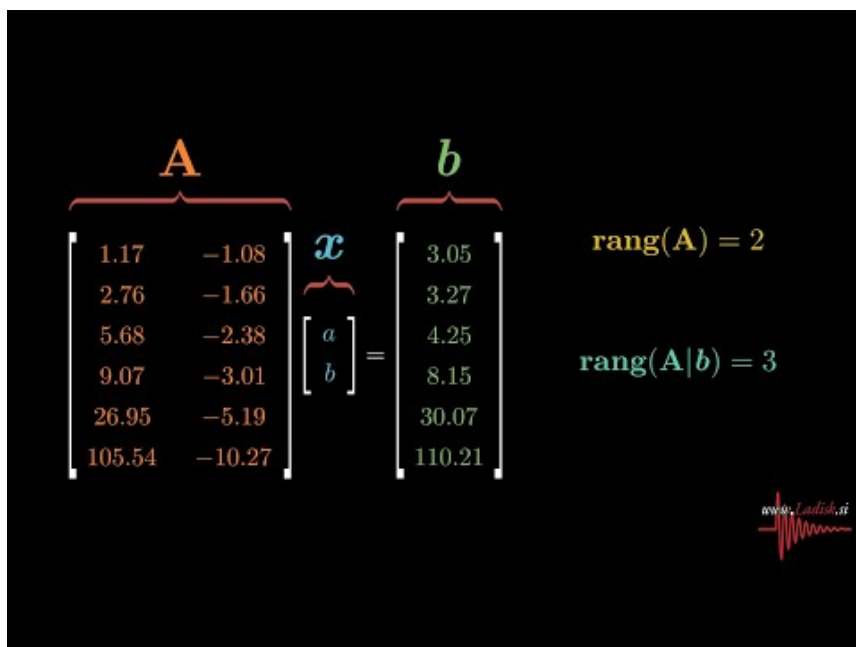
Z ustrezno uporabo funkcij iz modula SciPy lahko do rešitve pridemo še hitreje. V funkcijo `lu_solve` lahko vstavimo vektor **b** ali matriko **B**, katere posamezni stolpec *i* predstavlja nov vektor konstant **b_i**.

```
#%%timeit
lu_solve(lu_piv, np.identity(len(A)))
```

```
array([[ -0.16666667, -0.16666667,  0.          ],
       [-0.27777778, -0.44444444, -0.33333333],
       [-0.11111111, -0.44444444, -0.33333333]])
```

6.5 Reševanje predoločenih sistemov

```
from IPython.display import YouTubeVideo
YouTubeVideo('N6upu99LbfQ', width=800, height=300)
```



Kadar rešujemo sistem m linearnih enačb z n neznankami ter velja $m > n$ in je rang razširjene matrike $[A | b]$ enak $n + 1$, imamo predoločeni (nekonsistenten) sistem.

Predoločeni (tudi **nekonsistenten sistem**):

$$A x = b, \quad (6.14)$$

nima rešitve. Lahko pa poiščemo najboljši približek rešitve z metodo **najmanjših kvadratov**.

Vsota kvadratov preostankov je definirana s skalarnim produktom:

$$\|r\|^2 = (\mathbf{A} \mathbf{x} - \mathbf{b})^T (\mathbf{A} \mathbf{x} - \mathbf{b}) \quad (6.15)$$

kar preoblikujemo v:

$$\|r\|^2 = \mathbf{x}^T \mathbf{A}^T \mathbf{A} \mathbf{x} - 2\mathbf{b}^T \mathbf{A} \mathbf{x} + \mathbf{b}^T \mathbf{b}, \quad (6.16)$$

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:

$$\nabla_x \|r\|^2 = 2\mathbf{A}^T \mathbf{A} \mathbf{x} - 2\mathbf{A}^T \mathbf{b} = 0 \quad (6.17)$$

Tako iz normalne enačbe:

$$\mathbf{A}^T \mathbf{A} \mathbf{x} = \mathbf{A}^T \mathbf{b} \quad (6.18)$$

določimo najboljši približek rešitve:

$$\mathbf{x} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{b}. \quad (6.19)$$

Z vpeljavo **psevdo inverzne matrike**:

$$\mathbf{A}^+ = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \quad (6.20)$$

k matriki $\mathbf{A}$ je rešitev predoločenega sistema zapisana:

$$\mathbf{x} = \mathbf{A}^+ \mathbf{b}. \quad (6.21)$$

Zgoraj predstavljen postopek je relativno enostaven, je pa numerično zahteven in lahko v nekaterih primerih slabo pogojen; priporočeno je, da v praksi psevdo inverzno matriko izračunamo z uporabo funkcij:

- `numpy.linalg.pinv` iz modula `numpy` ([dokumentacija](#)),
- Funkcij `pinv` ali `pinvh` iz modula `scipy.linalg` (izbira je odvisna od obravnavanega problema; glejte [dokumentacijo](#)),

ki temeljijo na boljših numerični metodah.

Primer sistema z enolično rešitvijo:

```
## število enačb enako številu neznank
A = np.array([[1., 2],
              [2, 3]])
b = np.array([5., 8])
np.linalg.solve(A, b)

array([1., 2.]
```

Naredimo sedaj predoločeni sistem (funkcija `numpy.vstack` ([dokumentacija](#)) zloži sezname kot vrstice, `numpy.random.seed` ([dokumentacija](#)) ponastavi generator naključnih števil na vrednost semena `seed`, `numpy.random.normal` ([dokumentacija](#)) pa generira normalno porazdeljeni seznam dolžine `size` in standardne deviacije `scale`):

```
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
```

```
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]])

vb = np.hstack([b,b,b])
vb += np.random.normal(scale=0.01, size=vb.shape)
vb # vektor konstant

array([5.00761038, 8.00121675, 5.00443863, 8.00333674, 5.01494079,
       7.99794842])
```

Rešimo sedaj predoločen sistem:

```
Ap = np.linalg.pinv(vA)
Ap.dot(vb)

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:

```
%%timeit # preverite hitrost!
Ap2 = np.linalg.inv(vA.T@vA) @ vA.T

Ap2 - Ap

array([[ 4.44089210e -16,  3.44169138e -15, -1.44328993e -15,
         3.88578059e -15, -2.66453526e -15,  3.21964677e -15],
       [ 4.44089210e -16, -1.05471187e -15,  1.55431223e -15,
        -9.99200722e -16,  2.10942375e -15, -1.38777878e -15]])
```

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-Seidelova metoda

V nadaljevanju si bomo pogledali idejo *Gauss-Seidelove* iterativne metode. Najprej sistem enačb $\mathbf{Ax} = \mathbf{b}$ zapišemo kot:

$$\sum_{j=0}^{n-1} A_{ij}x_j = b_i \quad i = 0, 1, \dots, n-1. \quad (6.22)$$

Predpostavimo, da smo v $k-1$ koraku iterativne metode in so znani približki $x_j^{(k-1)}$ ($j = 0, 1, \dots, n-1$). Iz zgornje vsote izpostavimo člen i :

$$A_{ii} x_i^{(k-1)} + \sum_{j=0, j \neq i}^{n-1} A_{ij} x_j^{(k-1)} = b_i \quad i = 0, 1, \dots, n-1. \quad (6.23)$$

Ker približki $x_j^{(k-1)}$ ne izpolnjujejo natančno linearnega problema, lahko iz zgornje enačbe določimo nov približek $x_i^{(k)}$:

$$x_i^{(k)} = \frac{1}{A_{ii}} \left(b_i - \sum_{j=0}^{i-1} A_{ij} x_j^{(k)} - \sum_{j=i+1}^{n-1} A_{ij} x_j^{(k-1)} \right) \quad (6.24)$$

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 željeno natančnost rešitve ϵ :

$$\left| x_i^{(k)} - x_i^{(k-1)} \right| < \epsilon \quad (6.25)$$

6.6.2 Zgled

```
A = np.array([[8, -1, 1],
              [-1, 6, -1],
              [0, -1, 6]], dtype=float)
b = np.array([-14, 36, 6], dtype=float)
```

Začetni približek:

```
x = np.zeros(len(A))
x
array([0., 0., 0.])
```

Pripravimo matriko **A** brez diagonalnih elementov (Zakaj? Poskusite odgovoriti spodaj, ko bomo izvedli iteracije.)

```
K = A.copy() #naredimo kopijo, da ne povozimo podatkov
np.fill_diagonal(K, np.zeros(3)) #spremenimo samo diagonalne elemente
K
array([[ 0., -1.,  1.],
       [-1.,  0., -1.],
       [ 0., -1.,  0.]])
```

Izvedemo iteracije:

$$x_i^{(k)} = \frac{1}{A_{ii}} \left(b_i - \sum_{j=0}^{i-1} A_{ij} x_j^{(k)} - \sum_{j=i+1}^{n-1} A_{ij} x_j^{(k-1)} \right) \quad (6.26)$$

(Ker bomo vrednosti takoj zapisali v **x**, ni treba razbiti vsote na dva dela)

```
for k in range(3):
    xk_1 = x.copy()
    print(5*'-' + f'iteracija {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}')
```

```

- - - - -iteracija 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
- - - - -iteracija 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
- - - - -iteracija 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

```
A@x
```

```
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).

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
$\dots$	$\dots$
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

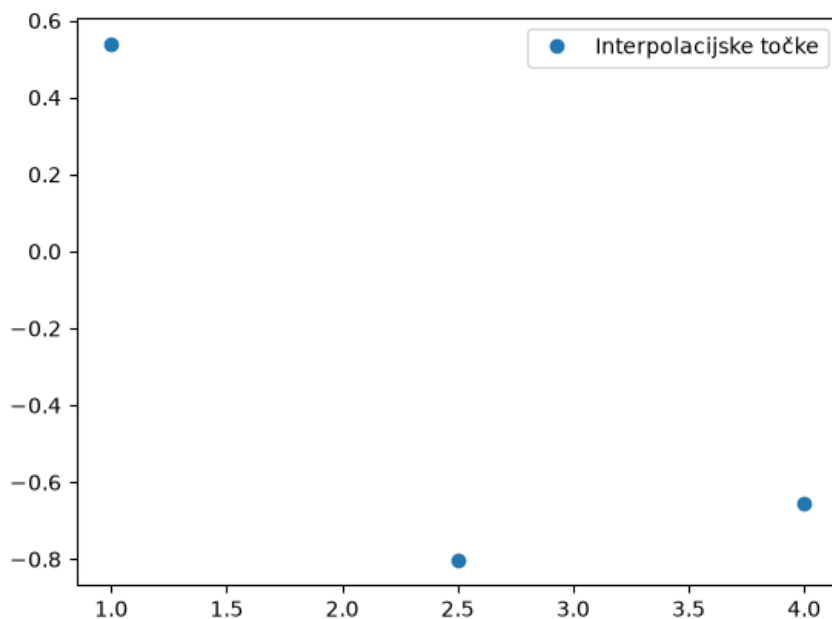
Pri interpolaciji izhajamo iz tabele vrednosti. Da bomo pozneje lahko enostavno prikazali napako, smo zgornjo tabelo generirali s pomočjo izraza $y = \cos(x)$!

Pripravimo numerični zgled; najprej uvozimo pakete:

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

Nato pripravimo tabelo ter prikaz:

```
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) + np.random.rand(len(x))*0.3
plt.plot(x, y, 'o', label='Interpolacijske točke');
plt.legend();
```



7.2 Interpolacija s polinomom

Interpolacija s polinomom se zdi najbolj primerna, saj je enostavna!

Polinom stopnje $n - 1$:

$$y = a_0 x^{n-1} + a_1 x^{n-2} + \cdots + a_{n-2} x + a_{n-1}. \quad (7.1)$$

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:

$$y_i = a_0 x_i^{n-1} + a_1 x_i^{n-2} + \cdots + a_{n-2} x_i + a_{n-1}. \quad (7.2)$$

Ker imamo podanih n parov, lahko določimo n neznanih konstant a_i , ki definirajo polinom stopnje $n - 1$. Sistem n linearnih enačb lahko zapišemo:

$$\begin{bmatrix} x_0^{n-1} & x_0^{n-2} & \cdots & x_0^0 \\ x_1^{n-1} & x_1^{n-2} & \cdots & x_1^0 \\ \vdots & \vdots & \ddots & \vdots \\ x_{n-1}^{n-1} & x_{n-1}^{n-2} & \cdots & x_{n-1}^0 \end{bmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} \quad (7.3)$$

Sistem linearnih enačb zapišemo v obliki:

$$\mathbf{M} \mathbf{a} = \mathbf{b}. \quad (7.4)$$

Definirajmo matriko koeficientov $\mathbf{M}$:

```
M = np.asarray([[_**p for p in reversed(range(len(x)))] for _ in x])
M
array([[ 1.  ,  1.  ,  1.  ],
       [ 6.25,  2.5 ,  1.  ],
       [16.  ,  4.  ,  1.  ]])
```

Izračunamo koeficiente $a_0, a_1, \dots$:

```
resitev = np.linalg.solve(M, y)
resitev
```

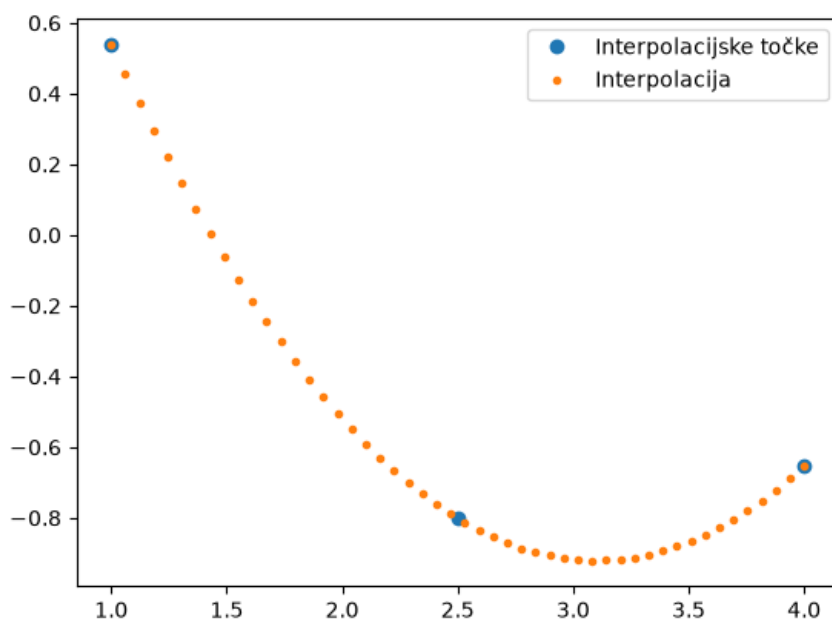
```
| array([ 0.33087687, -2.05236633,  2.26179176])
```

Pripravimo interpolacijski polinom kot Pythonovo funkcijo:

```
| 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:

```
| 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):

```
| np.linalg.cond(M)
```

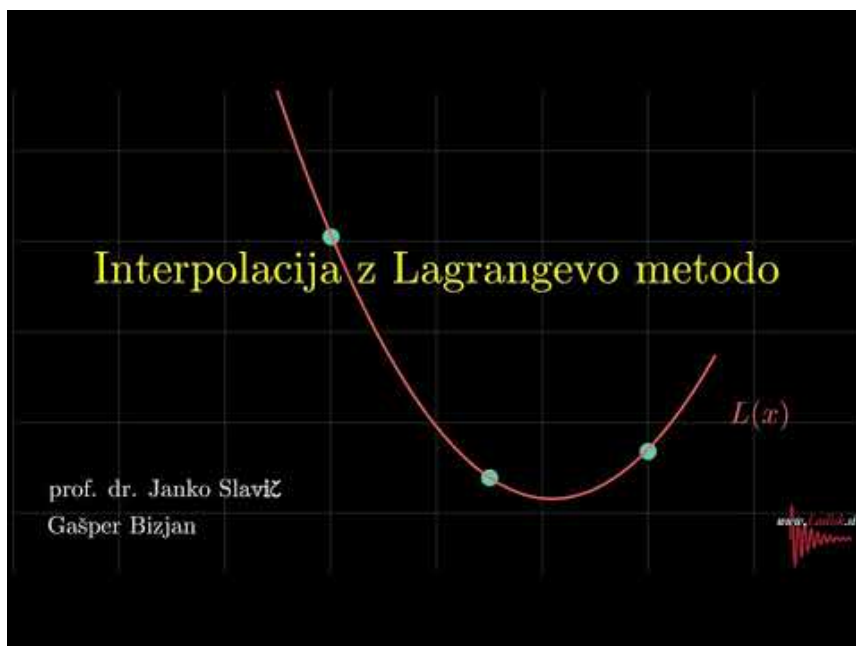
```
| np.float64(71.3022787031109)
```

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](#))) 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!

```
| from IPython.display import YouTubeVideo
| YouTubeVideo('c_TbU8XBHsw', width=800, height=300)
```



Lagrangev interpolacijski polinom stopnje $n - 1$ je definiran kot:

$$P_{n-1}(x) = \sum_{i=0}^{n-1} y_i l_i(x), \quad (7.5)$$

kjer je l_i Lagrangev polinom:

$$l_i(x) = \prod_{j=0, j \neq i}^{n-1} \frac{x - x_j}{x_i - x_j}. \quad (7.6)$$

Poglejmo si interpolacijo za zgoraj prikazane x in y podatke.

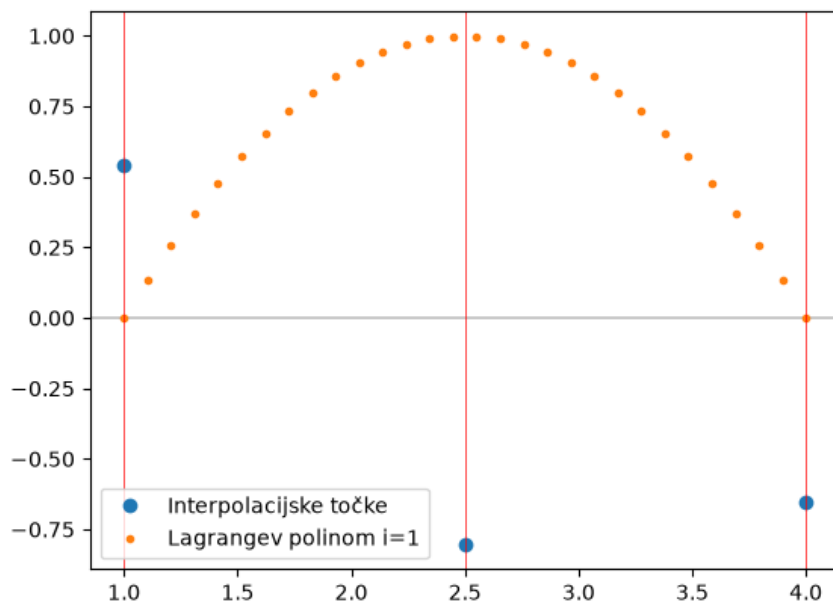
Definirajmo najprej Lagrangeve polinome $l_i(x) = \prod_{j=0, j \neq i}^{n-1} \frac{x - x_j}{x_i - x_j}$:

```
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

def slika(i=0):
    xint = np.linspace(np.min(x), np.max(x), 30)
    plt.plot(x, y, 'o', label='Interpolacijske točke')
    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()

slika(i=1)
```



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; implementirajmo torej Lagrangev interpolacijski polinom:

$$P_{n-1}(x) = \sum_{i=0}^{n-1} y_i l_i(x), \quad (7.7)$$

```
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:

```
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 polinom');
    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](#).

Uvoz funkcije `interact`

```
| from ipywidgets import interact
|
| interact(slika);
|
| interactive(children=(IntSlider(value=0, description='i', max=1), Output()),
| _dom_classes=('widget -interact',)...
```

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:

$$e = f(x) - P_{n-1}(x) = \frac{f^{(n)}(\xi)}{n!} (x - x_0)(x - x_1) \cdots (x - x_{n-1}), \quad (7.8)$$

kjer je $f^{(n)}$ odvod funkcije, $n - 1$ stopnja interpolacijskega polinoma in ξ vrednost na interpoliranem intervalu $[x_0, x_{n-1}]$.

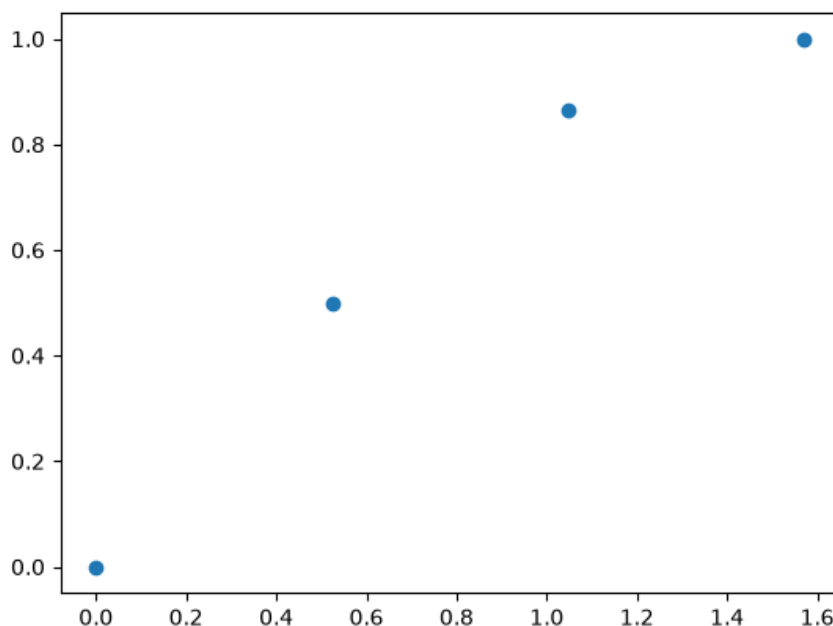
7.3.2 Zgled

Tukaj si bomo ogledali interpolacijo točk:

```
| x = np.array([0.          ,  0.52359878,  1.04719755,  1.57079633])
| y = np.array([ 0.          ,  0.5          ,  0.8660254 ,  1.          ])
```

Točke prikažimo:

```
| plt.plot(x, y, 'o');
```



Linearna interpolacija za vrednost pri $x=1.57079633/2$:

```

| y_linearna = lagrange_interpolacija(x=x[ -1]/2, x_int=x[1:3], y_int=y[1:3])
| y_linearna
|
| np.float64(0.6830127)
|
| Kvadratna:
|
| y_kvadratna = lagrange_interpolacija(x=x[ -1]/2, x_int=x[0:3], y_int=y[0:3])
| y_kvadratna
|
| np.float64(0.6997595236464176)
|
| Kubična
|
| y_kubična = lagrange_interpolacija(x=x[ -1]/2, x_int=x, y_int=y)
| y_kubična
|
| np.float64(0.705889286844634)

```

7.3.3 Zgled ocene napake

Pri interpolaciji ponavadi funkcije $f(x)$ ne poznamo in napako ocenimo s pomočjo formule:

$$e = \frac{f^{(n)}(\xi)}{n!} (x - x_0) (x - x_1) \cdots (x - x_{n-1}). \quad (7.9)$$

Pri tem vrednost ξ ni znana; ker je v primeru linearne interpolacije ($n = 2$) drugi odvod sinusne funkcije ($f^{(n)}$) med -1 in $+1$, velja:

$$|e| \leq \left| \frac{-1}{2!} (\pi/4 - \pi/6) (\pi/4 - \pi/3) \right| = \frac{1}{2} \frac{\pi}{12} \frac{\pi}{12} = \frac{\pi^2}{288} = 0,034 \quad (7.10)$$

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](#)).

Uporabili bomo funkcijo za interpoliranje tabele z zlepk, `scipy.interpolate.interp1d` ([dokumentacija](#)):

```

| interp1d(x, y, kind='linear', axis= -1, copy=True, bounds_error=None, fill_value=nan,
| assume_sorted=False)

```

Podati moramo vsaj dva parametra: seznama interpolacijskih točk x in y . Privzeti parameter `kind='linear'` pomeni, da interpoliramo 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.

Opomba: `interp1d` je v SciPy označena kot *legacy* (starejši vmesnik, ki se ne razvija več, a še deluje). Za nove programe SciPy priporoča `numpy.interp` (odsekoma linearna interpolacija), `scipy.interpolate.CubicSpline` ([dokumentacija](#)) in `scipy.interpolate.make_interp_spline` ([dokumentacija](#)); vse vrnejo objekt, ki ga kličemo enako kot f spodaj.

```

| from scipy.interpolate import interp1d

```

Definirajmo tabelo podatkov:

```

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])

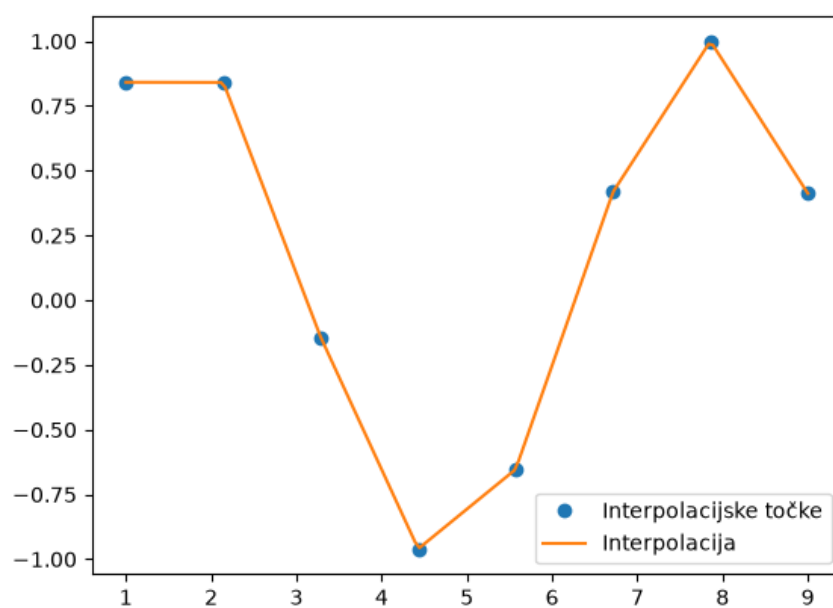
len(x)

8

from scipy.interpolate import interp1d

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();

```



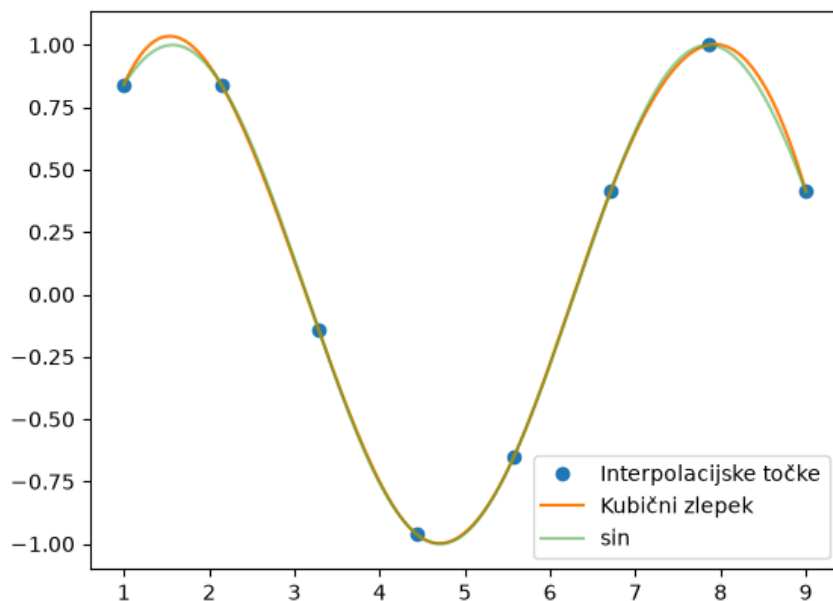
7.4 Kubični zleпки

Preden gremo v teorijo zlepkov, si pogledjmo rezultat, ki ga dobimo s klicanjem funkcije `interp1d` s parametrom `kind='cubic'` (rezultat je kubični zlepek).

```

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 zleпки so pogost način interpolacije.

Zahtevamo, da je: $x_0 < x_1 < \dots < x_n$.

Od točke x_i do x_{i+1} naj bo zlepek polinom:

$$f_{i,i+1}(x) = a_{i,3}x^3 + a_{i,2}x^2 + a_{i,1}x + a_{i,0}, \quad (7.11)$$

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:

$$f(x) = \begin{cases} f_{0,1}(x); & x \in [x_0, x_1) \\ f_{1,2}(x); & x \in [x_1, x_2) \\ \vdots & \\ f_{n-1,n}(x); & x \in [x_{n-1}, x_n] \end{cases} \quad (7.12)$$

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 $4n$ neznank potrebujemo $4n$ enačb. Poglejmo si, kako jih dobimo:

- n enačb dobimo iz interpolacijskega pogoja: $y_i = f_{i,i+1}(x_i)$, $i = 0, 1, 2, \dots, n-1$
- 1 enačbo iz zadnje točke: $y_n = f_{n-1,n}(x_n)$
- $3(n-1)$ enačb dobimo iz pogoja C^2 zveznosti:

$$\lim_{x \rightarrow x_i^-} f(x) = \lim_{x \rightarrow x_i^+} f(x), \quad (7.13)$$

$$\lim_{x \rightarrow x_i^-} f'(x) = \lim_{x \rightarrow x_i^+} f'(x) \quad (7.14)$$

in

$$\lim_{x \rightarrow x_i^-} f''(x) = \lim_{x \rightarrow x_i^+} f''(x). \quad (7.15)$$

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 zleпки

Naravni kubični zleпки temeljijo na ideji Eulerjevega nosilca:

$$EI \frac{d^4 y}{dx^4} = q(x), \quad (7.16)$$

kjer je E modul elastičnosti, I drugi moment preseka in $q(x)$ zunanja porazdeljena sila. Ker zunanje porazdeljene sile ni ($q(x) = 0$), velja:

$$EI \frac{d^4 y}{dx^4} = 0. \quad (7.17)$$

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):

$$f''(x_0) = 0 \quad \text{in} \quad f''(x_n) = 0 \quad (7.18)$$

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 namen 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 ($i = 1, \dots, n-1$):

$$k_{i-1} + 4k_i + k_{i+1} = \frac{6}{h^2} (y_{i-1} - 2y_i + y_{i+1}). \quad (7.19)$$

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 zleپke:

$$k_0 = k_n = 0. \quad (7.20)$$

Ko določimo neznake k_i , jih uporabimo v odsekoma definirani funkciji:

$$f_{i,i+1}(x) = \frac{k_{i+1}}{6} \left(\frac{(x-x_i)^3}{h} - (x-x_i)h \right) - \frac{k_i}{6} \left(\frac{(x-x_{i+1})^3}{h} - (x-x_{i+1})h \right) + \frac{y_{i+1}(x-x_i) - y_i(x-x_{i+1})}{h}. \quad (7.21)$$

7.4.2 Numerična implementacija

Najprej pripravimo funkcijo, katera za podane interpolacijske točke reši sistem linearnih enačb in vrne koeficiente k_i :

```
def kubichni_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 linearnem problemu, lahko izračun zelo pohitrimo, če upoštevamo tridiagonalnost matrike koeficientov! (Glejte [odgovor na stackoverflow.com](#)).

Poglejmo si primer izračuna koeficientov:

```

x = np.asarray([1, 2, 3, 4, 5])
y = np.asarray([0, 1, 0, 1, 0])

k, A, b = kubicki_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:

$$f_{i,i+1}(x) = \frac{k_{i+1}}{6} \left(\frac{(x-x_i)^3}{h} - (x-x_i)h \right) - \frac{k_i}{6} \left(\frac{(x-x_{i+1})^3}{h} - (x-x_{i+1})h \right) + \frac{y_{i+1}(x-x_i) - y_i(x-x_{i+1})}{h} \quad (7.22)$$

```

def kubicki_zlepki(k, x, y, x_najdi):
    """ Vrne kubični 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[1] - x[0]
    i = int((x_najdi - x[0])/h)
    if i >= len(k) - 1:
        i = len(k) - 2
    out = ((x_najdi - x[i])**3/h - (x_najdi - x[i])*h)*k[i+1]/6.0 \
          - ((x_najdi - x[i+1])**3/h - (x_najdi - x[i+1])*h)*k[i]/6.0 \
          + (y[i+1]*(x_najdi - x[i]) \
            - y[i]*(x_najdi - x[i+1]))/h
    return out

```

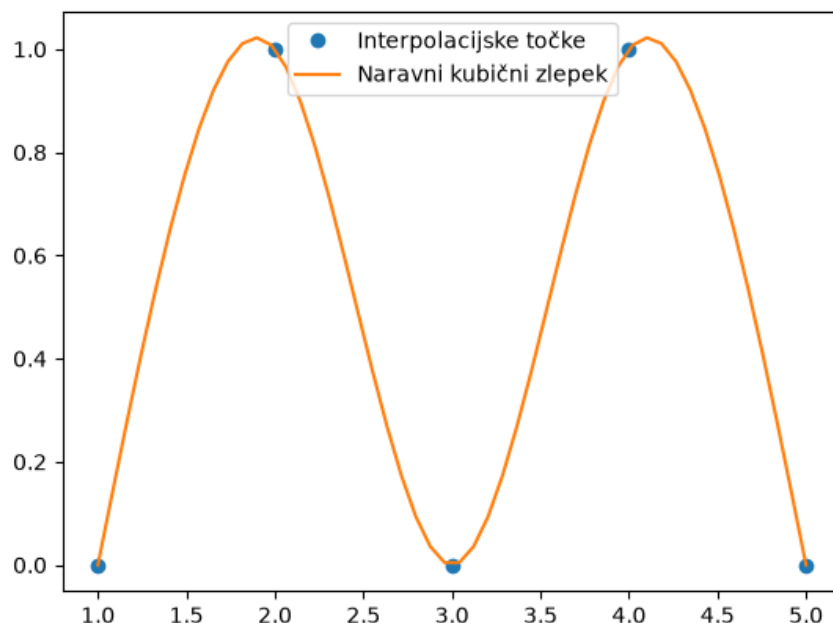
Izračunamo interpolirane vrednosti:

```

xint = np.linspace(np.min(x), np.max(x), 50)
yint = np.asarray([kubicki_zlepki(k, x, y, _) for _ in xint])

plt.plot(x, y, 'o', label='Interpolacijske točke')
plt.plot(xint, yint, label='Naravni kubični zlepek')
plt.legend();

```



7.5 Dodatno

- 2D interpolacija: https://www.youtube.com/watch?v=_cJLVhdj0j4
- Strojno prevajanje: <https://pypi.org/project/goslate>

7.6 Nekaj komentarjev modula `scipy.interpolate`

SciPy ima implementiranih večje število različnih interpolacij (glejte [dokumentacijo](#)). S stališča uporabe se bomo tukaj dotaknili objektne implementacije `scipy.interpolate.InterpolatedUnivariateSpline` ([dokumentacija](#)) (starejši pristop temelji na proceduralnem programiranju, glejte [dokumentacijo](#) `scipy.interpolate.splrep`):

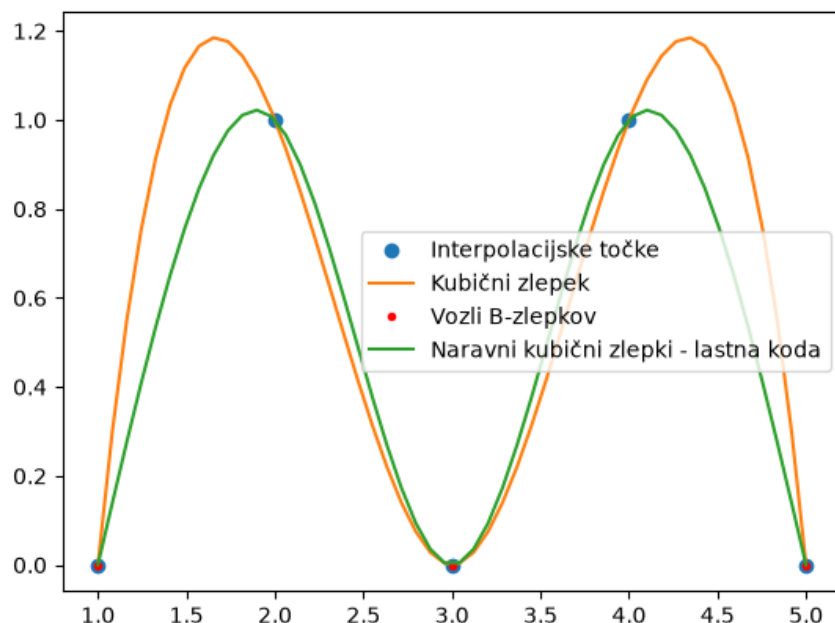
```
InterpolatedUnivariateSpline(x, y, w=None, bbox=[None, None], k=3, ext=0,
check_finite=False)
```

Pri inicializaciji objekta `InterpolatedUnivariateSpline` moramo posredovati interpolacijske točke `x` in `y`. Argument `k` s privzeto vrednostjo `k=3` definira red interpolacijskega zleпка ($1 \leq k \leq 5$). Pomemben opcijski parameter je tudi `w`, ki definira uteži posameznim interpolacijskim točkam (uporabimo ga, če želimo določenim področjem dati večji poudarek).

Opomba: `InterpolatedUnivariateSpline` (in sorodni `UnivariateSpline`) sta od SciPy 1.15 označena kot *legacy*; sodobnejša ekvivalenta sta `make_interp_spline` (interpolacija) in `make_smoothing_spline` oz. `make_splrep` (aproksimacija, glejte naslednje predavanke). Objektni vmesnik spodaj še vedno deluje in je zaradi preglednosti primeren za učenje.

```
from scipy.interpolate import InterpolatedUnivariateSpline

spl = InterpolatedUnivariateSpline(x, y, k=3) # pogledajte 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 zleпки - lastna koda');
plt.legend();
```



Ker gre za B-zlepke, je rezultat drugačen kot tisti, ki smo ga izpeljali z naravnimi kubičnimi zlepk. V nasprotju z naravnimi kubičnimi zlepk, ki imajo vozle (angl. *knots*) v interpolacijskih točkah, se vozli B-zlepkov prilagodijo podatkom. V konkretnem primeru so vozli v točkah:

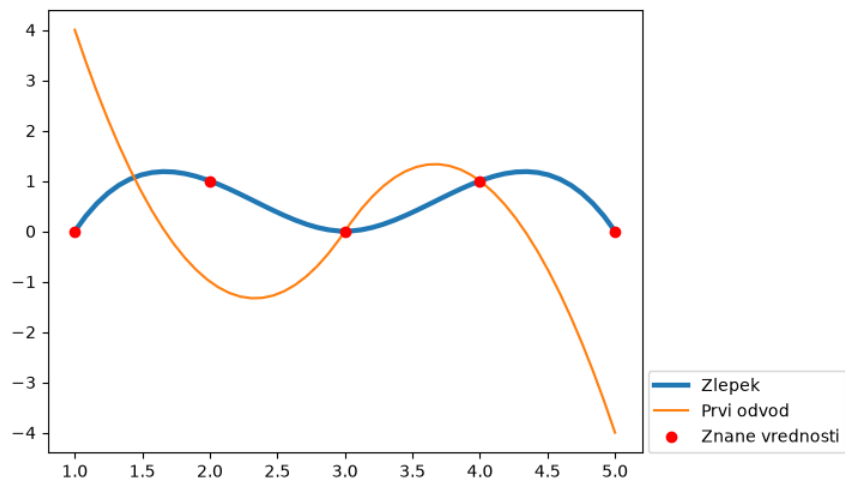
```
| spl.get_knots()
|
| array([1., 3., 5.])
```

7.6.1 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](#)).

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):

```
| 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.

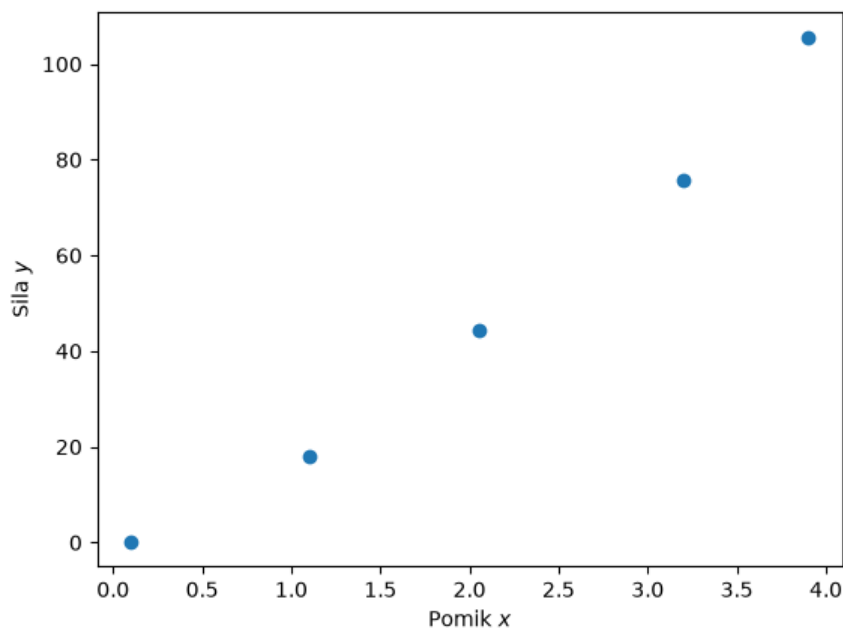
Oglejmo si primer meritve (linearne) vzmeti (x je raztezek, y je pomejena sila):

```
import numpy as np

x = np.array([0.1, 1.1, 2.05, 3.2, 3.9 ])
y = np.array([0.22, 18.15, 44.33, 75.59, 105.63])
```

Poglejmo si podatke na sliki, najprej uvozimo potrebne pakete:

```
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:

$$f(x) = a_0 x + a_1 \tag{8.1}$$

Poznamo tabelo n podatkov x_i, y_i za $i = 0, 1, \dots, n-1$; teh je več, kot jih potrebujemo za določitev dveh konstant a_0 in a_1 , zato imamo torej predoločen sistem linearnih enačb:

$$y_i = a_0 x_i + a_1 \quad \text{za} \quad i = 0, 1, \dots, n-1. \quad (8.2)$$

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 P najmanjša:

$$P(a_0, a_1) = \sum_{i=0}^{n-1} |y_i - (a_0 x_i + a_1)|. \quad (8.3)$$

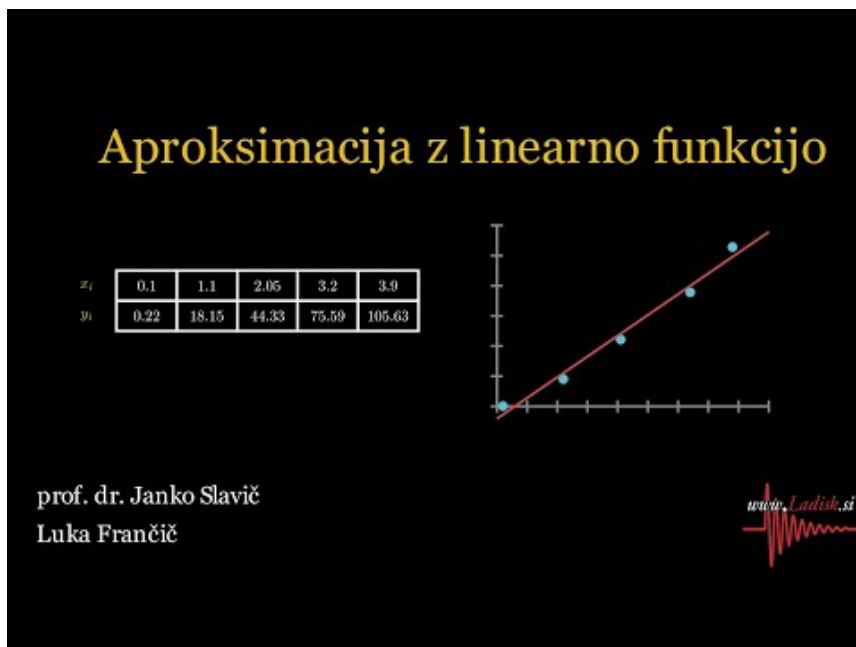
Ker pa taka funkcija $P(a_0, a_1)$ ni zvezno odvedljiva, raje uporabimo **metodo najmanjših kvadratov**:

$$S(a_0, a_1) = \sum_{i=0}^{n-1} (y_i - (a_0 x_i + a_1))^2. \quad (8.4)$$

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

```
from IPython.display import YouTubeVideo
YouTubeVideo('8 -oixY6GG9E', width=800, height=300)
```



Poiskati moramo konstanti a_0, a_1 , da bo vsota kvadratov razlik med funkcijo in tabelirano vrednostjo (x_i, y_i) , kjer $i = 0, 1, \dots, n-1$ in je n število tabeliranih podatkov):

$$S(a_0, a_1) = \sum_{i=0}^{n-1} (y_i - (a_0 x_i + a_1))^2 \quad (8.5)$$

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 :

$$\frac{\partial S(a_0, a_1)}{\partial a_0} = 2 \sum_{i=0}^{n-1} (y_i - a_0 x_i - a_1) (-x_i) \quad (8.6)$$

Izraz uredimo:

$$\frac{\partial S(a_0, a_1)}{\partial a_0} = -2 \left(\sum_{i=0}^{n-1} y_i x_i - a_0 \sum_{i=0}^{n-1} x_i^2 - a_1 \sum_{i=0}^{n-1} x_i \right) \quad (8.7)$$

Podobno postopamo še za a_1 :

$$\frac{\partial S(a_0, a_1)}{\partial a_1} = 2 \sum_{i=0}^{n-1} (y_i - a_0 x_i - a_1) (-1) \quad (8.8)$$

$$\frac{\partial S(a_0, a_1)}{\partial a_1} = -2 \left(\sum_{i=0}^{n-1} y_i - a_0 \sum_{i=0}^{n-1} x_i - a_1 \sum_{i=0}^{n-1} 1 \right) \quad (8.9)$$

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:

$$a_0 \sum_{i=0}^{n-1} x_i^2 + a_1 \sum_{i=0}^{n-1} x_i = \sum_{i=0}^{n-1} y_i x_i \quad (8.10)$$

in

$$a_0 \sum_{i=0}^{n-1} x_i + a_1 n = \sum_{i=0}^{n-1} y_i. \quad (8.11)$$

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:

```
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: [764.7085 243.92 ]
```

Sedaj moramo rešiti linearni sistem:

$$\mathbf{A} \mathbf{a} = \mathbf{b}, \quad (8.12)$$

Opomba, tukaj smo vektor neznank zapisali kot $\mathbf{a} = (a_0, a_1)$.

Sistem rešimo:

```
a0, a1 = np.linalg.solve(A, b)
a0, a1
(np.float64(27.497258679085512), np.float64(-8.135325465707014))
```

Preverimo še število pogojenosti:

```
| np.linalg.cond(A)
| np.float64(25.20071350870242)
```

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 .

```
| 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()
|
| from ipywidgets import interact
| interact(slika, naklon=(0, 50, 2), premik=( -10, 10, 1));
|
| interactive(children=(IntSlider(value=27, description='naklon', max=50, step=2),
| IntSlider(value= -8, descripti...
```

8.2.1 Uporaba psevdo inverzne matrike

Do enakega 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 ($i = 0, 1, \dots, n-1$):

```
| A=np.array([x, np.ones_like(x)]).T
| A
|
| 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 ($i = 0, 1, 2, \dots, n-1$):

```
| y
|
| array([ 0.22, 18.15, 44.33, 75.59, 105.63])
```

Preverimo rang matrike koeficientov:

```

| np.linalg.matrix_rank(A)
|
| np.int64(2)
|
| In še rang razširjene matrike:
|
| Ab = np.hstack((A,np.array([y]).T))
| Ab
|
| array([[1.0000e -01, 1.0000e+00, 2.2000e -01],
|        [1.1000e+00, 1.0000e+00, 1.8150e+01],
|        [2.0500e+00, 1.0000e+00, 4.4330e+01],
|        [3.2000e+00, 1.0000e+00, 7.5590e+01],
|        [3.9000e+00, 1.0000e+00, 1.0563e+02]])
|
| np.linalg.matrix_rank(Ab)
|
| np.int64(3)

```

Ker rešujemo sistem m linearnih enačb z n neznankami ter velja $m > n$ in je rang razširjene matrike $n + 1$, imamo predoločeni sistem.

Vektor neznanih parametrov $\mathbf{a}$ določimo z uporabo psevdo inverzne matrike:

$$\mathbf{a} = \mathbf{A}^+ \mathbf{y} \quad (8.13)$$

```

| np.linalg.pinv(A).dot(y)
|
| array([27.49725868, -8.13532547])

```

8.3 Metoda najmanjših kvadratov za poljubni polinom

Linearno aproksimacijo, predstavljeno zgoraj, bomo posplošili za poljubni polinom stopnje m :

$$f(a_0, a_1, \dots, a_m, x) = \sum_{s=0}^m a_s \underbrace{x^{m-s}}_{f_s(x)}, \quad (8.14)$$

kjer $f_s(x) = x^{m-s}$ imenujemo bazna funkcija ($s = 0, 1, 2, \dots, m$).

Tabela podatkov naj bo definirana z x_i, y_i , kjer je $i = 0, 1, 2, \dots, n-1$.

Opomba: zaradi kompaktnosti zapisa bomo konstante a zapisali v vektorski obliki $\mathbf{a} = [a_0, a_1, \dots, a_m]$.

Uporabimo metodo najmanjših kvadratov:

$$S(\mathbf{a}) = \sum_{i=0}^{n-1} (y_i - f(\mathbf{a}, x_i))^2 = \sum_{i=0}^{n-1} \left(y_i - \sum_{s=0}^m a_s x_i^{m-s} \right)^2. \quad (8.15)$$

Potreben pogoj za nastop ekstreme 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 :

$$\frac{\partial S(\mathbf{a})}{\partial a_v} = \sum_{i=0}^{n-1} -2 \left(y_i - \sum_{s=0}^m a_s x_i^{m-s} \right) x_i^{m-v} \quad (8.16)$$

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:

$$\sum_{i=0}^{n-1} \left(\sum_{s=0}^m a_s x_i^{m-s} \right) x_i^{m-v} = \sum_{i=0}^{n-1} y_i x_i^{m-v} \quad (8.17)$$

Izraz uredimo:

$$\sum_{i=0}^{n-1} \sum_{s=0}^m a_s x_i^{2m-s-v} = \sum_{i=0}^{n-1} y_i x_i^{m-v} \quad (8.18)$$

Zamenjamo vrstni red seštevanja ter izpeljemo:

$$\sum_{s=0}^m \left(a_s \sum_{i=0}^{n-1} x_i^{2m-s-v} \right) = \sum_{i=0}^{n-1} y_i x_i^{m-v} \quad \text{za: } v = 0, 1, \dots, m \quad (8.19)$$

Izpeljali smo enačbo v sistema $m + 1$ linearnih enačb:

$$\mathbf{A} \mathbf{a} = \mathbf{b} \quad (8.20)$$

Element $A_{v,s}$ matrike koeficientov je:

$$A_{v,s} = \sum_{i=0}^{n-1} x_i^{2m-v-s}, \quad (8.21)$$

Element vektorja konstant je:

$$b_v = \sum_{i=0}^{n-1} y_i x_i^{m-v} \quad (8.22)$$

8.3.1 Numerični zgled

Uporabimo podatke iz prve naloge in poskusimo aproksimirati s polinomom 2. stopnje ($m = 2$).

Tabela podatkov je:

```
| x
| array([0.1 , 1.1 , 2.05, 3.2 , 3.9 ])
| y
| array([ 0.22, 18.15, 44.33, 75.59, 105.63])
```

Izračunajmo matriko koeficientov:

$$A_{v,s} = \sum_{i=0}^{n-1} x_i^{2m-v-s} \quad (8.23)$$

```
| 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
| array([[355.32690625, 102.034125 , 30.8725   ],
|        [102.034125 , 30.8725   , 10.35      ],
|        [ 30.8725   , 10.35      , 5.         ]])
```

Izračunajmo še vektor konstant:

$$b_v = \sum_{i=0}^{n-1} y_i x_i^{m-v} \quad (8.24)$$

```

b = np.zeros(m+1)
for v in range(m+1):
    b[v] = np.dot(y, x**(m - v))
b

array([2588.934425,  764.7085  , 243.92   ])

```

Preverimo število pogojenosti:

```

np.linalg.cond(A)

np.float64(963.2125856290679)

```

Rešimo sistem:

```

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

array([ 3.17760897, 14.67380042, -1.21091344])

```

Glede na definicijo aproksimacijskega polinoma:

$$f(a_0, a_1, \dots, a_m, x) = \sum_{v=0}^m a_v x^{m-v} \quad (8.25)$$

Kar v konkretnem primeru je aproksimacijski polinom:

$$f(x) = 3.18393375 x^2 + 14.64106847 x - 1.22621065 \quad (8.26)$$

Definirajmo numerično implementacijo:

```

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)

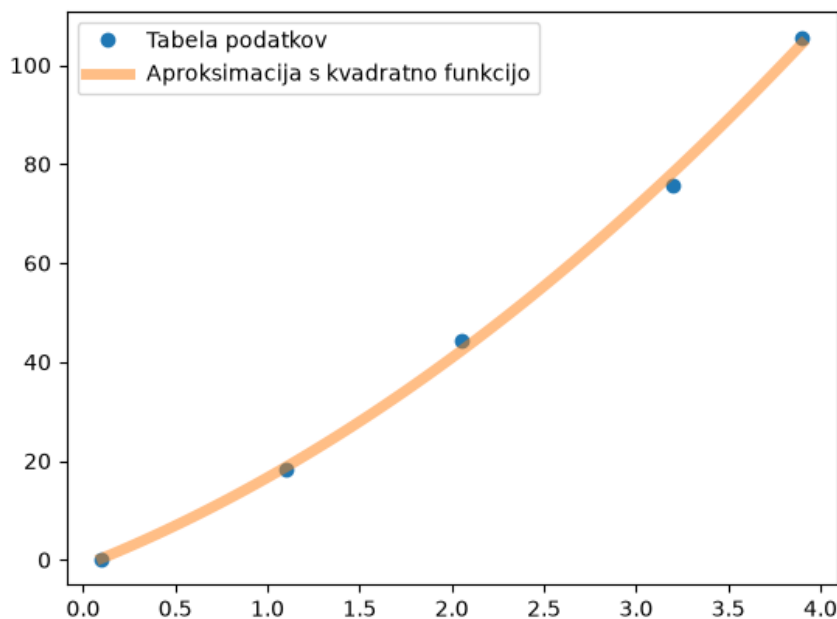
```

Prikažemo:

```

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='Aproximacija s kvadratno
funkcijo')
plt.legend();

```



Poglejmo še napako aproksimacije:

$$e_i = y_i - f(x_i) \quad (8.27)$$

za $i = 0, 1, 2, \dots, n - 1$.

Pri pravilno izvedeni aproksimaciji je nekaj e_i pozitivnih in nekaj negativnih. Poglejmo, če je to res v našem primeru:

```
| e = y - apr_polinom(x, a)
| e
|
| array([-0.06824269, -0.62517387,  2.1057209 , -2.69396374,  1.28165939])
```

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](#)):

```
| 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).

Opomba: `numpy.polyfit` in `numpy.polyld` spadata v starejši vmesnik za polinome; `numpy` za nove programe priporoča razred `numpy.polynomial.Polynomial` ([dokumentacija](#)): `p = Polynomial.fit(x, y, deg=2)` vrne objekt, ki ga kličemo kot funkcijo (`p(x)`), z `p.convert().coef` pa dobimo koeficiente v **naraščajočem** vrstnem redu potenc (ravno obratno kot `polyfit`).

Poglejmo si uporabo za predhodno obravnavani primer:

```
| koef = np.polyfit(x, y, deg=2)
| koef
```

```
| array([ 3.17760897, 14.67380042, -1.21091344])
| a # rezultat lastne implementacije
| array([ 3.17760897, 14.67380042, -1.21091344])
```

Ko imamo koeficiente, lahko ustvarimo objekt polinoma s klicem `numpy.poly1d` ([dokumenta-cija](#)):

```
| poly1d(c_or_r, r=False, variable=None)
```

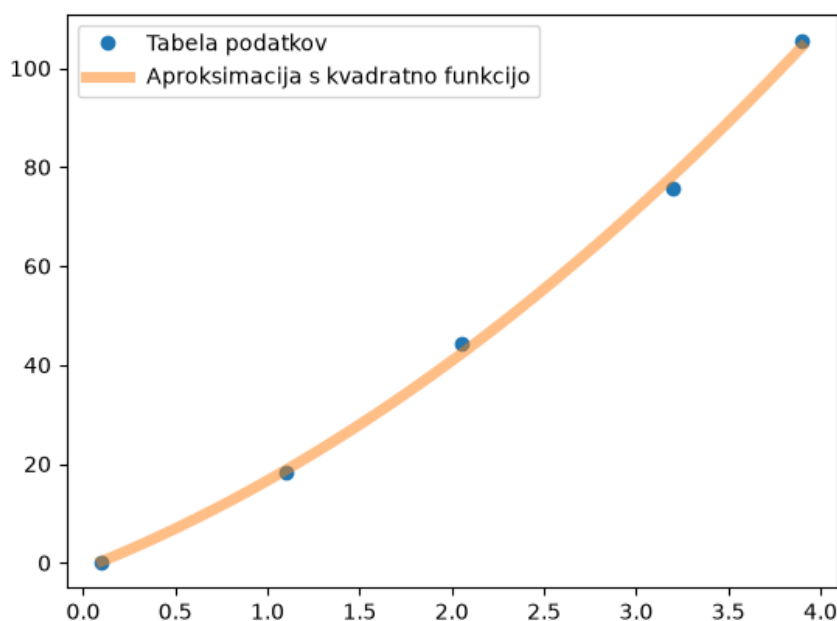
kjer `c_or_r` predstavlja seznam koeficientov polinoma oz. ničle polinoma v primeru, da je `r=True`. Funkcija vrne instanco objekta, s klicem katere lahko izračunamo vrednosti aproksimacijskega polinoma pri `x`, lahko pa izračunamo tudi druge stvari, kot na primer ničle polinoma.

Poglejmo si primer:

```
| p = np.poly1d(koef) # p je instanca objekta poly1d. Aproksimacijo sedaj dobimo z y =
| p(x).
```

Izrišimo vrednosti:

```
| 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:

```
| p.roots
| array([-4.69897273,  0.08109792])
```

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 funkcijo, ki temelji na metodi najmanjših kvadratov.

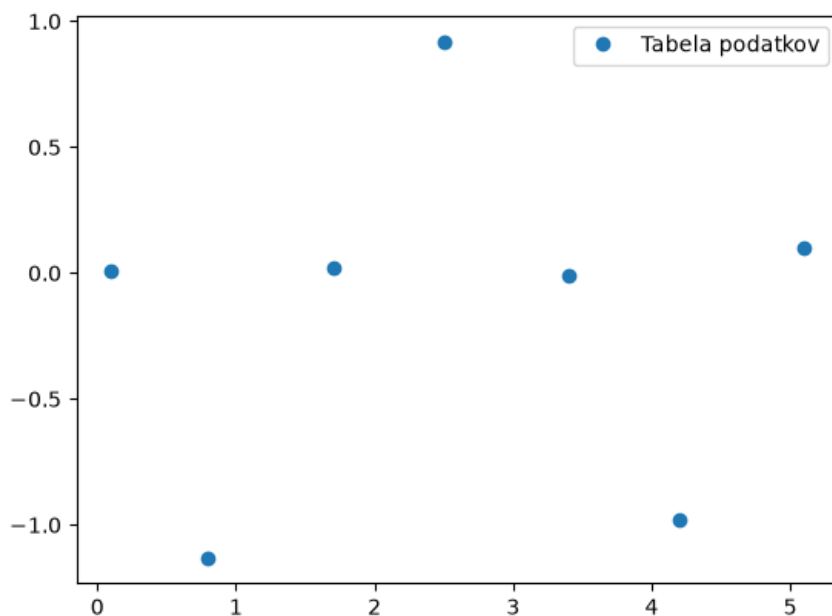
8.4.1 Aproksimacija s harmonsko funkcijo

Tabela podatkov je definirana kot:

```
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:

```
plt.plot(x, y, 'o', label='Tabela podatkov')
plt.legend();
```



Aproksimacijo z nelinearno funkcijo bomo izvedli s pomočjo `scipy.optimize.curve_fit` ([dokumentacija](#)):

```
curve_fit(f, xdata, ydata, p0=None, sigma=None, absolute_sigma=False,
check_finite=True, bounds=( -inf, inf), method=None, jac=None, **kwargs)
```

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 predstavlja ocenjeno kovarianco `popt`.

Definirajmo najprej Python funkcijo, katere prvi parameter je neodvisna spremenljivka `x`, nato pa sledijo parametri, ki jih želimo določiti:

```
def func(x, A, \omega, \phi):
    return A*np.sin( \omega*x+ \phi)
```

kjer je A amplituda, ω krožna frekvenca in ϕ faza harmonske funkcije. S pomočjo slike lahko ugibamo prve približke: $A=1$, $\omega=1$, $\phi=0$

Sedaj uvozimo `curve_fit` in izvedemo optimizacijski postopek:

```
from scipy.optimize import curve_fit

popt, pcov = curve_fit(func, x, y, p0=[1, 1, 0])
popt
```

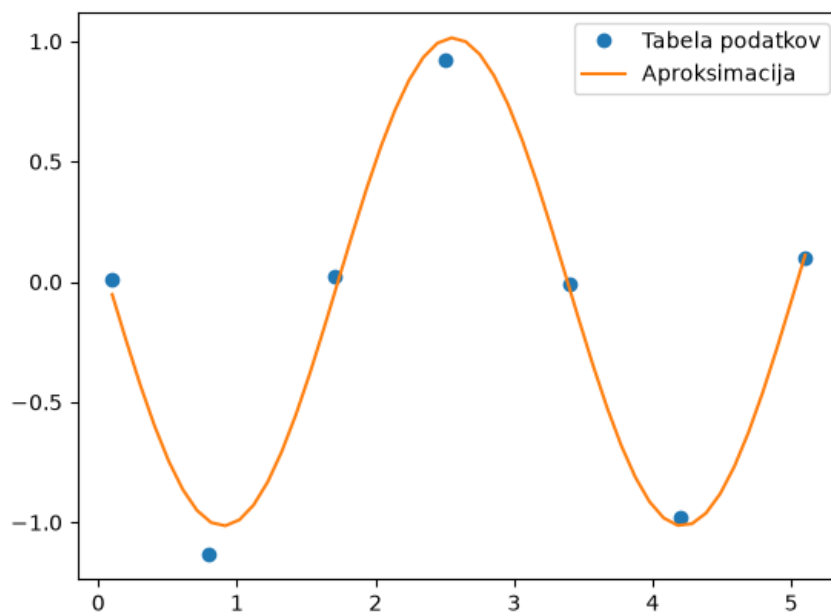
```
| array([ 1.01461945,  1.89652046, -3.27943097])
```

Izračunali smo pričakovane vrednosti (glejte zgoraj).

```
| func(x, *popt)
```

```
| array([ -0.0525477 , -0.99608783, -0.05612665,  1.00860622, -0.02753942,  
|        -1.01426499,  0.1110182  ])
```

```
| 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 Dodatno

Naredite `.exe` svojega programa:

- <https://pypi.org/project/py2exe/>
- <http://www.pyinstaller.org/>



Poglejte [pandas paket](#).

8.5.1 Aproksimacija z zlepci in uporabo SciPy

Tabela podatkov naj bo:

```
| x = np.linspace( -3, 3, 20)  
| x
```

```
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.          ])

np.random.seed(0) # seme generatorja naključnih števil
y = np.exp( -x**2) + 0.1 * np.random.normal(scale=.5, size=len(x))
y

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](#)), ki omogoča tako interpolacijo kot aproksimacijo z zleпки:

```
UnivariateSpline(x, y, w=None, bbox=[None, None], k=3, s=None, ext=0,
check_finite=False)
```

Parametra `x` in `y` predstavljata tabelo podatkov.

Opomba: `UnivariateSpline` je od SciPy 1.15 označen kot *legacy*; sodobnejši ekvivalent za aproksimacijo (glajenje) z zleпки je `scipy.interpolate.make_smoothing_spline` ([dokumentacija](#)) oz. `make_splrep`. Objektni vmesnik spodaj še deluje in je preglednejši za učenje.

Opcijski parameter `s` določa vrednost, katere vsota kvadratov razlik aproksimacijskega zleпка in aproksimacijskih točk ne sme preseči:

```
sum((w[i] * (y[i] -spl(x[i])))**2, axis=0) <= s
```

`w` so uteži posameznih točk.

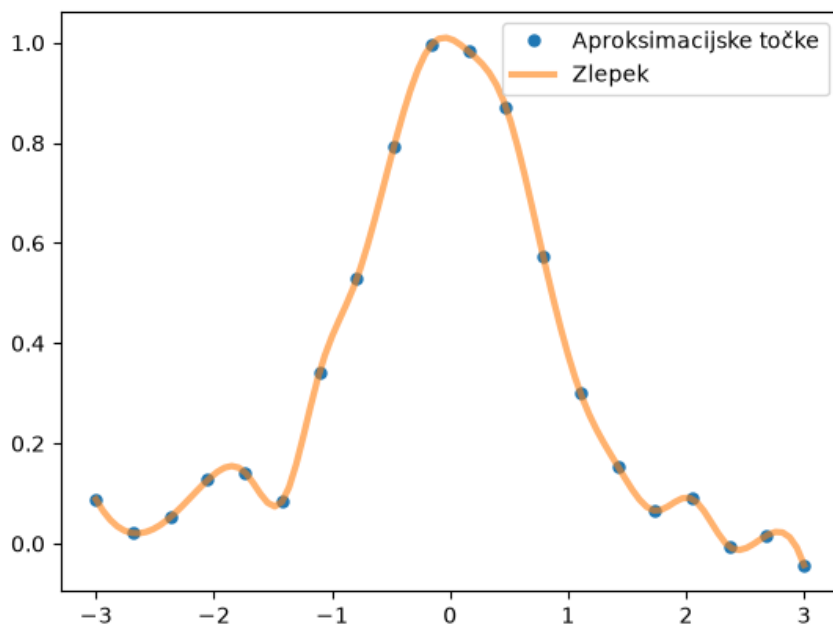
Če definiramo `s=0`, zahtevamo interpolacijo.

Parameter `k` definira stopnjo polinomskega zleпка (privzeto je `k=3`).

Aproksimacijo z zleпки izvedemo tako, da ob tabeli podatkov `x` in `y` definiramo še parameter `s`. Izvedimo interpolacijo:

```
from scipy.interpolate import UnivariateSpline
spl = UnivariateSpline(x, y, s=0.)

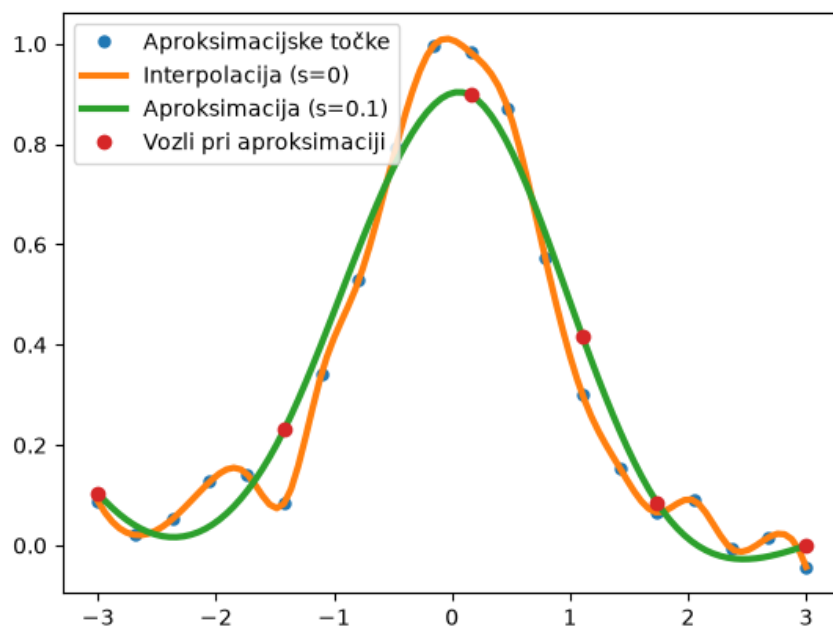
x_g = np.linspace( -3, 3, 100)
plt.plot(x, y, 'o', ms=5, label='Aproksimacijske točke')
plt.plot(x_g, spl(x_g), lw=3, label='Zlepek', alpha=0.6)
plt.legend();
```



Izvedimo še aproksimacijo:

```
| spl_a = UnivariateSpline(x, y, s=.1)

| 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=0.1)');
| plt.plot(spl_a.get_knots(), spl_a(spl_a.get_knots()), 'o', label='Vozli pri aproksimaciji');
| plt.legend();
```



Dejanski preostanek:

```
| spl_a.get_residual()

| 0.10002662427563205
```

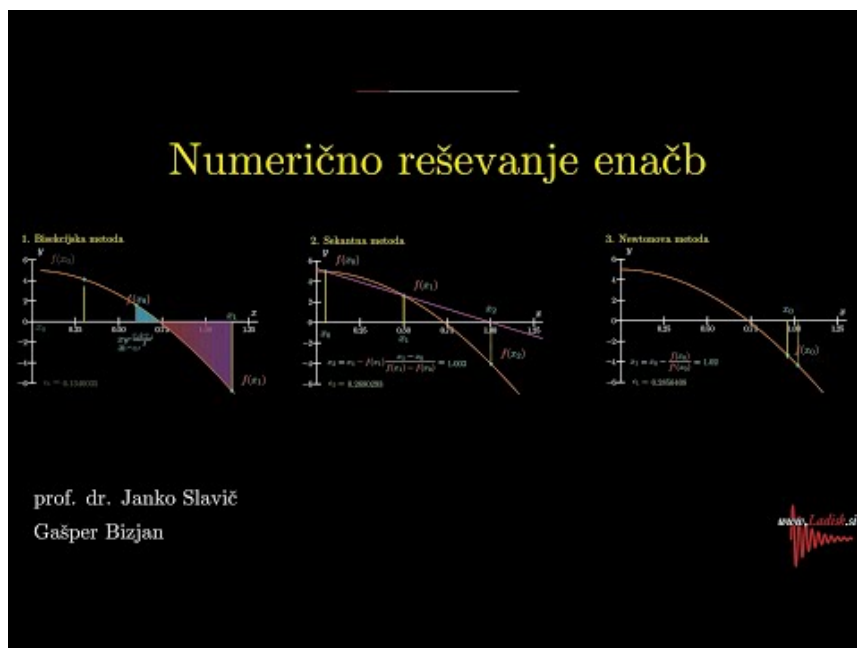
Poglavje 9

Reševanje enačb

9.1 Uvod

Za uvod si oglejmo spodnji video:

```
from IPython.display import YouTubeVideo
YouTubeVideo('17d55KE8SIU', width=800, height=300)
```



V okviru reševanja enačb obravnavamo poljubno enačbo, ki je odvisna od spremenljivke x in iščemo rešitev:

$$f(x) = 0. \quad (9.1)$$

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:

```
def f(x):
    return 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`:

```
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()`.

Opomba: `np.poly1d` spada v starejši vmesnik; sodobnejši je razred `numpy.polynomial.Polynomial` ([dokumentacija](#)), ki koeficiente pričakuje v **naraščajočem** vrstnem redu potenc: `Polynomial([5, 0, -10, 1])`. Objekt prav tako kličemo kot funkcijo, ničle pa dobimo z metodo `roots()`.

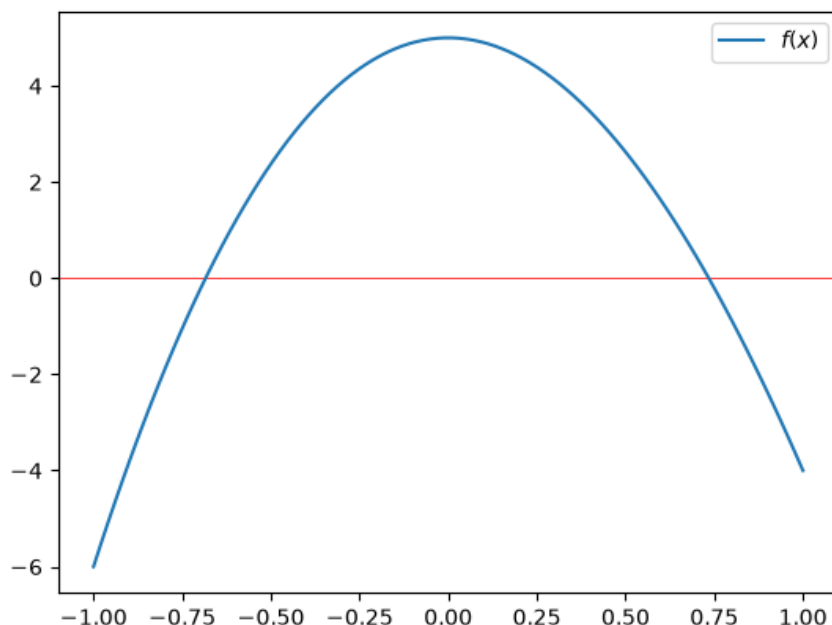
Uvozimo `numpy` in definirajmo polinom:

```
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 $f(x)$:

```
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();
```



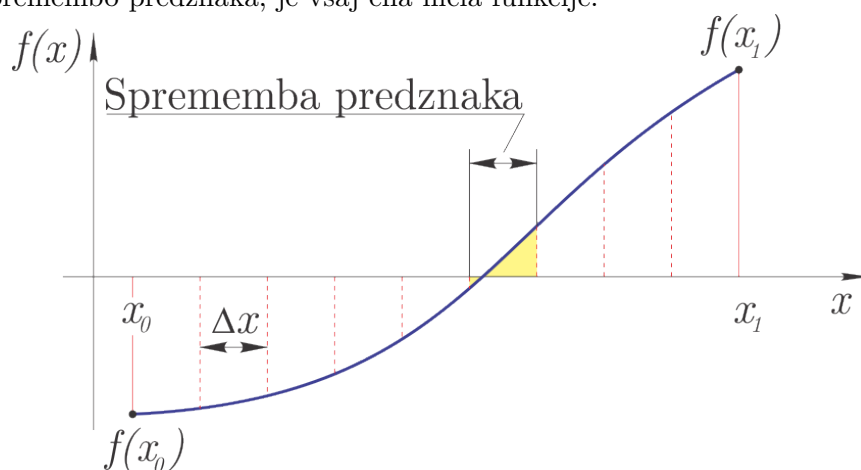
Opazimo, da so ničle funkcije $f(x)$ blizu $-0,7$ in $+0,7$. Objekt `poly1d` ima atribut `roots` ali tudi `r` (glejte dokumentacijo), ki vrne te ničle:

```
| f.r
| 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:

$$|x_{i+1} - x_i| < \varepsilon \quad \text{in} \quad |f(x_{i+1})| + |f(x_i)| < D, \quad (9.2)$$

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 si 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:

```
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:

```
rez_inkr, D = inkrementalna(f, 0., 1., 0.001)
rez_inkr
```

```
array([0.734, 0.735])
```

Ničla je izolirana z natančnostjo 0,001, preverimo še vsoto absolutnih funkcijskih vrednosti:

```
D
```

```
np.float64(0.0130715290000000304)
```

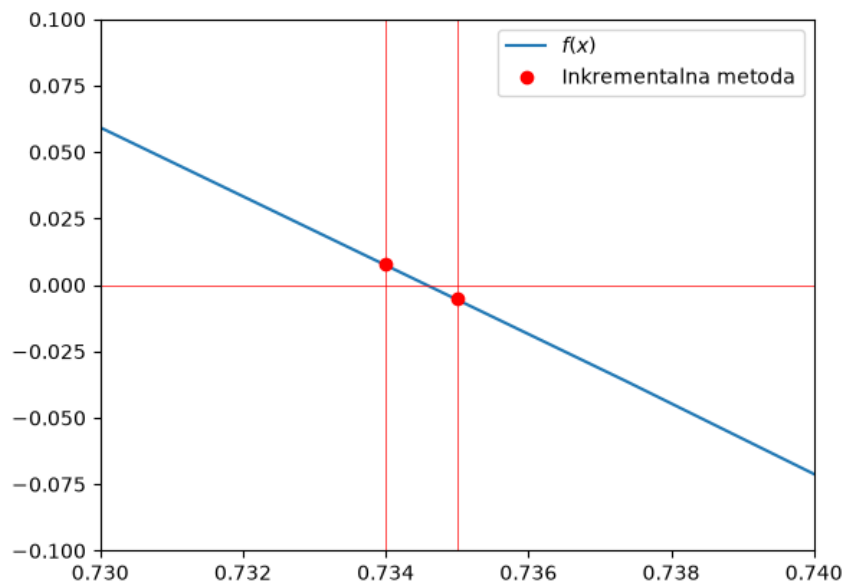
Ugotovimo, da je relativno majhna; bomo pa se s sledečimi metodami trudili rezultat bistveno izboljšati.

Pripravimo sliko:

```
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:

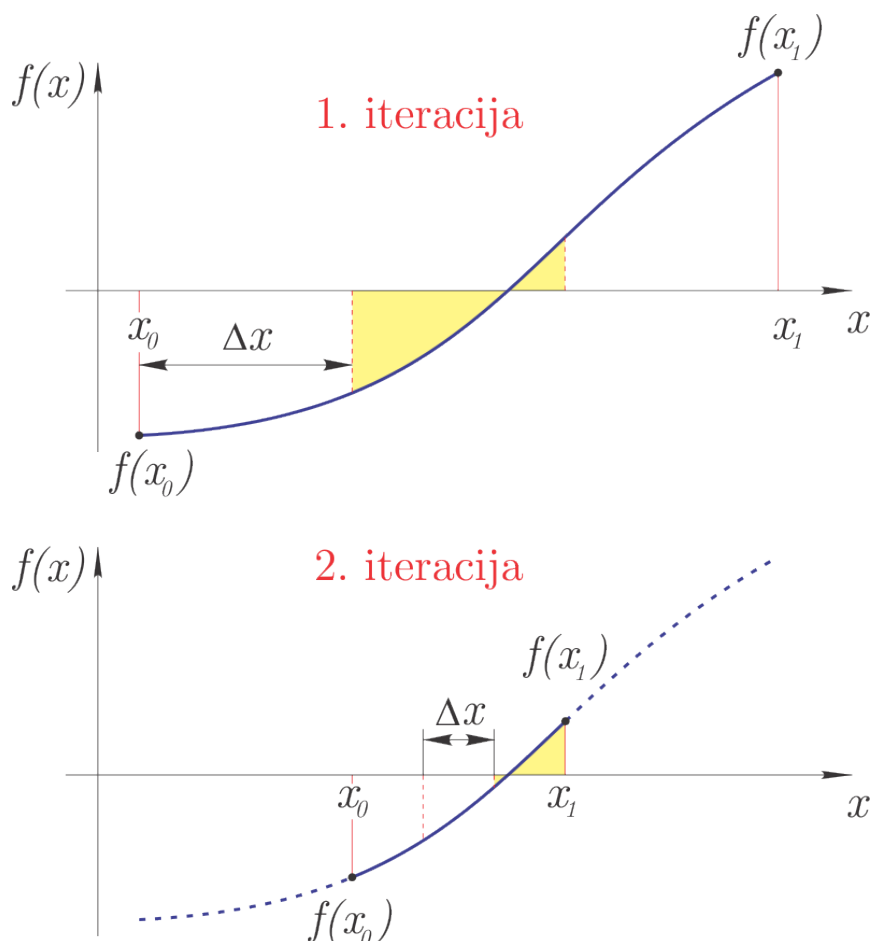
```
fig()
```



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 iteracija 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

```
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:

```
rez30, D30 = inkrementalna_super(f, 0., 1., iteracij=3)
rez30

array([0.734, 0.735])

rez_inkr

array([0.734, 0.735])
```

Seveda pa lahko natančnost bistveno izboljšamo z večanjem števila iteracij:

```
rez80, D80 = inkrementalna_super(f, 0., 1., iteracij=8)
rez80
array([0.7346035 , 0.73460351])
```

Preverimo še kriterij vsote absolutnih funkcijskih vrednosti, ki mora biti majhen:

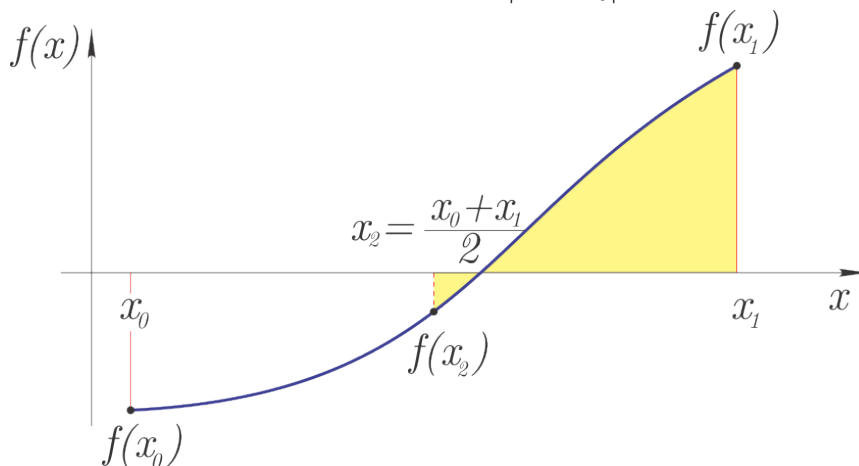
```
D80
np.float64(1.3073143190212022e -07)
```

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:

$$\varepsilon_1 = \Delta x / 2, \quad (9.3)$$

po drugem koraku:

$$\varepsilon_2 = \Delta x / 2^2 \quad (9.4)$$

in po n korakih:

$$\varepsilon_n = \Delta x / 2^n. \quad (9.5)$$

Ponavadi zahtevamo, da je rešitev podana z natančnostjo ε in iz zgornje enačbe lahko izpeljemo število potrebnih korakov bisekcijske metode:

$$n = \frac{\log\left(\frac{\Delta x}{\varepsilon}\right)}{\log(2)}. \quad (9.6)$$

Seveda je število korakov celo število.

9.4.2 Numerična implementacija

```
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) # števil 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 $1e-3$:

```
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:

```
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](#)), 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.

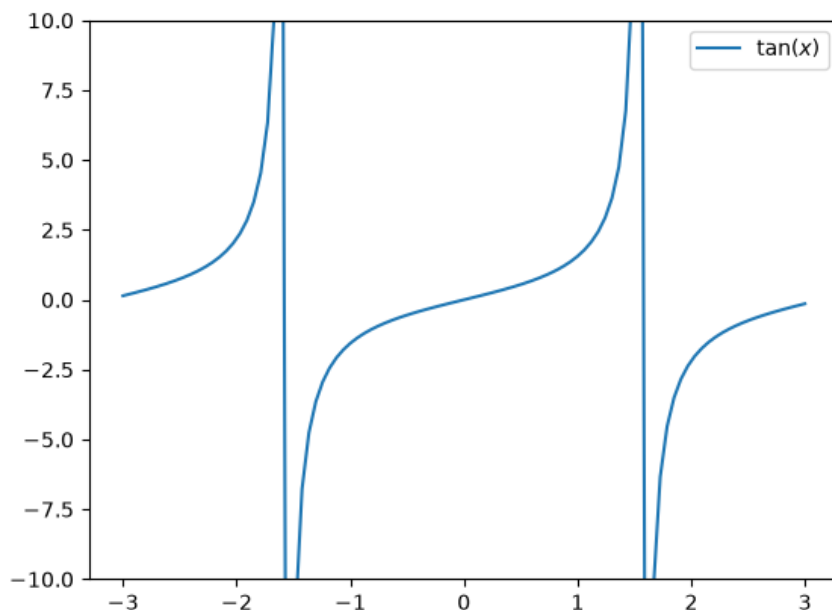
```
%%timeit
bisekcija(f, 0., 1., izpis=False)
```

```
350 \mus \pm 2.06 \mus per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)
```

Iskanje ničle v okolici pola

Poglejmo sedaj iskanje ničle funkcije `tan` v okolici pola (ničla dejansko ne obstaja):

```
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:

```
| 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 tukaj prekine):

```
| ## 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.root_scalar`

Bisekcijska metoda je *počasna*, vendar zanesljiva metoda iskanja ničel in je implementirana znotraj `scipy`. Najprej jo uvozimo:

```
| from scipy.optimize import root_scalar
|
| root_scalar(f, args=(), method='bisect', bracket=[a,b], fprime=None, fprime2=None,
| x0=None, x1=None,xtol=None, rtol=None, maxiter=None, options=None)
```

Da s funkcijo `root_scalar` priključimo bisekcijsko metodo moramo funkciji podati tri parametre: funkcijo `f` ter zaprti interval `[a, b]` (prek parametra `bracket`). 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](#).

Poglejmo uporabo:

```
| root_scalar(f, bracket=[0, 1], method='bisect')
|
| converged: True
|         flag: converged
| function_calls: 41
|       iterations: 39
|         root: 0.7346035077880515
|       method: bisect
```

in hitrost:

```
%timeit root_scalar(f, bracket=[0, 1], method='bisect')
318 \mus \pm 6.72 \mus per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)
```

Preverimo še lahko funkcijo $\tan(x)$, najprej na intervalu, kjer je funkcija zvezna:

```
root_scalar(np.tan, bracket=[ -1, 1], method='bisect')

converged: True
      flag: converged
function_calls: 3
      iterations: 1
          root: 0.0
      method: bisect
```

Potem še v okolici pola, kjer ni zvezna:

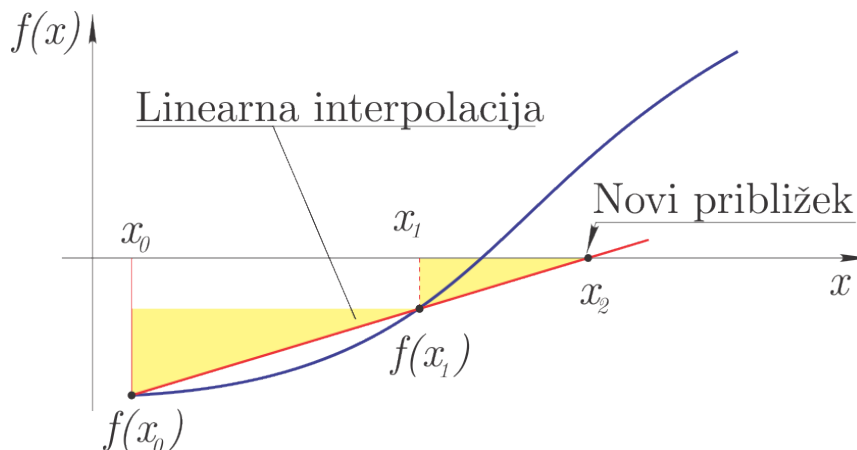
```
root_scalar(np.tan, bracket=[ -3, -1], method='bisect')

converged: True
      flag: converged
function_calls: 42
      iterations: 40
          root: -1.5707963267941523
      method: bisect
```

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:

lahko zapišemo (podobna trikotnika sta na sliki označena z rumeno):

$$\frac{f(x_1)}{x_2 - x_1} = \frac{f(x_0) - f(x_1)}{x_1 - x_0}. \quad (9.7)$$

Sledi, da je nov približek ničle:

$$x_2 = x_1 - f(x_1) \frac{x_1 - x_0}{f(x_1) - f(x_0)}. \quad (9.8)$$

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:

$$\varepsilon = |x_{n-1} - x_n| \quad (9.9)$$

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.

Če ε označimo napako približka in napako z vsakim korakom iteracije linearno zmanjšamo (C je konstanta):

$$\varepsilon_n = C \varepsilon_{n-1}^1, \quad (9.10)$$

govorimo o **redu konvergence 1** (ε ima potenco 1)!

Pri predhodno obravnavani bisekcijski metodi napako na vsakem koraku razpolovimo ($\varepsilon_n/\varepsilon_{n-1} = C = 1/2$). *Bisekcijska metoda ima red konvergence 1.*

Red konvergence *sekantne metode* je *višji* in jo je mogoče oceniti z:

$$\varepsilon_n = C \varepsilon_{n-1}^{1.618}. \quad (9.11)$$

Iz zgornje ocene sledi, da se na vsakem koraku iteracije število točnih cifer poveča za približno 60%. Ker je red konvergence višji od 1 in manjši od kvadratične, tako konvergenco imenujemo *superlinearna* konvergenca.

9.5.3 Numerična implementacija

```
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
    :param max_iter: maksimalno število iteracij preden se izvajanje prekine
    :param Dtol: največja vsota absolutnih vrednosti rešitve
    :param 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:

```
| 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

| %timeit sekantna(f, 0, 1., tol=1.e -8, izpis=False)

| 119 \mus \pm 1.89 \mus per loop (mean \pm std. dev. of 7 runs, 10,000 loops each)
```

Kakor smo zapisali zgoraj, je sekantna metoda odprtega tipa. Rešitev enačbe je lahko zunaj podanega intervala. Poglejmo si primer:

```
| 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.root_scalar`

Znotraj `scipy` je sekantna metoda definirana v okviru `scipy.optimize.root_scalar` funkcije:

```
| root_scalar(f, x0=1, method='secant')

| converged: True
|           flag: converged
| function_calls: 7
| iterations: 6
|           root: 0.7346035077893033
|           method: secant
```

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 še hitrost

```
| %timeit root_scalar(f, x0=1, method='secant')
```

```
| 155 \mus \pm 904 ns per loop (mean \pm std. dev. of 7 runs, 10,000 loops each)
```

Sorodna sekantni metodi je *Ridderjeva* metoda; v podrobnosti metode tukaj ne gremo, je pa zaprtega tipa, ima kvadratičen red konvergence ter jo kličemo s pomočjo funkcije `root_scalar`:

```
| root_scalar(f, bracket=[0, 1], method='ridder')

| converged: True
|           flag: converged
| function_calls: 14
|   iterations: 6
|         root: 0.7346035077883111
|       method: ridder
```

Brentova metoda (`brentq`) je v Scipy privzeta in priporočena izbira za iskanje ničel na intervalu. Združuje zanesljivost bisekcije in hitrost sekantne metode/inverzne kvadratne interpolacije. Uporaba in hitrost:

```
| root_scalar(f, bracket=[0, 1], method='brentq')

| converged: True
|           flag: converged
| function_calls: 9
|   iterations: 8
|         root: 0.7346035077893034
|       method: brentq
```

Primerjava hitrosti kaže, da je običajno hitrejša od bisekcije:

```
| %timeit root_scalar(f, bracket=[0, 1], method='brentq')

| 77.9 \mus \pm 427 ns per loop (mean \pm std. dev. of 7 runs, 10,000 loops each)
```

Sicer pa Scipy podpira tudi algoritem [TOMS 748](#), ki velja za asimptotično najučinkovitejšo metodo za iskanje ničel na intervalu:

```
| root_scalar(f, bracket=[0, 1], method='toms748')

| converged: True
|           flag: converged
| function_calls: 9
|   iterations: 4
|         root: 0.7346035077893033
|       method: toms748
```

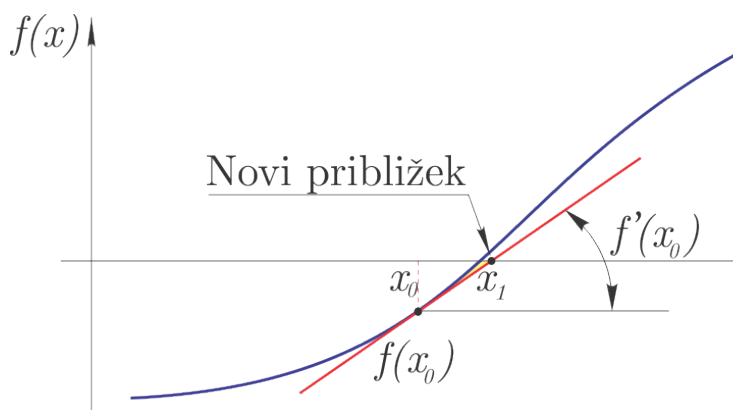
Primerjava hitrosti (v večini primerov bi naj bila najhitrejša):

```
| %timeit root_scalar(f, bracket=[0, 1], method='toms748')

| 524 \mus \pm 4.25 \mus per loop (mean \pm std. dev. of 7 runs, 1,000 loops each)
```

9.6 Newtonova metoda

Doslej predstavljene metode ne zahtevajo dodatnih odvodov funkcije; Newtonova metoda, ki si jo bomo pogledali v nadaljevanju, zahteva en začetni približek x_0 , poleg definicije funkcije $f(x)$ pa tudi njen odvod $f'(x)$. V literaturi za **Newtonovo** metodo tudi najdemo izraza **tangentna** in **Newton-Raphsonova** metoda.



Princip delovanja metode je prikazan na sliki:

Metodo bi lahko izpeljali grafično (s slike), tukaj pa si pogledjmo izpeljavo s pomočjo Taylorjeve vrste:

$$f(x_{i+1}) = f(x_i) + f'(x_i) (x_{i+1} - x_i) + \mathcal{O}((x_{i+1} - x_i)^2), \quad (9.12)$$

če naj bo pri x_{i+1} vrednost funkcije nič, potem velja:

$$0 = f(x_i) + f'(x_i) (x_{i+1} - x_i) + \mathcal{O}((x_{i+1} - x_i)^2). \quad (9.13)$$

Naredimo napako metode in zanemarimo člene višjega reda v Taylorjevi vrsti. Lahko izpeljemo:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}. \quad (9.14)$$

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 konvergence

Red konvergence Newtonove metode je kvadraten:

$$\varepsilon_n = C \varepsilon_{n-1}^2, \quad (9.15)$$

kjer je C :

$$C = -\frac{f''(x)}{2f'(x)}. \quad (9.16)$$

Konvergenca je torej hitra, v vsaki novi iteraciji se število točnih števk v približku podvoji.

9.6.2 Numerična implementacija

```
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
    :param max_iter: maksimalno število iteracij preden se izvajanje prekine
    :param Dtol: največja vsota absolutnih vrednosti rešitve
    :param 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 :

```
def f(x):
    return x**3 - 10*x**2 + 0*x + 5
def df(x):
    return 3*x**2 - 20*x
```

Izračunajmo sedaj ničlo:

```
newtonova(fun=f, dfun=df, x0=1, tol=1e-8);
```

```
Rešitev: 0.73460351, število iteracij: 5, D: 0.00000000
```

Preverimo hitrost izvajanja:

```
%timeit newtonova(fun=f, dfun=df, x0=1, tol=1e-8, izpis=False)
```

```
3.98 \mus \pm 16.8 ns per loop (mean \pm std. dev. of 7 runs, 100,000 loops each)
```

9.6.3 Uporaba `scipy.optimize.root_scalar`

Znotraj `scipy` je Newtonova metoda definirana v okviru `scipy.optimize.root_scalar` funkcije:

```
root_scalar(f, x0=1, method='newton', fprime=df)
```

```
converged: True
      flag: converged
function_calls: 10
  iterations: 5
        root: 0.7346035077893033
      method: newton
```

In izmerimo hitrost:

```
%timeit root_scalar(f, x0=1, method='newton', fprime=df)
88.1 \mus \pm 754 ns per loop (mean \pm std. dev. of 7 runs, 10,000 loops each)
```

Če pri `root_scalar` podamo tudi drugi odvod `fprime2`, se uporabi Halleyeva metoda, ki temelji na izrazu:

$$x_{n+1} = x_n - \frac{2f(x_n)f'(x_n)}{2f'^2(x_n) - f(x_n)f''(x_n)} \quad (9.17)$$

```
def ddf(x):
    return 6*x - 20

root_scalar(f, x0=1, method='halley', fprime=df, fprime2=ddf)

converged: True
  flag: converged
function_calls: 10
  iterations: 3
      root: 0.7346035077893033
  method: halley
```

In hitrost:

```
%timeit root_scalar(f, x0=1, method='halley', fprime=df, fprime2=ddf)
62.1 \mus \pm 799 ns per loop (mean \pm std. dev. of 7 runs, 10,000 loops each)
```

9.7 Reševanje sistemov nelinearnih enačb

Rešujemo sistem enačb, ki ga v vektorski obliki zapišemo takole:

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}. \quad (9.18)$$

V skalarni obliki zgornji vektorski izraz zapišemo:

$$\begin{aligned} f_0(x_0, x_1, \dots, x_{n-1}) &= 0 \\ f_1(x_0, x_1, \dots, x_{n-1}) &= 0 \\ &\vdots \\ f_{n-1}(x_0, x_1, \dots, x_{n-1}) &= 0. \end{aligned} \quad (9.19)$$

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 najenostavnejša 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:

$$f_i(\mathbf{x} + \Delta\mathbf{x}) = f_i(\mathbf{x}) + \sum_{j=0}^{n-1} \frac{\partial f_i}{\partial x_j} \Delta x_j + \mathcal{O}(\|\Delta\mathbf{x}\|^2). \quad (9.20)$$

Naredimo napako metode, ko zanemarimo člene drugega in višjih redov ter zapišemo izraz v matrični obliki:

$$\mathbf{f}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{f}(\mathbf{x}) + \mathbf{J}(\mathbf{x}) \Delta\mathbf{x}, \quad (9.21)$$

kjer je $\mathbf{J}(\mathbf{x})$ Jakobijeva matrika pri vrednostih $\mathbf{x}$. Elementi Jakobijeve matrike so:

$$J_{ij} = \frac{\partial f_i}{\partial x_j}. \quad (9.22)$$

Če naj bo $\mathbf{x} + \Delta\mathbf{x}$ rešitev sistema enačb, mora veljati:

$$\mathbf{0} = \mathbf{f}(\mathbf{x}) + \mathbf{J}(\mathbf{x}) \Delta\mathbf{x} \quad (9.23)$$

in torej sledi:

$$\mathbf{J}(\mathbf{x}) \Delta\mathbf{x} = -\mathbf{f}(\mathbf{x}). \quad (9.24)$$

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:

$$\frac{\partial f_i}{\partial x_j} \approx \frac{f_i(\mathbf{x} + \mathbf{e}_j h) - f_i(\mathbf{x})}{h}, \quad (9.25)$$

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](#)).

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:

$$\sum_{j=0}^{n-1} |x_{i,j} - x_{i-1,j}| < \varepsilon, \quad (9.26)$$

kjer je i indeks iteracije in j indeks elementa.

9.7.2 Uporaba `scipy.optimize.root`

Funkcija `scipy.optimize.root` ima obsežno [dokumentacijo](#) 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:

```
| from scipy.optimize import root
```

Poglejmo si uporabo na zgledu (gre za zgled na str. 76, Jože Petrišič, Reševanje enačb, 1996, FS, UNI-LJ):

$$f_0(\mathbf{x}) = x_0^2 + x_0 x_1 - 10 = 0, \quad (9.27)$$

$$f_1(\mathbf{x}) = x_1 + 3 x_0 x_1^2 - 57 = 0, \quad (9.28)$$

kjer je vektor $\mathbf{x} = [x_0, x_1]$.

Najprej definirajmo Python funkcijo, ki vrne seznam rezultatov funkcij $[f_0(\mathbf{x}), f_1(\mathbf{x})]$:

```
| 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:

```
| def J(x):
|     return np.array([[2*x[0]+x[1], x[0]],
|                     [3*x[1]**2, 1+6*x[0]*x[1]]])
```

Uporabimo začetne vrednosti $x_0 = 1,5$ in $x_1 = 3,5$ ter rešimo problem:

```
| rešitev = root(fun=func, x0=[1.5, 3.5], jac=J)
| rešitev
```

```
| message: The solution converged.
| success: True
| status: 1
|   fun: [ -6.972e -12  3.148e -11]
|     x: [ 2.000e+00  3.000e+00]
| method: hybr
|   nfev: 12
|   njev: 1
|  fjac: [[ -2.235e -01 -9.747e -01]
|         [ 9.747e -01 -2.235e -01]]
|     r: [ -2.954e+01 -3.782e+01 -6.940e+00]
|    qtf: [ -2.836e -08 -1.324e -08]
```

Funkcija `root` vrne obsežen rezultat. Najbolj pomembna sta atribut `x`, ki predstavlja iskano rešitev, in atribut `success`, ki pove, ali je rešitev konvergirala:

```
| rešitev.x
```

```
| array([2., 3.])
```

```
| rešitev.success
```

```
| True
```

```
| import matplotlib.pyplot as plt

| # Priprava mreže točk
| x0_range = np.linspace( -5, 5, 400)
| x1_range = np.linspace( -5, 5, 400)
| X0, X1 = np.meshgrid(x0_range, x1_range)

| # Definicija funkcij sistema
| F0 = X0**2 + X0*X1 - 10
```

```

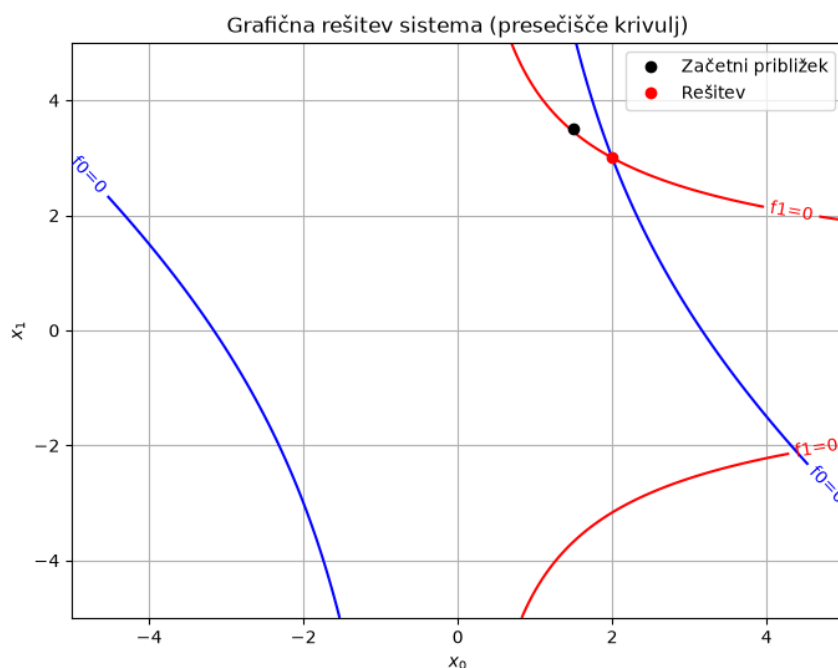
F1 = X1 + 3*X0*X1**2 - 57

plt.figure(figsize=(8, 6))

# Izris ničelnih izolinij (kjer je f(x) = 0)
CS0 = plt.contour(X0, X1, F0, levels=[0], colors='blue')
CS1 = plt.contour(X0, X1, F1, levels=[0], colors='red')

# Označbe
plt.clabel(CS0, inline=1, fontsize=10, fmt='f0=0')
plt.clabel(CS1, inline=1, fontsize=10, fmt='f1=0')
plt.plot(1.5, 3.5, 'ko', label='Začetni približek')
plt.plot(rešitev.x[0], rešitev.x[1], 'ro', label='Rešitev')
plt.xlabel('$x_0$')
plt.ylabel('$x_1$')
plt.title('Grafična rešitev sistema (presečišče krivulj)')
plt.grid(True)
plt.legend()
plt.show()

```



9.8 Dodatno

Tisti, ki ste navdušeni nad [Raspberry Pi](#) in uporabljate njihovo kamero (npr. [tole brez infrardečega filtra](#)), vas bo morebiti zanimala knjižnica [picamera](#).

9.8.1 Uporaba `sympy.solve` za reševanje enačb

Za manjše sisteme lahko rešitev najdemo tudi simbolno. Poglejmo si zgornji primer:

```

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

```

$$[x^2 + xy - 10, 3xy^2 + y - 57] \quad (9.29)$$

```
| sol = sym.solve(funkcije, x, y)
| print(f'Stevílo rešitev: {len(sol)}')
| print(f'Prva rešitev: {sol[0]}')
```

```
| Števílo rešitev: 4
| Prva rešitev: (2, 3)
```

Poglavje 10

Numerično odvajanje

10.1 Uvod

Vsako elementarno funkcijo lahko analitično odvajamo. Definicija odvoda je:

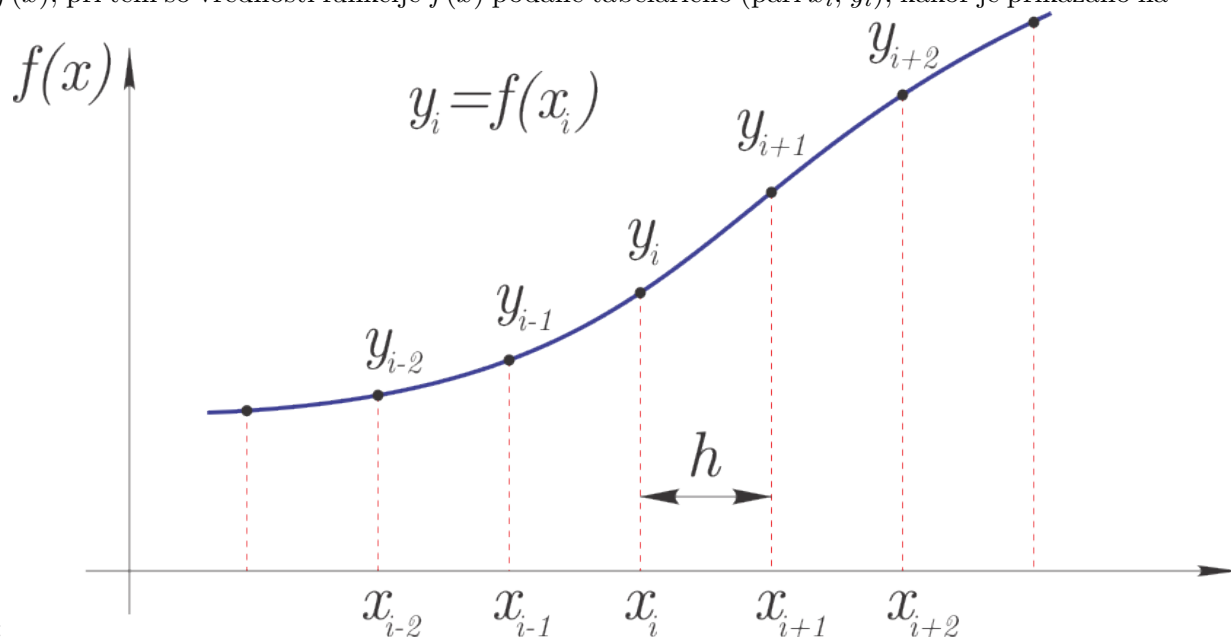
$$f'(x) = \lim_{\Delta x \rightarrow 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}. \quad (10.1)$$

Neposredna uporaba zgornje enačbe vodi v odštevanje zelo podobnih funkcijskih vrednostih ($f(x + \Delta x)$, $f(x)$), obremenjenih z zaokrožitveno napako, ki jih delimo z majhno vrednostjo Δx ; posledično ima odvod bistveno manj signifikantnih števk kakor pa funkcijske vrednosti. Numeričnemu odvajanju se izognemo, če imamo to možnost; je pa v nekaterih primerih (npr. reševanje diferencialnih enačb) nepogrešljivo orodje!

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**.

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 (pari x_i , y_i), kakor je prikazano na



sliki:

Najprej se bomo osredotočili na ekvidistantno, s korakom h , razporejene vrednosti x_i ; vrednosti funkcije pa bodo $y_i = f(x_i)$.

Glede na zgornjo definicijo odvoda, bi prvi odvod (za mesto i) lahko zapisali:

$$y'_i = \frac{y_{i+1} - y_i}{h}, \quad (10.2)$$

kjer je $h = x_{i+1} - x_i$. S preoblikovanjem enačbe:

$$y'_i = -\frac{y_i}{h} + \frac{y_{i+1}}{h}, \quad (10.3)$$

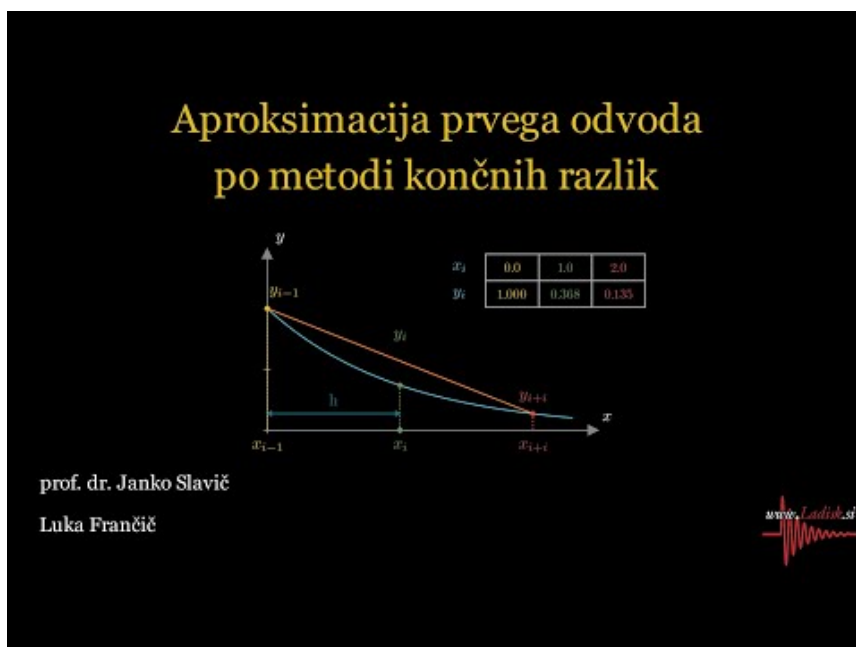
lahko tudi rečemo, da za prvi odvod funkcije na mestu i , **utežimo** funkcijsko vrednost pri i z $-1/h$ in funkcijsko vrednost pri $i+1$ z $+1/h$.

V nadaljevanju si bomo pogledali teoretično ozadje kako določimo ustrezne **uteži** za različne stopnje odvodov, katere možnosti pri tem imamo in kako to vpliva na red natančnosti.

10.2 Aproksimacija prvega odvoda po metodi končnih razlik

Za uvod si oglejmo spodnji video:

```
from IPython.display import YouTubeVideo
YouTubeVideo('YYuGL -VP2BE', width=800, height=300)
```



Odvod $f'(x)$ lahko aproksimiramo na podlagi razvoja Taylorjeve vrste. To metodo imenujemo **metoda končnih razlik** ali tudi **diferenčna metoda**.

Razvijmo **Taylorjevo vrsto naprej** (naprej, zaradi člena $+h$):

$$f(x+h) = \sum_{n=0}^{\infty} \frac{h^n}{n!} \frac{d^n}{dx^n} f(x) = f(x) + h f'(x) + \underbrace{\frac{h^2}{2} f''(x) + \dots}_{\mathcal{O}(h^2)} \quad (10.4)$$

Člen $\mathcal{O}(h^2)$ označuje napako drugega reda. Če iz enačbe izrazimo prvi odvod:

$$f'(x) = \frac{1}{h} (f(x+h) - f(x)) - \underbrace{\frac{h}{2} f''(x) + \dots}_{\mathcal{O}(h^1)} \quad (10.5)$$

Ugotovimo, da lahko ocenimo prvi odvod v točki x_i (to je: $f'_o(x_i)$) na podlagi dveh zaporednih funkcijskih vrednosti:

$$f'_o(x_i) = \frac{1}{h} (y_{i+1} - y_i) \quad (10.6)$$

in pri tem naredimo **napako metode**, ki je prvega reda $\mathcal{O}(h^1)$.

Uporabili smo $y_i = f(x_i)$ (glejte sliko zgoraj).

Napaka je:

$$e = -\frac{h}{2} f''(\xi), \quad (10.7)$$

kjer je ξ neznana vrednost na intervalu $[x_i, x_{i+1}]$ in smo zanemarili višje člene.

Velja torej izraz:

$$f'(x_i) = f'_o(x_i) + e \quad (10.8)$$

Sedaj si pogledjmo, kako pridemo do istega rezultata s strojno izpeljavo; najprej uvozimo **sympy**:

```
import sympy as sym
sym.init_printing()
```

Definirajmo simbole:

```
f = sym.Function('f')
x, h = sym.symbols('x, h')
```

Nato nadaljujemo z razvojem **Taylorjeve vrste naprej** (angl. *forward Taylor series*):

```
f(x+h).series(h, n=2)
```

$$f(x) + h \left. \frac{d}{d\xi_1} f(\xi_1) \right|_{\xi_1=x} + \mathcal{O}(h^2) \quad (10.9)$$

Člen $\mathcal{O}(h^2)$ vsebuje člene drugega in višjega reda. V zgornji enačbi je uporabljena začasna spremenljivko za odvajanje ξ_1 ; izvedmo odvajanje in vstavimo $\xi_1 = x$:

```
f(x+h).series(h, n=2).doit()
```

$$f(x) + h \frac{d}{dx} f(x) + \mathcal{O}(h^2) \quad (10.10)$$

Zapišemo enačbo:

```
enačba = sym.Eq(f(x+h), f(x+h).series(h, n=2).doit())
enačba
```

$$f(h+x) = f(x) + h \frac{d}{dx} f(x) + \mathcal{O}(h^2) \quad (10.11)$$

Rešimo jo za prvi odvod $f'(x)$:

```
f1_naprej_točno = sym.solve(enačba, f(x).diff(x))[0]
f1_naprej_točno
```

$$\frac{f(h+x) - f(x) + \mathcal{O}(h^2)}{h} \quad (10.12)$$

V kolikor odvoda drugega in višjih redov ne upoštevamo, smo naredili torej napako:

```
f1_naprej_0 = f1_naprej_točno.expand().get0()
f1_naprej_0
```

$$O(h) \quad (10.13)$$

Napaka $\mathcal{O}(h^1)$ je torej prvega reda in če ta člen zanemarimo, naredimo *napako metode* in dobimo oceno odvoda:

```
f1_naprej_ocena = f1_naprej_točno.expand().remove0()
f1_naprej_ocena
```

$$-\frac{f(x)}{h} + \frac{f(h+x)}{h} \quad (10.14)$$

Ugotovimo, da gre za isti izraz, kakor smo ga izpeljali zgoraj, torej je:

$$y'_i = \frac{1}{h} (-y_i + y_{i+1}). \quad (10.15)$$

Uteži torej so:

Odvod $\downarrow \setminus$ Vrednosti $\rightarrow$	y_i	y_{i+1}
$y'_i = \frac{1}{h}$	-1	1

10.3 Centralna diferenčna shema

10.3.1 Odvod $f'(x)$

Najprej si pogledjmo razvoj **Taylorjeve vrste nazaj** (angl. *backward Taylor series*):

```
| f(x -h).series(h, n=3).doit()
```

$$f(x) - h \frac{d}{dx} f(x) + \frac{h^2 \frac{d^2}{dx^2} f(x)}{2} + O(h^3) \quad (10.16)$$

Ugotovimo, da se pri razliki vrste naprej in nazaj odštevajo členi sodega reda; definirajmo:

```
def razlika(n=3):
    return f(x+h).series(h, n=n).doit() - f(x -h).series(h, n=n).doit()
razlika(n=3)
```

$$2h \frac{d}{dx} f(x) + O(h^3) \quad (10.17)$$

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:

```
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:

```
| f1_cent_ocena
```

$$-\frac{f(-h+x)}{2h} + \frac{f(h+x)}{2h} \quad (10.18)$$

Ali:

$$y'_i = \frac{1}{2h} (-y_{i-1} + y_{i+1}) \quad (10.19)$$

Uteži torej so:

Odvod ↓ \ Vrednosti	y_{i-1}	y_i	y_{i+1}
$\rightarrow$			
$y'_i = \frac{1}{2h}$	-1	0	1

Napaka metode pa je torej drugega reda:

```
| f1_cent_0
```

$$O(h^2) \quad (10.20)$$

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:

```
| 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:

```
| f1_naprej_ocena # da se spomnimo
```

$$-\frac{f(x)}{h} + \frac{f(h+x)}{h} \quad (10.21)$$

```
| f1_naprej0 = (y0[1:] - y0[: -1])/h0 # korak h_0
| f1_naprej1 = (y1[1:] - y1[: -1])/h1 # korak h_1
```

Izračunajmo napako pri $x = 1,0$:

```
| f1_točno0 - f1_naprej0[1]
```

$$-0.135335283236613 \quad (10.22)$$

```
| f1_točno1 - f1_naprej1[1] # korak h1
```

$$-0.0177958664378144 \quad (10.23)$$

Potrdimo lahko, da je napaka pri koraku $h/10$ res približno $1/10$ tiste pri koraku h .
Poglejmo sedaj še napako za centralno diferenčno shemo, ki je drugega reda:

```
| f1_cent_ocena
```

$$-\frac{f(-h+x)}{2h} + \frac{f(h+x)}{2h} \quad (10.24)$$

```
| f1_cent0 = (y0[2:] - y0[: -2])/(2*h0) # korak h_0
| f1_cent1 = (y1[2:] - y1[: -2])/(2*h1) # korak h_1
```

Analizirajmo napako:

```
| f1_točno0 - f1_cent0[0] # korak h0
```

$$0.0644529172102513 \quad (10.25)$$

```
| f1_točno1 - f1_cent1[0] # korak h1
```

$$0.000613439041155517 \quad (10.26)$$

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:

```
| def vsota(n=3):
|     return f(x+h).series(h, n=n).doit() + f(x -h).series(h, n=n).doit()
| vsota(n=4)
```

$$2f(x) + h^2 \frac{d^2}{dx^2} f(x) + O(h^4) \quad (10.27)$$

Določimo drugi odvod:

```
| 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:

```
| f2_cent_ocena
```

$$-\frac{2f(x)}{h^2} + \frac{f(-h+x)}{h^2} + \frac{f(h+x)}{h^2} \quad (10.28)$$

Ali:

$$y_i'' = \frac{1}{h^2} (y_{i-1} - 2y_i + y_{i+1}) \quad (10.29)$$

Uteži torej so:

Odvod ↓ \ Vrednosti	y_{i-1}	y_i	y_{i+1}
→			
$y_i'' = \frac{1}{h^2}$	1	-2	1

Napaka metode pa je ponovno drugega reda:

```
| f2_cent_0
```

$$O(h^2) \quad (10.30)$$

10.3.4 Odvod $f'''(x)$

Če želimo določiti tretji odvod, moramo Taylorjevo vrsto razviti do stopnje 5:

```
| eq_h = sym.Eq(f(x+h) - f(x -h), razlika(n=5))
| eq_h
```

$$-f(-h+x) + f(h+x) = 2h \frac{d}{dx} f(x) + \frac{h^3 \frac{d^3}{dx^3} f(x)}{3} + O(h^5) \quad (10.31)$$

Uporaba 1. odvoda, ki smo ga izpeljali zgoraj, nam ne bi koristila, saj je red napake $\mathcal{O}(h^2)$, kar pomeni, da bi v zgornji enačbi pri deljenju s h^3 dobili $\mathcal{O}(h^{-1})$.

Uporabimo trik: ponovimo razvoj, vendar na podlagi dodatnih točk, ki sta od x oddaljeni za $2h$ in $-2h$:

```
| eq_2h = eq_h.subs(h, 2*h)
| eq_2h
```

$$-f(-2h+x) + f(2h+x) = 4h \frac{d}{dx} f(x) + \frac{8h^3 \frac{d^3}{dx^3} f(x)}{3} + O(h^5) \quad (10.32)$$

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:

```
| 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:

```
| f3_cent_ocena
```

$$-\frac{f(-2h+x)}{2h^3} + \frac{f(-h+x)}{h^3} - \frac{f(h+x)}{h^3} + \frac{f(2h+x)}{2h^3} \quad (10.33)$$

Ali:

$$y_i''' = \frac{1}{h^3} (-y_{i-2}/2 + y_{i-1} - y_{i+1} + y_{i+2}/2) \quad (10.34)$$

Uteži torej so:

Odvod ↓ \	y_{i-2}	y_{i-1}	y_i	y_{i+1}	y_{i+2}
Vrednosti →					
$y_i''' = \frac{1}{h^3} \cdot$	-0.5	1	0	-1	0.5

Potrdimo, da je napaka metode drugega reda:

```
| f3_cent_0
```

$$O(h^2) \quad (10.35)$$

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:

```
| eq_h = sym.Eq(f(x+h)+f(x-h), vsota(n=6))
| eq_h
```

$$f(-h+x) + f(h+x) = 2f(x) + h^2 \frac{d^2}{dx^2} f(x) + \frac{h^4 \frac{d^4}{dx^4} f(x)}{12} + O(h^6) \quad (10.36)$$

Pripravimo dodatno enačbo na podlagi točk, ki sta od x oddaljeni za $2h$ in $-2h$:

```
| eq_2h = eq_h.subs(h, 2*h)
| eq_2h
```

$$f(-2h+x) + f(2h+x) = 2f(x) + 4h^2 \frac{d^2}{dx^2} f(x) + \frac{4h^4 \frac{d^4}{dx^4} f(x)}{3} + O(h^6) \quad (10.37)$$

Iz dveh enačb določimo 4. odvod:

```
| 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:

```
| f4_cent_ocena
```

$$\frac{6f(x)}{h^4} + \frac{f(-2h+x)}{h^4} - \frac{4f(-h+x)}{h^4} - \frac{4f(h+x)}{h^4} + \frac{f(2h+x)}{h^4} \quad (10.38)$$

Ali:

$$y_i^{(4)} = \frac{1}{h^4} (y_{i-2} - 4y_{i-1} + 6y_i - 4y_{i+1} + y_{i+2}) \quad (10.39)$$

Uteži torej so:

Odvod ↓ \	y_{i-2}	y_{i-1}	y_i	y_{i+1}	y_{i+2}
Vrednosti →					
$y_i^{(4)} = \frac{1}{h^4} \cdot$	1	-4	6	-4	1

Potrdimo, da je napaka metode drugega reda:

```
| f4_cent_0
```

$$O(h^2) \quad (10.40)$$

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. Če ste neučakani, lahko skočite na tabelo spodaj. Z branjem nadaljujte, če pa želite spoznati, kako predhodno izpeljane izraze strojno uredimo.

Najprej zberimo vse ocene odvodov v seznam:

```
| odvodi = [f1_cent_ocena, f2_cent_ocena, f3_cent_ocena, f4_cent_ocena]
| odvodi
```

$$\left[-\frac{f(-h+x)}{2h} + \frac{f(h+x)}{2h}, -\frac{2f(x)}{h^2} + \frac{f(-h+x)}{h^2} + \frac{f(h+x)}{h^2}, -\frac{f(-2h+x)}{2h^3} + \frac{f(-h+x)}{h^3} - \frac{f(h+x)}{h^3} + \frac{f(2h+x)}{2h^3} \right] \quad (10.41)$$

Na razpolago imamo 5 funkcijskih vrednosti (pri legah $x-2h, x-h, x, x+h, x+2h$), ki jih damo v seznam:

```
| 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:

```
| f1_cent_ocena.expand().coeff(funkcijske_vrednosti[1])
```

$$-\frac{1}{2h} \quad (10.42)$$

Sedaj posplošimo in izračunajmo uteži za vse funkcijske vrednosti in za vse ocene odvodov:

```
| centralna_diff_shema = [[odvod.expand().coeff(fv) for fv in funkcijske_vrednosti] \
|                          for odvod in odvodi]
| centralna_diff_shema
```

$$\left[\left[0, -\frac{1}{2h}, 0, \frac{1}{2h}, 0 \right], \left[0, \frac{1}{h^2}, -\frac{2}{h^2}, \frac{1}{h^2}, 0 \right], \left[-\frac{1}{2h^3}, \frac{1}{h^3}, 0, -\frac{1}{h^3}, \frac{1}{2h^3} \right], \left[\frac{1}{h^4}, -\frac{4}{h^4}, \frac{6}{h^4}, -\frac{4}{h^4}, \frac{1}{h^4} \right] \right] \quad (10.43)$$

Zgornje povzetke lahko tudi zapišemo v tabelarični obliki:

Odvod ↓ \ Vrednosti →	y_{i-2}	y_{i-1}	y_i	y_{i+1}	y_{i+2}
$y'_i = \frac{1}{h}$	0	-0.5	0	0.5	0
$y''_i = \frac{1}{h^2}$	0	1	-2	1	0
$y'''_i = \frac{1}{h^3}$	-0.5	1	0	-1	0.5
$y^{(4)}_i = \frac{1}{h^4}$	1	-4	6	-4	1

Prikazana centralna diferenčna shema ima napako 2. reda $\mathcal{O}(h^2)$.

Opomba: v kolikor vas zanima posplošitev diferenčne metode, prosim glejte paket [findiff](#).

10.3.7 Izboljšan približek - Richardsonova ekstrapolacija

Če je točen odvod je izračunan kot:

$$f'(x_i) = f'_o(x_i) + e, \quad (10.44)$$

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:

$$f'(x_i) = f'_o(x_i, h) + K h^n, \quad (10.45)$$

kjer je K neznana konstanta.

Če korak razpolovimo in predpostavimo, da se K ne spremeni, velja:

$$f'(x_i) = f'_o\left(x_i, \frac{h}{2}\right) + K \left(\frac{h}{2}\right)^n. \quad (10.46)$$

Iz obeh enačb izločimo konstanto K in določimo izboljšan približek:

$$\bar{f}'(x_i) = \frac{2^n f'_o\left(x_i, \frac{h}{2}\right) - f'_o(x_i, h)}{2^n - 1} \quad (10.47)$$

Zgled

Poglejmo si zgled $f(x) = \sin(x)$ (analitični odvod je: $f'(x) = \cos(x)$):

```
x = np.linspace(0, 2*np.pi, 9)
y = np.sin(x)
```

Pri koraku h imamo funkcijske vrednosti definirane pri:

```
x
array([0., 0.78539816, 1.57079633, 2.35619449, 3.14159265,
       3.92699082, 4.71238898, 5.49778714, 6.28318531])
```

Numerični odvod pri $x = \pi$:

```
x[4]
```

$$3.14159265358979 \quad (10.48)$$

in koraku h je

```
h = x[1] - x[0]
f_ocena_h = ( -0.5*y[3] + 0.5*y[5])/h
f_ocena_h
```

$$-0.900316316157106 \quad (10.49)$$

Izračun pri koraku $2h$:

```
h2 = x[2] - x[0]
f_ocena_2h = ( -0.5*y[2] + 0.5*y[6])/h2
f_ocena_2h
```

$$-0.636619772367581 \quad (10.50)$$

Izračunajmo izboljšano oceno za $x = \pi$:

```
f_ocena_izboljsana = (2**2 * f_ocena_h - f_ocena_2h)/(2**2 -1)
f_ocena_izboljsana
```

$$-0.988215164086948 \quad (10.51)$$

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 $\mathcal{O}(h^1)$:

Odvod $\downarrow \setminus$	y_i	y_{i+1}	y_{i+2}	y_{i+3}	y_{i+4}
Vrednosti $\rightarrow$					
$y'_i = \frac{1}{h}$	-1	1	0	0	0
$y''_i = \frac{1}{h^2}$	1	-2	1	0	0
$y'''_i = \frac{1}{h^3}$	-1	3	-3	1	0
$y^{(4)}_i = \frac{1}{h^4}$	1	-4	6	-4	1

Diferenčna shema nazaj z redom napake $\mathcal{O}(h^2)$:

Odvod ↓ \	y_i	y_{i+1}	y_{i+2}	y_{i+3}	y_{i+4}	y_{i+5}
Vrednosti →						
$y'_i = \frac{1}{2h} \cdot$	-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
$y^{(4)}_i = \frac{1}{h^4} \cdot$	3	-14	26	-24	11	-2

10.4.2 Diferenčna shema nazaj

Diferenčna shema nazaj z redom napake $\mathcal{O}(h^1)$:

Odvod ↓ \	y_{i-4}	y_{i-3}	y_{i-2}	y_{i-1}	y_i
Vrednosti →					
$y'_i = \frac{1}{h} \cdot$	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
$y^{(4)}_i = \frac{1}{h^4} \cdot$	1	-4	6	-4	1

Diferenčna shema nazaj z redom napake $\mathcal{O}(h^2)$:

Odvod ↓ \	y_{i-5}	y_{i-4}	y_{i-3}	y_{i-2}	y_{i-1}	y_i
Vrednosti →						
$y'_i = \frac{1}{2h} \cdot$	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
$y^{(4)}_i = \frac{1}{h^4} \cdot$	-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](#)):

```
| 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 **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](#).

Za odvajanje funkcije (in ne tabele vrednosti), glejte: [scipy.diffentiate](#).

10.5.1 Zgled

Pogledali si bomo zgled, kako uporabimo **uteži**, funkcijo `gradient` in posebnosti na robovih. Najprej pripravimo tabelo podatkov:

```
x, h = np.linspace(0, 1, 20, retstep=True)
y = np.sin(2*np.pi*x)
```

Uteži diferenčnih shem:

```
centralna = np.array([-0.5, 0, 0.5]) # uteži centralne sheme; glejte tudi paket
findiff (spodaj)
naprej = np.array([-3/2, 2, -1/2])
nazaj = np.array([1/2, -2, 3/2])
```

Sedaj izvedemo odvod notranjih točk (prvi način je z izpeljevanjem seznamov, drugi je vektoriziran):

```
odvod_notranje = np.array([y[i -1:i+2] @ centralna/h for i in range(1, len(x) -1)]) #
izpeljevanje seznamov
odvod_notranje = np.convolve(y, centralna[: -1], mode='valid') / h # vektoriziran
```

Na robovih uporabimo diferenčno shemo naprej oziroma nazaj:

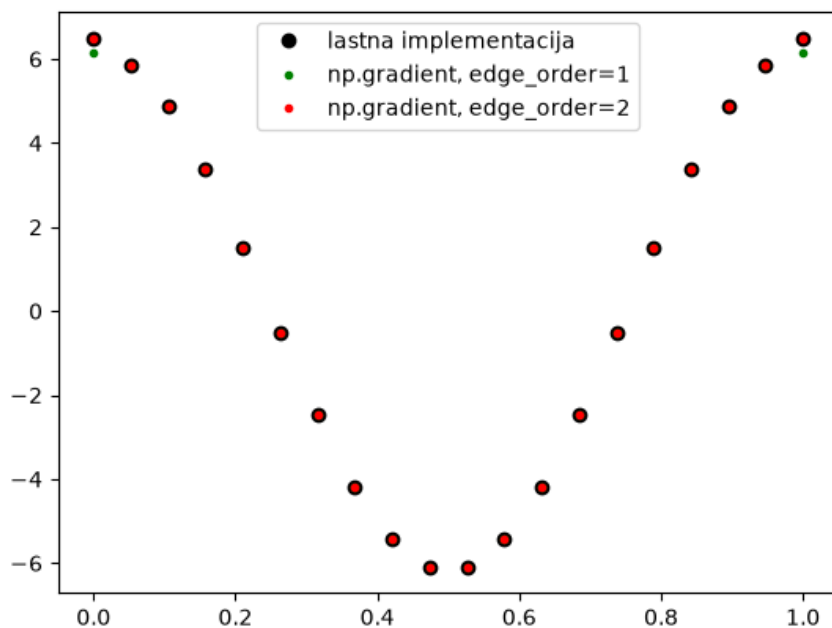
```
odvod_prva = y[:len(naprej)] @ naprej / h # naprej
odvod_zadnja = y[-len(nazaj):] @ nazaj / h # nazaj
```

Sestavimo rezultat:

```
odvod_cel = np.hstack([odvod_prva, odvod_notranje, odvod_zadnja])
```

Prikažemo rezultat skupaj z rezultatom funkcije `np.gradient`:

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(x, odvod_cel, 'ko', lw=3, label='lastna implementacija')
plt.plot(x, np.gradient(y, h), 'g.', label='np.gradient, edge_order=1')
plt.plot(x, np.gradient(y, h, edge_order=2), 'r.', label='np.gradient, edge_order=2')
plt.legend();
```



10.5.2 Uporaba `numpy.diff`

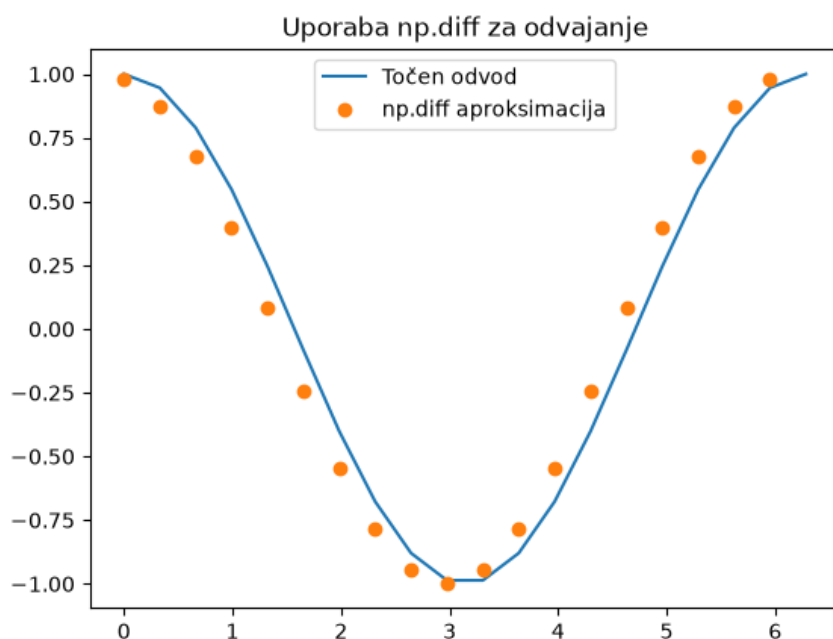
Dajmo tukaj predstaviti še pogosto uporabljeno funkcijo `numpy.diff` ([dokumentacija](#)) izračuna n -to diskretno razliko vzdolž podane osi. Prva razlika je podana kot $a[i+1] - a[i]$. To je osnovna operacija za numerično odvajanje (gre za odvod prve stopnje natančnosti), vendar moramo rezultat deliti s korakom h .

Pri uporabi `np.diff` se dolžina polja zmanjša za 1.

```
x_d = np.linspace(0, 2*np.pi, 20)
y_d = np.sin(x_d)
h_d = x_d[1] - x_d[0]

# Prvi odvod z np.diff (naprej diferenca)
dy = np.diff(y_d) / h_d
x_diff = x_d[:-1] # x os se zmanjša

plt.figure()
plt.plot(x_d, np.cos(x_d), label='Točen odvod')
plt.plot(x_diff, dy, 'o', label='np.diff aproksimacija')
plt.legend()
plt.title('Uporaba np.diff za odvajanje')
plt.show()
```



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! Poglejmo si prvi odvod (po centralni diferenčni shemi) zapisan z napako metode ($k h^2$) in zaokrožitveno napako ε :

$$y'_i = \frac{1}{2h} ((-y_{i-1} \pm \varepsilon) + (y_{i+1} \pm \varepsilon)) + k h^2 \quad (10.52)$$

V najslabšem primeru se zaokrožitvena napaka sešteje in je skupna napaka:

$$n = \frac{\varepsilon}{h} + k h^2 \quad (10.53)$$

Ko je h velik prevladuje napaka metode $k h^2$; ko pa je h majhen, pa prevladuje zaokrožitvena napaka. Napaka ima minimum, ko velja:

$$n' = -\frac{\varepsilon}{h^2} + 2kh = 0. \quad (10.54)$$

Sledi:

$$h = \sqrt[3]{\frac{\varepsilon}{2k}}. \quad (10.55)$$

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](#)

Določimo sedaj osnovno zaokrožitveno napako za posamezni tip:

```
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:
[np.float16(0.000977), np.float32(1.1920929e-07), np.float64(2.220446049250313e-16)]
```

Kot primer si pogledajmo 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.:

```
(1.+eps16/2).astype('float16')
np.float16(1.0)
```

Definirajmo najprej funkcijo `exp(x)`, ki bo dala rezultat natančnosti, ki jo definira parameter `dtype`:

```
def fun(x, dtype=float): # funkcija
    return np.exp(-dtype(x)).astype(dtype)
```

Definirajmo še funkcijo za analitično določljiv odvod (to bomo pozneje potrebovali za določitev relativne napake):

```
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:

```
fun(1., dtype=np.float16)
np.float16(0.368)
fun(1+eps16/2, dtype=np.float16)
np.float16(0.368)
```

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:

```
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):

```
f1_CDS(fun, x=1., h=.01, dtype=np.float16)
(np.float16( -0.3662), np.float64(0.004535463210798897))
f1_CDS(fun, x=1., h=.01, dtype=np.float64)
```

$$(-0.367885572526119, -1.66667500022579 \cdot 10^{-5}) \quad (10.56)$$

Definirajmo sedaj korak:

```
h=0.25**np.arange(30)
h[:10]
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):

```
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)

/tmp/ipykernel_355276/1408319698.py:2: RuntimeWarning: invalid value encountered in
divide
    f1_ocena = (fun(x+h, dtype=dtype) - fun(x -h, dtype=dtype))/(2*dtype(h))
```

Izrišemo različne tipe v odvisnosti od velikosti koraka h . Najprej uvozimo potrebne knjižnice:

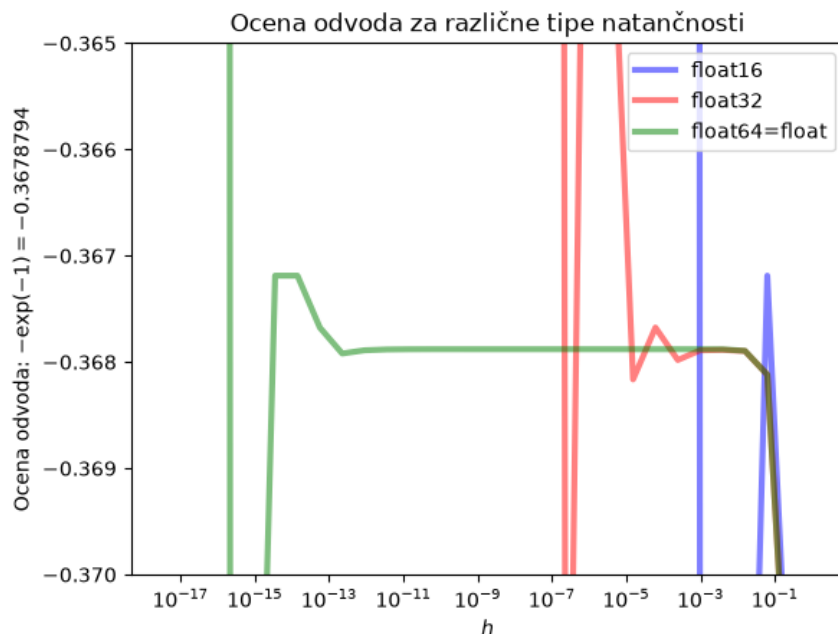
```
import matplotlib.pyplot as plt
%matplotlib inline
```

Definirajmo sliko:

```
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:

```
fig_ocena()
```



Pri relativno velikem koraku h prevladuje napaka metode, pri majhnem koraku pa zaokrožitvena napaka; optimalni korak lahko ocenimo glede na:

$$h = \sqrt[3]{\frac{\varepsilon}{2k}}. \quad (10.57)$$

V konkretnem primeru velja:

$$k = -\frac{f'''(x)}{6} = -\frac{-e^{-x}}{6} = \frac{1}{6e} \quad (x = 1) \quad (10.58)$$

Sledi:

$$h = \sqrt[3]{3\varepsilon e}. \quad (10.59)$$

Izračunamo primeren korak za 16, 32 in 64-bitni zapis:

```
| np.power(3*eps16*np.exp(1),1/3)
```

$$0.199697177510055 \quad (10.60)$$

```
| np.power(3*eps32*np.exp(1),1/3)
```

$$0.00990623467670444 \quad (10.61)$$

```
| np.power(3*eps64*np.exp(1),1/3)
```

$$1.21885484320102 \cdot 10^{-5} \quad (10.62)$$

Najbolje se izkaže 64-bitni zapis, vendar pa tudi pri tem korak manjši od cca $1e-5$ ni priporočen!

10.7 Aproksimacija odvoda s kompleksnim korakom

Aproksimacija odvoda s kompleksnim korakom je uporabna takrat, ko imamo definirano katerokoli analitično funkcijo, vendar pa nimamo na voljo analitičnega izraza za prvi odvod, za podrobnosti glejte (vir). Ideja izhaja iz razvoja funkcije $f(x)$ v Taylorjevo vrsto, vendar se izvede korak v smeri imaginarne osi ih ($i = \sqrt{-1}$):

$$f(x + ih) = f(x) + ih \frac{d}{dx} f(x) - \frac{h^2}{2} \frac{d^2}{dx^2} f(x) + O(h^3) \quad (10.63)$$

Če sedaj predpostavimo, da funkcija $f(x)$ realne vrednosti slika na realno os in da sta x in h realni vrednosti, potem lahko izpeljemo:

$$\mathbf{Im}(f(x + ih)) = h \frac{d}{dx} f(x) + O(h^3) \quad (10.64)$$

$$\mathbf{Re}(f(x + ih)) = f(x) - \frac{h^2}{2} \frac{d^2}{dx^2} f(x) + O(h^3) \rightarrow \mathbf{Re}(f(x + ih)) = f(x) + O(h^2) \quad (10.65)$$

Iz prve enačbe (zgoraj) potem izpeljemo izraz za prvi odvod:

$$f'(x) = \frac{\mathbf{Im}(f(x + ih))}{h} + O(h^2) \quad (10.66)$$

Rezultat je na nek način zelo presenetljiv. Zgoraj smo z eno koračno shemo naprej uspeli pridobiti natančnost $O(h^1)$, tukaj pa $O(h^2)$!

Pomembna prednost metode s kompleksnim korakom je, da **nima težav z odštevanjem skoraj enakih števil** (ang. cancellation error), ki pesti klasične diferenčne metode pri zelo majhnih korakih h . Zato lahko izberemo izjemno majhen h (npr. 10^{-200}) in dobimo praktično točen rezultat (do strojne natančnosti).

Če pogledamo sedaj uporabo metoda na primeru od zgoraj $\exp(-x)$, spomnimo se najprej točnega rezultata pri vrednosti $x = 1$:

```
| f1_točno0
```

$$-0.367879441171442 \quad (10.67)$$

Pri klicu s korakom $h = 0.1$ dobimo točni dve (tri) številki (podobno kakor pri centralni diferenčni shemi), zmanjšanjem koraka na desetino pa bi število točnih števk približno podvojili.

```
| h1_kompleksno = 0.1
| y1_kompleksno = np.exp( -(1.+1.j*h1_kompleksno))
| f1_kompleksno = np.imag(y1_kompleksno)/h1_kompleksno
| f1_kompleksno
```

$$-0.367266615262709 \quad (10.68)$$

Ker nimamo težav z odštevanjem skoraj enakih števil, lahko korak izjemno zmanjšamo in dobimo točen rezulta (do strojne natančnosti):

```
| h1_kompleksno = 1e -10
| y1_kompleksno = np.exp( -(1.+1.j*h1_kompleksno))
| f1_kompleksno = np.imag(y1_kompleksno)/h1_kompleksno
| f1_kompleksno
```

$$-0.367879441171442 \quad (10.69)$$

```

| np.sin(2*x0*1e -5j)
|
| array([0.+0.e+00j, 0.+2.e -05j, 0.+4.e -05j])
|
| np.sin(x0**2+2*x0*1e -5j)
|
| array([ 0.          +0.00000000e+00j,  0.84147098+1.08060461e -05j,
|        -0.7568025  -2.61457448e -05j])

```

Poglejmo si še primer odvoda funkcije $\sin(x^2)$. Iz kode in slike spodaj vidimo bistveno hitrejšo konvergenco metode kompleksnega koraka. Zakaj metoda deluje tako dobro? Če vstavimo $x + ih$ v x^2 , dobimo: $(x + ih)^2 = x^2 + 2ixh - h^2$

Ker je h zelo majhen, je h^2 zanemarljiv v realnem delu, imaginarni del $2xh$ pa se linearno prenese skozi sinus. Na tak način dobimo približek teoretičnega odvoda $2x \cos(x^2)$. Metoda *vidi* odvod v imaginarnem delu brez odštevanja, zato ne pride do napake zaokroževanja.

```

import matplotlib.pyplot as plt
import numpy as np

# Funkcija: sin(x^2)
def f(x):
    return np.sin(x**2)

# Analitični odvod: 2x * cos(x^2)
def df_analytical(x):
    return 2*x * np.cos(x**2)

x0 = 1.5
točno = df_analytical(x0)

h_vals = np.logspace( -10, 0, 100)
err_fd = []
err_cs = []

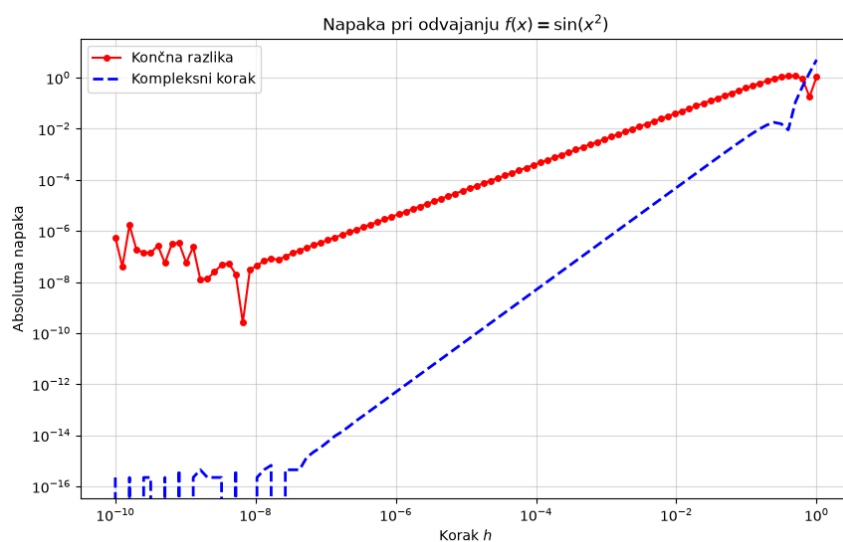
for h in h_vals:
    # Končna razlika (Forward)
    fd_approx = (f(x0 + h) - f(x0)) / h
    err_fd.append(abs(fd_approx - točno))

    # Kompleksni korak
    # f(x + ih) = sin((x+ih)^2) \approx sin(x^2 + 2ixh)
    # \approx sin(x^2) + i * 2xh * cos(x^2)
    # Imag/h nam da točno 2x*cos(x^2)
    cs_approx = f(x0 + 1j*h).imag / h
    err_cs.append(abs(cs_approx - točno))

plt.figure(figsize=(10, 6))
plt.loglog(h_vals, err_fd, 'r -o', label='Končna razlika', markersize=4)
plt.loglog(h_vals, err_cs, 'b - -', label='Kompleksni korak', linewidth=2)

plt.xlabel('Korak $h$')
plt.ylabel('Absolutna napaka')
plt.title(r'Napaka pri odvajanju $f(x)=\sin(x^2)$')
plt.grid(True, which="both", alpha=0.5)
plt.legend()
plt.show()

```



10.8 Dodatno

[Video predavanja na temo numeričnega odvajanja.](#)

Poglavje 11

Numerično integriranje

11.1 Uvod

V okviru tega poglavja bomo za dano funkcijo $f(x)$ izračunali določen integral:

$$\int_a^b f(x) \, dx \quad (11.1)$$

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

$$\int_a^b f(x) \, dx = I + E, \quad (11.2)$$

kjer je E napaka ocene integrala.

Numerični integral bomo računali na podlagi diskretne vsote:

$$I = \sum_{i=0}^m A_i f(x_i), \quad (11.3)$$

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:

$$\int_1^2 x \sin(x) \, dx \quad (11.4)$$

Pripravimo si vozlišča. Osnovni korak naj bo $h = 0.25$, v tem primeru imamo štiri podintervale in pet vozlišč, pri koraku $2h$ so tri vozliščne točke in pri koraku $4h$ samo dve (skrajni):

```
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:

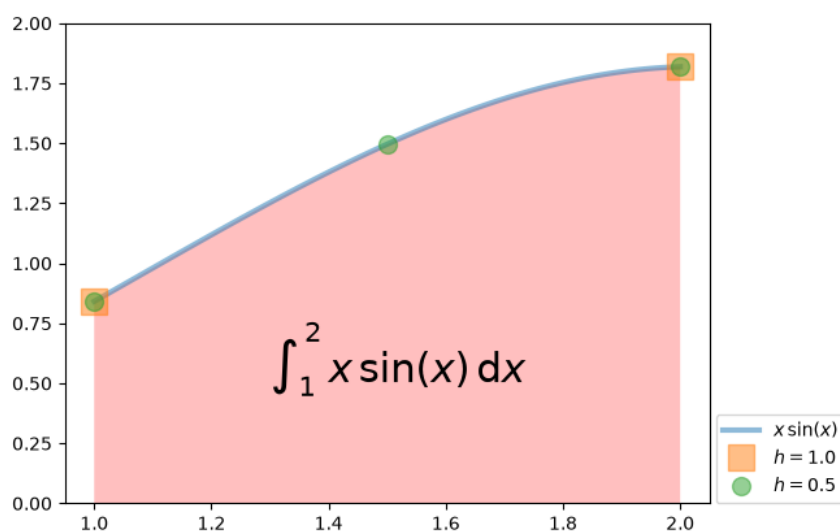
```
yg = xg * np.sin(xg)
y2v = x2v * np.sin(x2v)
y3v = x3v * np.sin(x3v)
y4v = x4v * np.sin(x4v)
y5v = x5v * np.sin(x5v)
```

Pripravimo prikaz podatkov:

```
import matplotlib.pyplot as plt
%matplotlib inline
def fig_motivacija():
    plt.fill_between(xg, yg, alpha=0.25, facecolor='r')
    plt.annotate(r'$\int_1^2 x \sin(x) \, dx$', (1.3, 0.5), fontsize=22)
    plt.plot(xg, yg, lw=3, alpha=0.5, label=r'$x \sin(x)$')
    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:

```
fig_motivacija()
```



Analitično izračunajmo točen rezultat:

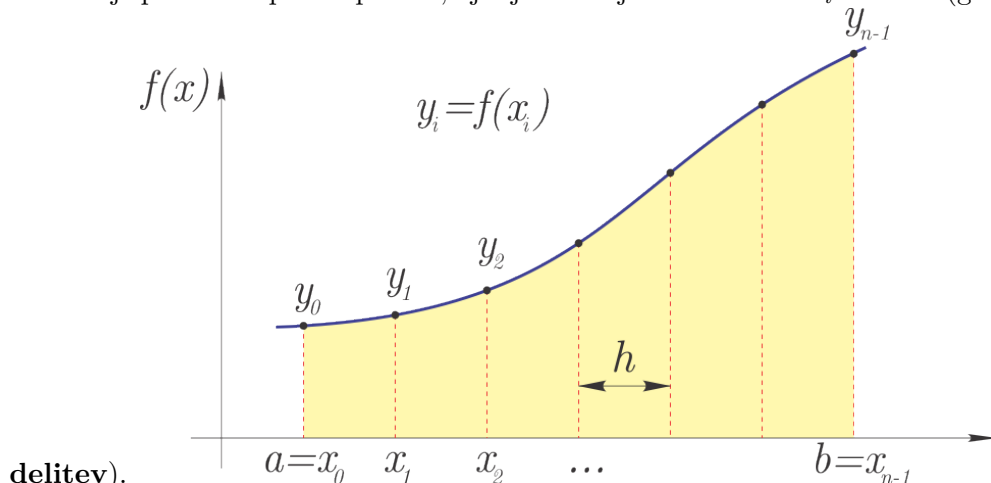
```
import sympy as sym
sym.init_printing()
x = sym.symbols('x')
I_točno = float(sym.integrate(x*sym.sin(x), (x, 1, 2)).evalf())
I_točno
```

1.44042242098021

(11.5)

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 bosta sledili še Simpsonova ter Rombergova metoda.

11.2.1 Trapezno pravilo

Trapezno pravilo vrednosti podintegralne 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:

$$I_{\text{trapezno}} = \sum_{i=0}^1 A_i f(x_i) = \frac{h}{2} \cdot (f(x_0) + f(x_1)). \quad (11.6)$$

In so uteži:

$$A_0 = A_1 = \frac{1}{2} h. \quad (11.7)$$

Numerična implementacija

Numerična implementacija je:

```
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 $h = 1$:

```
I_trapezno = trapezno(y2v, h=h2v)
I_trapezno
```

$$1.33003291922963$$

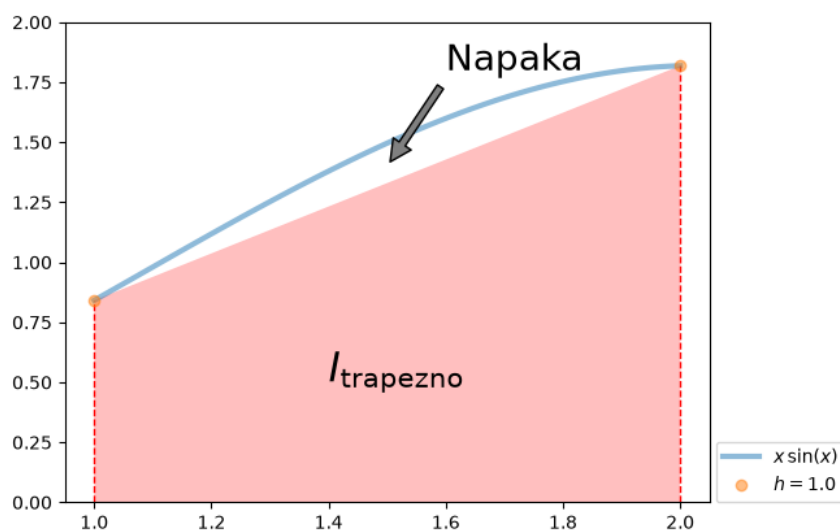
(11.8)

Pripravimo sliko:

```
def fig_trapezno():
    plt.fill_between(x2v, y2v, alpha=0.25, facecolor='r')
    plt.vlines(x2v, 0, y2v, color='r', linestyle='dashed', lw=1)
    plt.annotate('$I_{\mathrm{trapezno}}$', (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:

```
fig_trapezno()
```



Napaka trapeznega pravila

Razlika med analitično vrednostjo integrala in numeričnim približkom I je napaka metode:

$$E = \int_a^b f(x) dx - I, \quad (11.9)$$

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:

$$E_{\text{trapezno}} = -\frac{h^3}{12} f''(\xi), \quad (11.10)$$

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 *sestavljeno 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 $A_i = A_{i+1} = h/2$. Ker so notranja vozlišča podvojena, sledi:

$$A_0 = A_n = \frac{h}{2} \quad \text{in za ostala vozlišča:} \quad A_i = h. \quad (11.11)$$

Pri tem smo predpostavili podintervale enake širine:

$$h = \frac{x_n - x_0}{n} \quad (11.12)$$

Sledi torej:

$$I_{\text{trapezno sest}} = \sum_{i=0}^n A_i f(x_i) = \left(\frac{y_0}{2} + y_1 + y_2 + \cdots + y_{n-1} + \frac{y_n}{2} \right) h. \quad (11.13)$$

Numerična implementacija

Numerična implementacija je:

```
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):

```
x3v
array([1. , 1.5, 2. ])
h3v

0.5
```

(11.14)

Izračunajmo oceno integrala s sestavljenim trapeznim pravilom:

```
I_trapezno_sest = trapezno_sest(y3v, h=h3v)
I_trapezno_sest

1.41313769956786
```

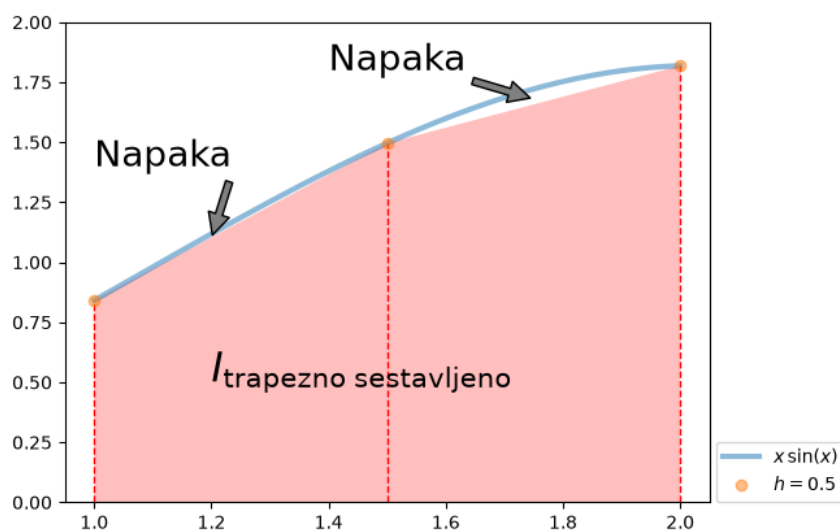
(11.15)

Pripravimo sliko:

```
def fig_trapezno_sest():
    plt.fill_between(x3v, y3v, alpha=0.25, facecolor='r')
    plt.vlines(x3v, 0, y3v, color='r', linestyle='dashed', lw=1)
    plt.annotate('$I_{\mathrm{trapezno}}$ 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:

| 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:

$$E_{\text{trapezno sest}} = -\frac{h^2(b-a)}{12} f''(\eta), \quad (11.16)$$

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 $2h$, velja:

$$\int_a^b f(x) dx = I_h + E_h = I_{2h} + E_{2h}, \quad (11.17)$$

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:

$$E_h = -\frac{h^2(b-a)}{12} f''(\eta) = h^2 K. \quad (11.18)$$

Ob predpostavki, da je $f''(\eta)$ pri koraku h in $2h$ enak (η je pri koraku h in $2h$ dejansko različen), zapišemo:

$$E_{2h} = -\frac{(2h)^2(b-a)}{12} f''(\eta) = 4h^2 K \quad (11.19)$$

Sledi:

$$I_{2h} - I_h = -3K h^2. \quad (11.20)$$

Sedaj lahko ocenimo napako pri koraku h :

$$E_h = h^2 K = \frac{I_h - I_{2h}}{3}. \quad (11.21)$$

Na podlagi ocene napake lahko izračunamo boljši numerični približek I_h^* :

$$I_h^* = I_h + \frac{1}{3} (I_h - I_{2h}) \quad (11.22)$$

ali

$$I_h^* = \frac{4}{3} I_h - \frac{1}{3} I_{2h} \quad (11.23)$$

Numerični zgled

Predhodno smo s trapeznim pravilom že izračunali integral pri koraku $h = 1$ in pri koraku $h = 0,5$, rezultata sta bila:

```
[I_trapezno, I_trapezno_sest]
```

$$[1.33003291922963, 1.41313769956786] \quad (11.24)$$

S pomočjo zgornje formule izračunamo boljši približek:

```
I_trapezno_boljsi = 4/3*I_trapezno_sest - 1/3*I_trapezno
print(f'Točen rezultat: {I_točno}\nBoljši približek: {I_trapezno_boljsi}')
```

```
Točen rezultat: 1.4404224209802097
Boljši približek: 1.4408392930139313
```

Sestavljeno trapezno pravilo je implementirano tudi v paketu `numpy`, s funkcijo `numpy.trapezoid` (v starejših različicah `numpy.trapz`):

```
trapezoid(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](#).

`numpy` implementacija sestavljenega trapeznega pravila

Poglejmo si primer:

```
%%timeit
I_trapezno_np = np.trapezoid(y3v, dx=h3v)
I_trapezno_np
```

$$1.41313769956786 \quad (11.25)$$

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:

$$\int_a^b f(x) dx. \quad (11.26)$$

Tabeliramo podintegralsko funkcijo $f(x)$ in tabelo interpoliramo z Lagrangevim interpolacijskim polinomom $P_n(x)$ stopnje n :

$$P_n(x) = \sum_{i=0}^n f(x_i) l_i(x), \quad (11.27)$$

kjer so $y_i = f(x_i)$ funkcijske vrednosti v vozliščih x_i in je Lagrangev polinom l_i definiran kot:

$$l_i(x) = \prod_{j=0, j \neq i}^n \frac{x - x_j}{x_i - x_j}. \quad (11.28)$$

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 $P_n(x)$:

$$I = \int_{x_0}^{x_n} P_n(x) dx = \int_{x_0}^{x_n} \sum_{i=0}^n f(x_i) l_i(x) dx. \quad (11.29)$$

Ker je integriranje linearna operacija, lahko zamenjamo integriranje in vsoto:

$$I = \sum_{i=0}^n f(x_i) \underbrace{\int_{x_0}^{x_n} l_i(x) dx}_{A_i}. \quad (11.30)$$

Lagrangev polinom integriramo in dobimo uteži A_i :

$$A_i = \int_{x_0}^{x_n} l_i(x) dx \quad (11.31)$$

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 x_0 in x_1 ter korak h :

```
| x, x0, x1, h = sym.symbols('x x0, x1, h')
```

Pripravimo Python funkcijo, ki v simbolni obliki vrne seznam n koeficientov Lagrangevih polinomov $[l_0(x), l_1(x), \dots, l_{n-1}(x)]$ stopnje $n - 1$:

```
| def lagrange(n, x, vozlišča_predpona='x'):
|     if isinstance(vozlišča_predpona, str):
|         vozlišča = sym.symbols(f'{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 pogledjmo Lagrangeva polinoma za linearno interpolacijo ($n = 2$):

```

lag = lagrange(2, x)
lag

```

$$\left[\frac{x - x_1}{x_0 - x_1}, \frac{x - x_0}{-x_0 + x_1} \right] \quad (11.32)$$

Sedaj Lagrangev polinom $l_0(x)$ integriramo čez celotni interval:

```

int0 = sym.integrate(lag[0], (x, x0, x1))
int0

```

$$-\frac{x_0^2}{2x_0 - 2x_1} + \frac{x_0x_1}{x_0 - x_1} + \frac{x_1^2}{2x_0 - 2x_1} - \frac{x_1^2}{x_0 - x_1} \quad (11.33)$$

Izraz poenostavimo in dobimo:

```

int1 = int0.factor()
int1

```

$$-\frac{x_0 - x_1}{2} \quad (11.34)$$

Ker je širina podintervala konstantna je $x_1 = h_0 + h$, izvedemo zamenjavo:

```

zamenjave = {x1: x0+h}
int1.subs(zamenjave)

```

$$\frac{h}{2} \quad (11.35)$$

Zgornje korake za Lagrangev polinom $l_0(x)$ lahko splošimo za seznam Lagrangevih polinomov:

```

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 polinom `li` v seznamu
             lagrange(2,x)
A_trapez

```

$$\left[\frac{h}{2}, \frac{h}{2} \right] \quad (11.36)$$

Izpeljali smo uteži, ki smo jih uporabili pri trapezni metodi:

$$A_0 = h/2 \quad A_1 = h/2. \quad (11.37)$$

Trapezno pravilo je:

$$I_{\text{trapezno}} = \frac{h}{2} (y_0 + y_1) \quad (11.38)$$

Ocena napake je (vir: Burden, Faires, Burden: Numerical Analysis 10th Ed):

$$E_{\text{trapezno}} = -\frac{h^3}{12} f''(\xi). \quad (11.39)$$

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:

```
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
```

$$\left[\frac{h}{3}, \frac{4h}{3}, \frac{h}{3} \right] \quad (11.40)$$

Simpsonovo pravilo (to pravilo se imenuje tudi Simpsonovo 1/3 pravilo) je:

$$I_{\text{Simpsonovo}} = \frac{h}{3} (y_0 + 4y_1 + y_2) \quad (11.41)$$

Ocena napake je (vir: Burden, Faires, Burden: Numerical Analysis 10th Ed):

$$E_{\text{Simpsonovo}} = -\frac{h^5}{90} f^{(4)}(\xi). \quad (11.42)$$

Pri tem je treba izpostaviti, da je napaka lokalno 5. reda $\mathcal{O}(h^5)$ in definirana z neznano vrednostjo četrtega odvoda $f^{(4)}(\xi)$, posledično je to pravilo točno za polinome stopnje 3 ali manj.

Primer uporabe:

```
I_Simps = h3v/3 * np.sum(y3v * [1, 4, 1])
I_Simps
```

$$1.44083929301393 \quad (11.43)$$

Pripravimo sliko. Ker Simpsonovo pravilo temelji na kvadratni interpolaciji, moramo najprej pripraviti interpolacijski polinom (pomagamo si z `numpy.polynomial.Polynomial`):

```
from numpy.polynomial import Polynomial # interpolacijski polinom skozi vozlišča

def fig_Simpsonovo():
    y_interpolate = Polynomial.fit(x3v, y3v, deg=len(x3v) - 1)
    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('$I_{\mathrm{Simpsonovo}}$', (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),
```

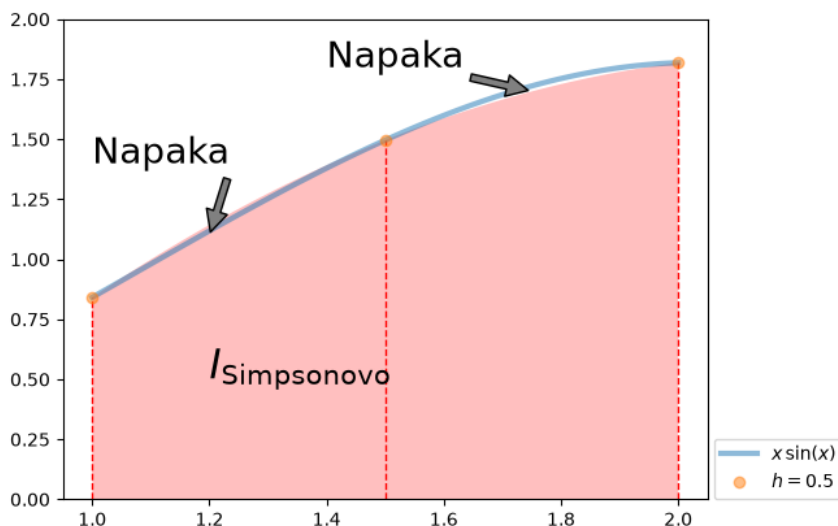
```

        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:

```
fig_Simpsonovo()
```



`scipy.integrate.newton_cotes`

Koeficiente integracijskega pristopa Newton-Cotes pridobimo tudi s pomočjo `scipy.integrate.newton_cotes()` [dokumentacija](#):

```
newton_cotes(rn, equal=0)
```

kjer sta parametra:

- `rn`, ki definira število podintervalov (mogoč je tudi nekonstanten korak, glejte [dokumentacijo](#)),
- `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:

```

from scipy import integrate
integrate.newton_cotes(2)

```

```
(array([0.33333333, 1.33333333, 0.33333333]), -0.011111111111111112)
```

Izračun uteži za Simpsonovo 3/8 pravilo

Nadaljujemo lahko s kubično interpolacijo čez štiri točke ($n = 4$):

```

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

```

$$\left[\frac{3h}{8}, \frac{9h}{8}, \frac{9h}{8}, \frac{3h}{8} \right] \quad (11.44)$$

Simpsonovo 3/8 pravilo je:

$$I_{\text{Simpsonovo } 3/8} = \frac{3h}{8} (y_0 + 3y_1 + 3y_2 + y_3) \quad (11.45)$$

Ocena napake je (vir: Burden, Faires, Burden: Numerical Analysis 10th Ed):

$$E_{\text{Simpsonovo } 3/8} = -\frac{3h^5}{80} f^{(4)}(\xi). \quad (11.46)$$

Poglejmo si primer uporabe. Uporabimo pripravljeno tabelo vrednosti funkcije v štirih točkah:

```
y4v
array([0.84147098, 1.2959172 , 1.65901326, 1.81859485])
I_Simps38 = 3*h4v/8 * np.sum(y4v * [1, 3, 3, 1])
I_Simps38
```

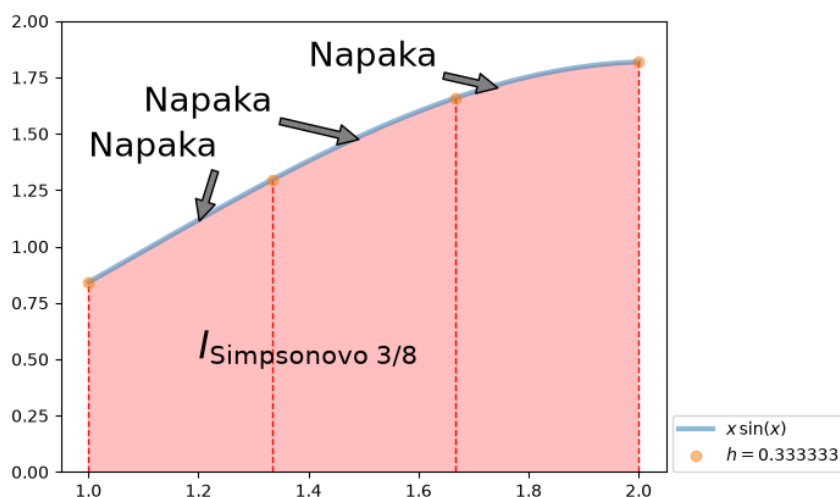
$$1.44060715408392 \quad (11.47)$$

Pripravimo še prikaz:

```
def fig_Simpsonovo38():
    y_interpolate = Polynomial.fit(x4v, y4v, deg=len(x4v) -1)
    plt.fill_between(xg, y_interpolate(xg), alpha=0.25, facecolor='r')
    plt.vlines(x4v, 0, y4v, color='r', linestyle='dashed', lw=1)
    plt.annotate('$I_{\mathrm{Simpsonovo}\;3/8}$', (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:

```
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 $i = 0, 1, \dots, n$. V tem primeru zapišemo sestavljeno Simpsonovo pravilo:

$$\int_a^b f(x) dx = \frac{h}{3} \left(f(x_0) + 4 \sum_{i=1}^{n/2} f(x_{2i-1}) + 2 \sum_{i=1}^{n/2-1} f(x_{2i}) + f(x_n) \right) - \underbrace{\frac{b-a}{180} h^4 f^{(4)}(\eta)}_{E_{\text{Sestavljeno Simpsonovo } 1/3}}, \quad (11.48)$$

kjer je η neznana vrednost na intervalu $[a, b]$. Napaka je četrtega reda $\mathcal{O}(h^4)$.

Numerična implementacija:

```
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])

I_Simps_sest = simpsonovo_sest(y5v, h=h5v)
I_Simps_sest
```

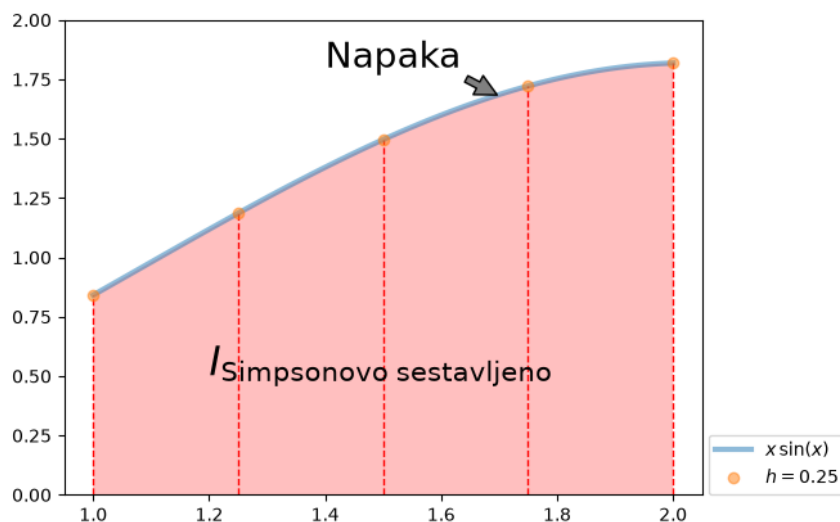
1.44044796026391 (11.49)

Pripravimo sliko:

```
def fig_Simpsonovo_sest():
    y_interpolate = Polynomial.fit(x5v, y5v, deg=len(x5v) - 1)
    plt.fill_between(xg, y_interpolate(xg), alpha=0.25, facecolor='r')
    plt.vlines(x5v, 0, y5v, color='r', linestyle='dashed', lw=1)
    plt.annotate('$I_{\mathrm{Simpsonovo}}$', (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:

```
fig_Simpsonovo_sest()
```



Boljša ocena integrala

Napaka sestavljene Simpsonove metode je definirana z:

$$E_{\text{Sestavljeno Simpsonovo } 1/3} = -\frac{b-a}{180} h^4 f^{(4)}(\eta), \quad (11.50)$$

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 $2h$, velja natančno:

$$\int_a^b f(x) dx = I_h + E_h = I_{2h} + E_{2h}, \quad (11.51)$$

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:

$$E_h = -\frac{b-a}{180} h^4 f^{(4)}(\eta) = h^4 K. \quad (11.52)$$

Ob predpostavki, da je $f^{(4)}(\eta)$ pri koraku h in $2h$ enak (η je pri koraku h in $2h$ dejansko različen), zapišemo:

$$E_{2h} = -\frac{(b-a)}{180} (2h)^4 f^{(4)}(\eta) = 16 h^4 K \quad (11.53)$$

Sledi:

$$I_{2h} - I_h = -15K h^4. \quad (11.54)$$

Sedaj lahko ocenimo napako pri koraku h :

$$E_h = h^4 K = \frac{I_h - I_{2h}}{15}. \quad (11.55)$$

Na podlagi ocene napake lahko izračunamo boljši približek I_h^* :

$$I_h^* = I_h + \frac{1}{15} (I_h - I_{2h}) \quad (11.56)$$

ali

$$I_h^* = \frac{16}{15} I_h - \frac{1}{15} I_{2h} \quad (11.57)$$

Numerični zgled

Predhodno smo s Simpsonovim pravilom že izračunali integral pri koraku $h = 0,5$ in pri koraku $h = 0,25$, rezultata sta bila:

```
| [I_Simps, I_Simps_sest]
|
| [1.44083929301393, 1.44044796026391] (11.58)
```

S pomočjo zgornje formule izračunamo boljši približek:

```
| 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.simpson`

V `scipy` je implementirano sestavljeno Simpsonovo pravilo v `scipy.integrate.simpson()` ([dokumentacija](#)):

```
| simpson(y, x=None, *, dx=1.0, axis= -1)
```

kjer so parametri:

- **y** tabela funkcijskih vrednosti, ki jih integriramo,
- **x** vozlišča, gre za opsijski 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).

V starejših različicah SciPy se je funkcija imenovala `sims` in je imela še parameter `even`; oboje je bilo odstranjeno v SciPy 1.14.

Poglejmo si primer, najprej uvozimo funkcijo `simpson`:

```
| from scipy.integrate import simpson
|
| %%timeit
| simpson(y5v, dx=h5v)
```

$$1.44044796026391 \quad (11.59)$$

11.2.5 Rombergova metoda

Rombergova metoda temelji na Richardsonovi ekstrapolaciji. Predpostavimo, da integral $\int_a^b f(x)dx$ integriramo na intervalu $[a, b]$, ki ga razdelimo na 2^{n-1} podintervalov ($n = 1, 2, 4, 8, \dots$).

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 $n = 1, 2, 4 \dots$ 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 $\mathcal{O}(h_4^2)$,
- ...
- $R_{n,1}$, korak $h_n = h/(2^{n-1})$, red natančnosti $\mathcal{O}(h_n^2)$.

Na podlagi Richardsonove 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 $\mathcal{O}(h_3^4)$,
- ...
- $R_{n,2} = R_{n,1} + \frac{1}{3} (R_{n,1} - R_{n-1,1})$, korak $h_n = h/(2^{n-1})$, red natančnosti $\mathcal{O}(h_n^4)$.

Nato nadaljujemo z Richardsonovo ekstrapolacijo:

- $R_{3,3} = R_{3,2} + \frac{1}{15} (R_{3,2} - R_{2,2})$, korak $h_3 = h/4$, red natančnosti $\mathcal{O}(h_3^6)$,
- ...
- $R_{n,3} = R_{n,2} + \frac{1}{15} (R_{n,2} - R_{n-1,2})$, korak $h_n = h/(2^{n-1})$, red natančnosti $\mathcal{O}(h_n^6)$.

Richardsonovo ekstrapolacijo lahko posplošimo:

- $R_{n,j} = R_{n,j-1} + \frac{1}{4^j - 1} (R_{n,j-1} - R_{n-1,j-1})$, 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:

```
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):

```
## 0(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]
```

$$[1.33003291922963, 1.41313769956786, 1.4336203950899, 1.43872310575291] \quad (11.60)$$

Nato izračunamo boljše približke (dobimo četrti red napake):

```
## 0(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]
```

$$[1.44083929301393, 1.44044796026391, 1.44042400930725] \quad (11.61)$$

Rezultati predstavljajo rezultat Simpsonove metode pri koraku $h = 0,5$, $h = 0,25$ in $h = 0.125$:

```
[simpsonovo_sest(y9v[:,4], 4*h9v), simpsonovo_sest(y9v[:,2], 2*h9v),
simpsonovo_sest(y9v, h9v)]
```

$$[1.44083929301393, 1.44044796026391, 1.44042400930725] \quad (11.62)$$

Ponovno izračunamo boljše približke (dobimo šesti red napake):

```
## 0(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]
```

$$[1.44042187141391, 1.44042241257681] \quad (11.63)$$

Rezultat predstavlja boljši rezultat Simpsonove pri koraku $h = 0,25$ in $h = 0.125$:

```
a4h, a2h, ah = [simpsonovo_sest(y9v[:,4], 4*h9v), simpsonovo_sest(y9v[:,2], 2*h9v),
simpsonovo_sest(y9v, h9v)]
[16/15*a2h - 1/15*a4h, 16/15*ah - 1/15*a2h]
```

$$[1.44042187141391, 1.44042241257681] \quad (11.64)$$

Ponovno izračunamo boljše približke (dobimo osmi red napake):

```
## 0(h8)
R4_4 = R4_3 + 1/63 * (R4_3 - R3_3)
R4_4
```

$$1.4404224211667 \quad (11.65)$$

Rombergova metoda nam torej ponuja visoko natančnost rezultata za relativno majhno numerično ceno. Napako pa ocenimo:

$$E = |R_{n,n} - R_{n-1,n-1}|. \quad (11.66)$$

Rombergova metoda v `scipy.integrate.romb`

V `scipy` je implementirana Rombergova metoda v `scipy.integrate.romb()` ([dokumentacija](#)):

```
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` os integriranja (pomembno v primeru večdimenzijskega numeričnega polja),
- `show` če je `True` prikaže elemente Richardsonove ekstrapolacije.

Poglejmo si primer od zgoraj:

```
from scipy.integrate import romb

y9v

array([0.84147098, 1.01505104, 1.18623077, 1.34872795, 1.49624248,
       1.62261343, 1.72197541, 1.78891084, 1.81859485])

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
=====			

$$1.4404224211667 \quad (11.67)$$

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; pri sodem n zaradi simetrije celo stopnje $n + 1$ (Simpsonovo pravilo, $n = 2$, je npr. točno tudi za kubične polinome).

Gaussov integracijski pristop je v principu drugačen: cilj je integral funkcije $f(x)$ nadomestiti z uteženo vsoto vrednosti funkcije pri diskretnih vrednostih $f(x_i)$:

$$\int_a^b f(x) dx \approx \sum_{i=0}^{n-1} A_i f(x_i). \quad (11.68)$$

Pri tem je neznana utež A_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):

$$f(x) = P_1(x) = a_0 + a_1 x. \quad (11.69)$$

Izračunajmo:

$$\int_{x_L}^{x_D} P_1(x) dx = \left(a_0 x + a_1 \frac{x^2}{2} \right)_{x_L}^{x_D} = -a_0 x_L + a_0 x_D - \frac{a_1 x_L^2}{2} + \frac{a_1 x_D^2}{2}. \quad (11.70)$$

Po drugi strani pa želimo integral izračunati glede na ustrezno uteženo A_0 vrednost funkcije $f(x_0)$ v neznanem vozlišču x_0 (samo eno vozlišče!):

$$\int_{x_L}^{x_D} P_1(x) dx = A_0 P_1(x_0) = A_0 a_0 + A_0 a_1 x_0. \quad (11.71)$$

Z enačenjem zgornjih izrazov izpeljemo:

$$-a_0 x_L + a_0 x_D - \frac{a_1 x_L^2}{2} + \frac{a_1 x_D^2}{2} = A_0 a_0 + A_0 a_1 x_0. \quad (11.72)$$

a_0 in a_1 sta koeficienta linearne funkcije, ki lahko zavzame poljubne vrednosti, zato velja:

$$a_0 (-x_L + x_D - A_0) = 0 \quad \text{in} \quad a_1 \left(-\frac{x_L^2}{2} + \frac{x_D^2}{2} - A_0 x_0 \right) = 0. \quad (11.73)$$

Gre za sistem linearnih enačb z rešitvijo:

$$A_0 = x_D - x_L, \quad x_0 = \frac{x_L + x_D}{2}. \quad (11.74)$$

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 x_L , x_D , pa uvedemo standardne meje.

Standardne meje: $x_L = -1$ in $x_D = 1$

Zaradi splošnosti meje $x \in [x_L, x_D]$ transformiramo v $\xi \in [-1, +1]$ s pomočjo:

$$x = \frac{x_D + x_L}{2} + \frac{x_D - x_L}{2}\xi \quad (11.75)$$

in

$$dx = \frac{x_D - x_L}{2} d\xi. \quad (11.76)$$

Velja:

$$\int_{x_L}^{x_D} f(x) dx = \int_{-1}^1 g(\xi) d\xi, \quad (11.77)$$

kjer je:

$$g(\xi) = \frac{x_D - x_L}{2} f\left(\frac{x_L + x_D}{2} + \frac{x_D - x_L}{2}\xi\right). \quad (11.78)$$

V primeru standardnih mej, je pri eni Gaussovi točki utež $A_0 = 2$ in $x_0 = 0$ vrednost, pri kateri moramo izračunati funkcijo $f(x_0)$.

Strojno izpeljevanje uteži in Gaussove točke

Definirajmo simbole in nastavimo enačbo:

```
a_0, a_1, x, x_L, x_D, A_0, x_0 = sym.symbols('a_0, a_1, x, x_L, x_D, A_0, x_0') #
simboli
P1 = a_0 + a_1*x # linearni polinom
eq = sym.Eq(P1.integrate((x, x_L, x_D)).expand(), A_0*P1.subs(x, x_0)) # teoretični
integral = ocen z utežmi
eq
```

$$a_0 x_D - a_0 x_L + \frac{a_1 x_D^2}{2} - \frac{a_1 x_L^2}{2} = A_0 (a_0 + a_1 x_0) \quad (11.79)$$

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 A_0 in x_0 :

```
| sym.solve([eq.subs(a_0, 0).subs(a_1, 1), eq.subs(a_1, 0).subs(a_0, 1)], [A_0, x_0])
```

$$\left[\left(x_D - x_L, \frac{x_D + x_L}{2} \right) \right] \quad (11.80)$$

Za dodatno razlago priporočam [video posnetek](#).

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:

$$\int_a^b P_{2v-1}(x) dx = \sum_{i=0}^{v-1} A_i P_{2v-1}(x_i). \quad (11.81)$$

Pri izpeljavi se bomo omejili na standardne meje $x_L = -1$, $x_D = 1$, kjer je polinom stopnje $n = 2v - 1$ definiran kot:

$$P_{2v-1}(x) = \sum_{i=0}^{2v-1} a_i x^i. \quad (11.82)$$

Z dvema Gaussovima točkama/vozliščema točno izračunamo integral polinoma do tretjega reda, s tremi Gaussovimi vozlišči pa točno izračunamo integral polinoma do petega reda!

Strojno izpeljevanje

Pripravimo si najprej simbolni zapis polinoma in ustreznih spremenljivk:

```
def P_etc(n=1, a='a', x='x'):
    ai = sym.symbols(f'{a}::{n+1}')
    x = sym.symbols(x)
    return ai, x, sum([ai[i]*x**i for i in range(n+1)])
```

n je stopnja polinoma
seznam a_i
spremenljivka x

Sedaj pa poiščimo uteži A_i in vozlišča x_i za primer dveh Gaussovih vozlišč; polinom je torej tretje stopnje.

```
v = 2 # število vozlišč
ai, x, P = P_etc(n=2*v -1)
xi = sym.symbols(f'x::{v}') # seznam x_i
Ai = sym.symbols(f'A::{v}') # seznam A_i
print(f'Vozlišča: {xi}\nUteži: {Ai}')
```

```
Vozlišča: (x0, x1)
Uteži: (A0, A1)
```

Polinom:

```
P
```

$$a_0 + a_1 x + a_2 x^2 + a_3 x^3 \quad (11.83)$$

Podobno kakor zgoraj za eno vozlišče, tukaj definirajmo enačbe:

```
eqs = [sym.Eq(P.integrate((x, -1, 1)).coeff(a_), \
    sum([Ai[i]*P.subs(x, xi[i]) \
        for i in range(v)]).expand().coeff(a_)) \
    for a_ in ai]
```

eqs

$$\left[2 = A_0 + A_1, 0 = A_0 x_0 + A_1 x_1, \frac{2}{3} = A_0 x_0^2 + A_1 x_1^2, 0 = A_0 x_0^3 + A_1 x_1^3 \right] \quad (11.84)$$

Rešimo jih za neznane x_i in A_i :

```
sol = sym.solve(eqs, sym.flatten((xi, Ai)))
sol
```

$$\left[\left(-\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, 1, 1 \right), \left(\frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3}, 1, 1 \right) \right] \quad (11.85)$$

Določili smo seznam dveh (enakih) rešitev.

Najprej sta definirani vozlišči: $x_0 = -\sqrt{3}/3$ in $x_1 = \sqrt{3}/3$, katerima pripadata uteži $A_0 = A_1 = 1$.

Koda zgoraj je izpeljana v splošnem - število vozlišč v lahko povečate ter izračunate vozlišča ter pripadajoče uteži.

Spodaj je podana tabela vozlišč in uteži za eno, dve in tri vozlišča (za meje $a = -1$, $b = 1$):

Število točk 1:

i	Vozlišče x_i	Utež A_i
0	0	2

Število točk 2:

i	Vozlišče x_i	Utež A_i
0	$-\frac{\sqrt{3}}{3}$	1
1	$+\frac{\sqrt{3}}{3}$	1

Število točk 3:

i	Vozlišče x_i	Utež A_i
0	$-\frac{\sqrt{15}}{5}$	$\frac{5}{9}$
1	0	$\frac{8}{9}$
2	$\frac{\sqrt{15}}{5}$	$\frac{5}{9}$

Za več vozlišč in tudi oceno napake, glejte [Mathworld Legendre-Gauss Quadrature](#).

Za primer treh Gaussovih točk numerični integral izračunamo (standardne meje):

$$I_{\text{Gauss3}} = \frac{5}{9} f\left(-\frac{\sqrt{15}}{5}\right) + \frac{8}{9} f(0) + \frac{5}{9} f\left(\frac{\sqrt{15}}{5}\right). \quad (11.86)$$

Numerična implementacija

Numerična implementacija (vključno s transformacijo mej) za eno, dve ali tri vozlišča:

```
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 vozlišč: 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:

```
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:

```
Gaussova(fun=f, a=1., b=2., vozlišč=1)
```

1.49624247990608 (11.87)

```
Gaussova(fun=f, a=1., b=2., vozlišč=2)
```

1.44014401845179 (11.88)

```
Gaussova(fun=f, a=1., b=2., vozlišč=3)
```

1.44042294912151 (11.89)

`scipy.integrate.fixed_quad`

Znotraj `scipy` je Gaussova integracijska metoda implementirana v okviru funkcije `scipy.integrate.fixed_quad()` ([dokumentacija](#)):

```
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:

```
from scipy.integrate import fixed_quad
fixed_quad(f, a=1, b=2, n=3)[0]
```

1.44042294912151 (11.90)

Rezultat je enak predhodnemu:

```
Gaussova(fun=f, a=1., b=2., vozlišč=3)
```

1.44042294912151 (11.91)

11.4 `scipy.integrate`

`scipy.integrate` je močno orodje za numerično integriranje (glejte [dokumentacijo](#)). V nadaljevanju si bomo pogledali nekatere funkcije.

11.4.1 Integracijske 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( ...)`,

`scipy.integrate.quad`

Poglejmo si zelo uporabno funkcijo za integriranje, `scipy.integrate.quad()` ([dokumentacija](#)):

```
quad(func, a, b, args=(), full_output=0, epsabs=1.49e -08, epsrel=1.49e -08,
    limit=50,
    points=None, weight=None, wvar=None, woargs=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:

```
from scipy import integrate

integrate.quad(f, a=1, b=2)
```

$$(1.44042242098021, 1.59919013695854 \cdot 10^{-14}) \quad (11.92)$$

`scipy.integrate.dblquad`

Gre za podobno funkcijo kot `quad`, vendar za integriranje po dveh spremenljivkah ([dokumentacija](#)). 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)
```

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:

$$\int_{-1}^1 \left(\int_0^{\sqrt{1-x^2}} 1 \, dy \right) dx = \frac{\pi}{2} \quad (11.93)$$

Definirajmo ustrezne Python funkcije in izračunajmo rezultat:

```
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)
```

$$(1.5707963267949, 1.00023545002159 \cdot 10^{-9}) \quad (11.94)$$

11.4.2 Integracijske funkcije, ki zahtevajo tabelo vrednosti:

- `trapezoid(y[, x, dx, axis])` sestavljeno trapezno pravilo,
- `cumulative_trapezoid(y[, x, dx, axis, initial])` kumulativni integral podintegral-ske funkcije (vrne rezultat v vsakem vozlišču),
- `simpson(y[, x, dx, axis])` Simpsonova metoda,
- `romb(y[, dx, axis, show])` Rombergova metoda.

Tukaj si bomo na primeru ogledali funkcijo `cumulative_trapezoid`. Oglejmo si primer, 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:

```
def pospešek(t):
    F = np.sin(t)
    m = 1
    return F/m
```

Zanima nas dogajanje v času 3 sekund:

```
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):

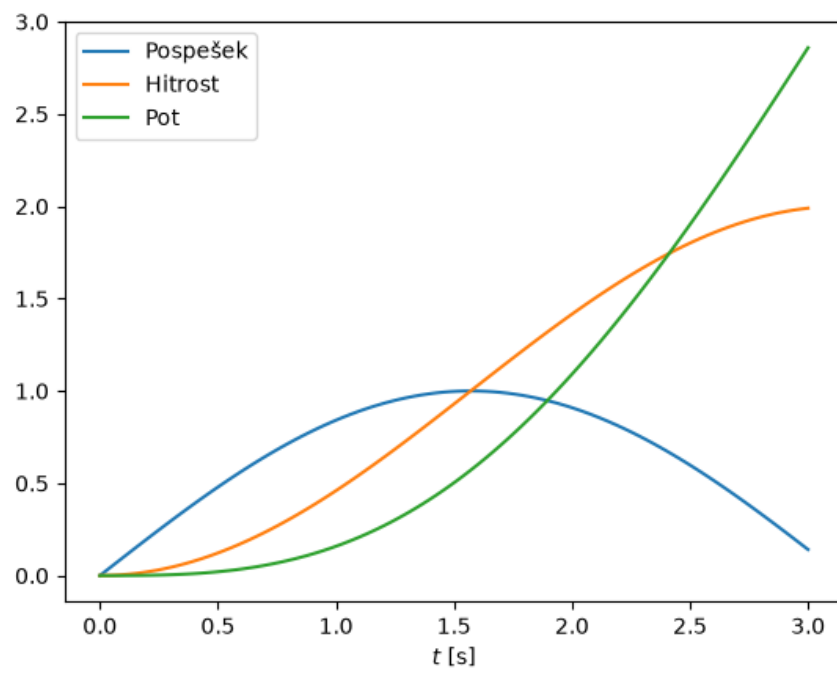
```
a = pospešek(t)
v = integrate.cumulative_trapezoid(y=a, dx=h, initial=0)
```

Hitrost sedaj še enkrat integrirajmo, da izračunamo pot:

```
s = integrate.cumulative_trapezoid(y=v, dx=h, initial=0)
```

Prikažimo rezultat:

```
plt.plot(t, a, label='Pospešek')
plt.plot(t, v, label='Hitrost')
plt.plot(t, s, label='Pot')
plt.xlabel('$t$ [s]')
plt.legend();
```



Poglavje 12

Numerično reševanje diferencialnih enačb - začetni problem

```
import numpy as np
from ipywidgets import interact
import matplotlib.pyplot as plt
%matplotlib inline
import sympy as sym
sym.init_printing()
```

12.1 Uvod

12.1.1 Zapis (ene) diferencialne enačbe

Predpostavimo, da je mogoče diferencialno enačbo prvega reda zapisati v *eksplicitni* obliki:

$$y' = f(t, y), \quad (12.1)$$

kjer je $f(t, y)$ podana funkcija in velja $y' = dy/dt$.

Dodatno je podan začetni pogoj:

$$y(t_0) = y_0. \quad (12.2)$$

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 intervalu, ki vsebuje t_0 .

Pri numeričnem reševanju vedno računamo tabelo funkcije $y(t_i)$, ki reši dan začetni problem. Pri tem so vozlišča t_i običajno ekvidistantna:

$$t_0, t_0 + h, t_0 + 2h, \dots \quad (12.3)$$

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:

$$y(t + h) = y(t) + f(t, y(t))h + \mathcal{O}(h^2). \quad (12.4)$$

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 $[t_0, t_n]$ in velja $h = (t_n - t_0)/n$. n je število integracijskih korakov (kolikokrat izvedemo korak 2 v zgornjem algoritmu).

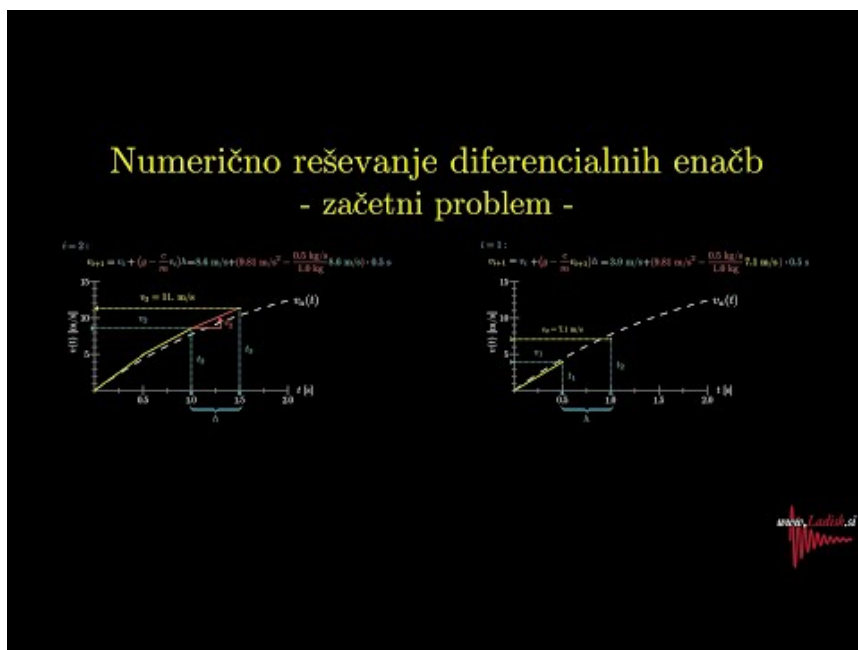
Numerična rešitev začetnega problema:

$$y_0, y_1, y_2, \dots, y_n \quad (12.5)$$

pri vrednostih neodvisne spremenljivke:

$$t_0, t_1, t_2 \dots t_n. \quad (12.6)$$

```
from IPython.display import YouTubeVideo
YouTubeVideo('cfZ8v0b -R8o', width=800, height=300)
```



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:

$$y(t_n) = y_{n,h} + E_h, \quad (12.7)$$

kjer je $y_{n,h}$ numerični približek in E_h napaka metode. Ker je globalna napaka prvega reda, lahko napako zapišemo kot:

$$E_h = k h. \quad (12.8)$$

Podobno lahko za velikost koraka $2h$ zapišemo:

$$y(t_n) = y_{n,2h} + E_{2h}, \quad (12.9)$$

kjer je $y_{n,2h}$ numerični približek in E_{2h} napaka metode:

$$E_{2h} = k \, 2h. \quad (12.10)$$

Ob predpostavki, da je konstanta k pri koraku h in koraku $2h$ enaka, lahko določimo oceno napake pri boljšem približku E_h . Očitno velja:

$$y_{n,h} + k h = y_{n,2h} + 2 k h \quad (12.11)$$

nato določimo *oceno napake*:

$$E_h = k h = y_{n,h} - y_{n,2h}. \quad (12.12)$$

12.2.2 Komentar na implicitno Eulerjevo metodo

Pri eksplisitni 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*):

$$y_{i+1} = y_i + f(t_{i+1}, y_{i+1}) h. \quad (12.13)$$

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 eksplisitna oblika, vendar pa je numerično bolj zahtevna (zaradi računanja rešitve enačbe).

12.2.3 Numerična implementacija

Najprej uvozimo potrebne knjižnice:

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

Nato definirajmo Eulerjevo metodo:

```
def euler(f, t, y0, *args, **kwargs):
    """
    Eulerjeva metoda za reševanje sistema diferencialnih enačb: y' = f(t, y)

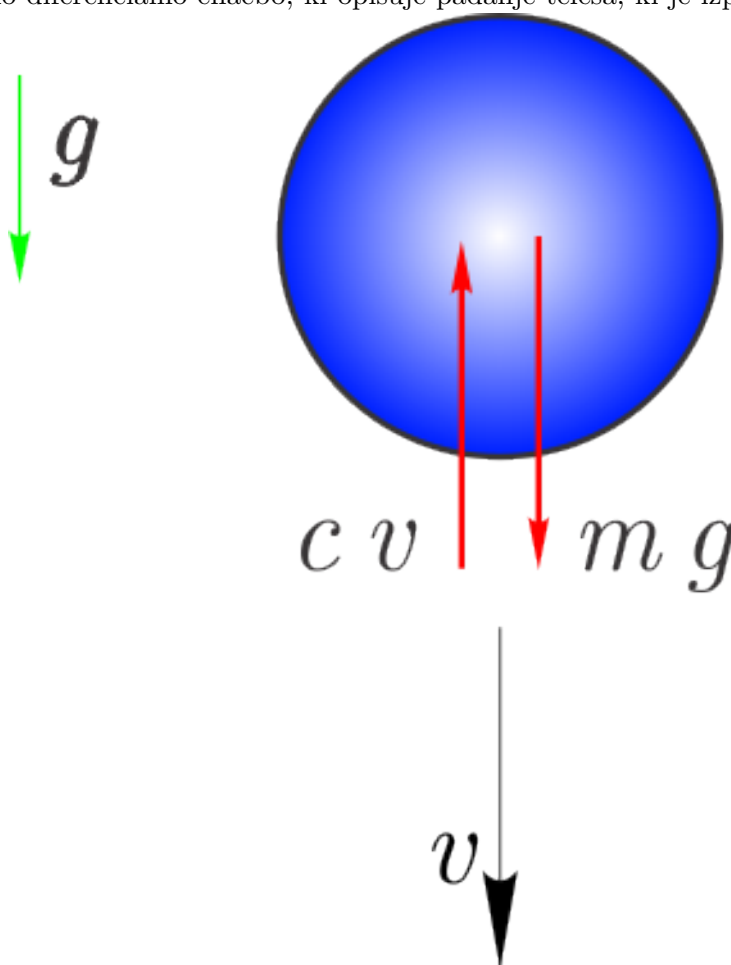
    :param f: funkcija, ki vrne prvi odvod - f(t, y)
    :param t: časovni vektor kjer računamo rešitev
    :param y0: začetna vrednosti
    :param args: dodatni argumenti funkcije f (brezimenski)
    :param kwargs: dodatni argumenti funkcije f (poimenovani)
    :return y: vrne np.array ``y`` vrednosti funkcije.
    """
    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 bolj oceno napake vključiti v funkcijo `euler`, vendar jo zaradi jasnosti predstavimo ločeno):

```
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 upor:

Glede na II. Newtonov zakon, lahko zapišemo diferencialno enačbo:

$$m g - c v = m v', \quad (12.14)$$

kjer je m masa, g gravitacijski pospešek, c koeficient zračnega upora in v hitrost. Diferencialno enačbo bi hoteli rešiti glede na začetni pogoj:

$$v(0) = 0 \text{ m/s}. \quad (12.15)$$

Funkcija desne strani / prvega odvoda $f(t, y)$ je:

$$f(t, y) = g - c \frac{y}{m} \quad (12.16)$$

in začetni pogoj:

$$y_0 = 0. \quad (12.17)$$

Definirajmo funkcijo desnih strani:

```
def f_zračni_upor(t, y, g=9.81, m=1., c=0.5):
    return g - c*y/m
```

Definirajmo začetni pogoj in časovni vektor, kjer nas zanima rezultat:

```
v0 = 0
t = np.linspace(0, 10, 11)
t
```

```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

Kličemo funkcijo `euler` za izračun vrednosti y (hitrost v):

```
y = euler(f_zračni_upor, t, y0=v0)
y
```

```
array([ 0.          ,  9.81          , 14.715         , 17.1675        , 18.39375       ,
        19.006875    , 19.3134375    , 19.46671875    , 19.54335938    , 19.58167969    ,
        19.60083984])
```

Prikažemo rezultat:

```
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()
```

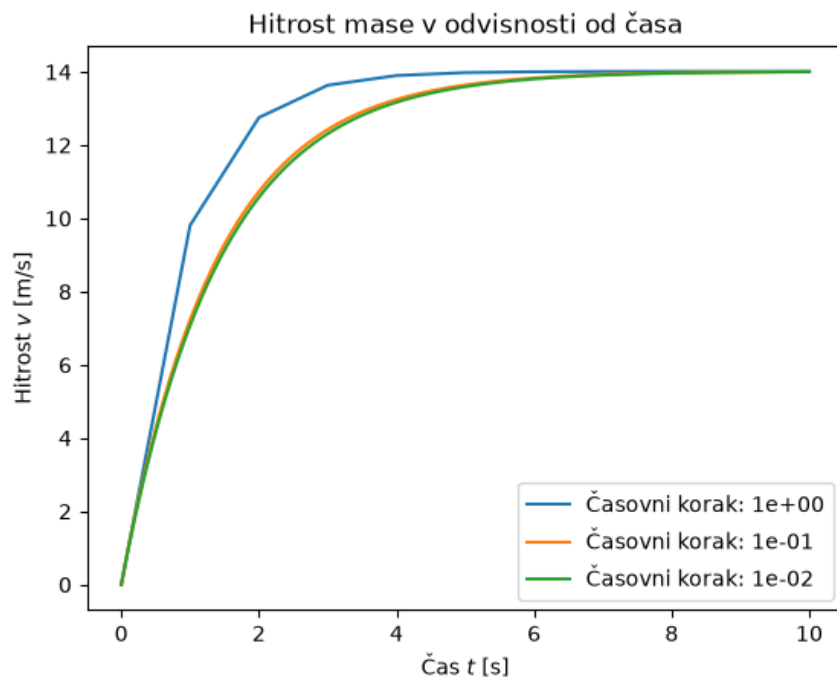


Preverimo sedaj vpliv časovnega koraka:

```

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:

```

n=101
t = np.linspace(0, 10, n)
euler_napaka(f_zračni_upor, t, y0=v0)

```

$$-0.015043713735345 \quad (12.18)$$

```

n=1001
t = np.linspace(0, 10, n)
euler_napaka(f_zračni_upor, t, y0=v0)

```

$$-0.00163798622449463 \quad (12.19)$$

Ko smo korak zmanjšali na desetino, se je proporcionalno zmanjšala tudi napaka (prvi red napake).

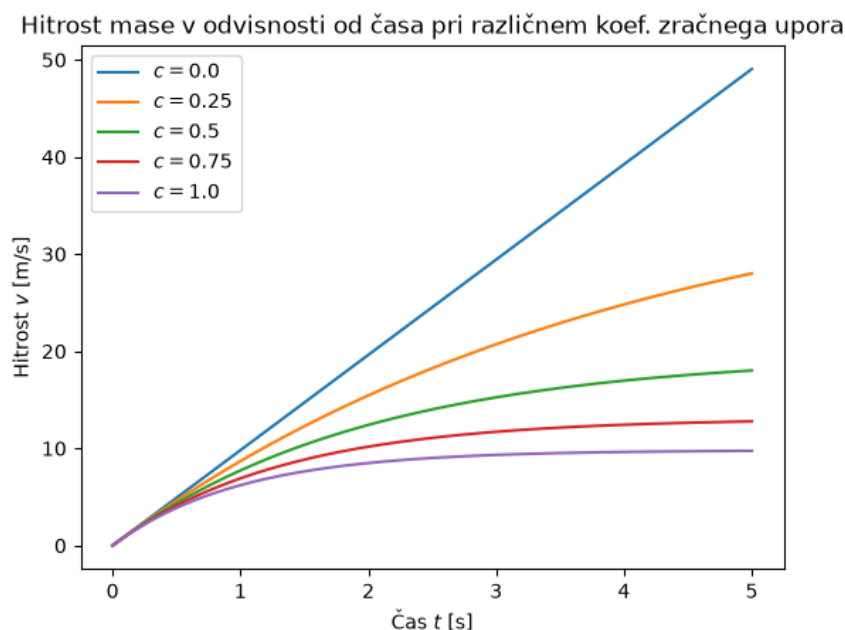
Poglejmo še primer, ko je zračni upor c argument funkcije `euler` in je prek `**kwargs` posredovan v funkcijo `f_zračni_upor()`:

```

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:

$$y(t+h) = y(t) + y'(t)h + \frac{1}{2}y''(t)h^2 + \mathcal{O}(h^3). \quad (12.20)$$

Lokalna napaka metode bo tako tretjega reda, globalna pa drugega reda.

Uporabimo zamenjavi $y'(t) = f(t, y)$ in $y''(t) = f'(t, y)$:

$$y(t+h) = y(t) + f(t, y)h + \frac{1}{2}f'(t, y)h^2 + \mathcal{O}(h^3). \quad (12.21)$$

Ker je desna stran $f(t, y)$ odvisna od neodvisne t in odvisne spremenljivke y , moramo uporabiti implicitno odvajanje:

$$f'(t, y) = \frac{\partial f(t, y)}{\partial t} + \frac{\partial f(t, y)}{\partial y} \underbrace{\frac{dy}{dt}}_{y'=f(t, y)} = \frac{\partial f(t, y)}{\partial t} + \frac{\partial f(t, y)}{\partial y} f(t, y). \quad (12.22)$$

Vstavimo v izraz za Taylorjevo vrsto:

$$y(t+h)_{\text{Taylor}} = y(t+h) = y(t) + f(t, y)h + \frac{1}{2} \left(\frac{\partial f(t, y)}{\partial t} + \frac{\partial f(t, y)}{\partial y} f(t, y) \right) h^2. \quad (12.23)$$

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 z uteženo dodatno vrednostjo funkcije f :

$$y(t+h)_{\text{Runge-Kutta}} = y(t) + c_0 f(t, y) h + c_1 \underbrace{f(t+ph, y+qh f(t, y))}_A h. \quad (12.24)$$

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:

$$f(t+ph, y+qh f(t, y)) = \underbrace{f(t, y) + \frac{\partial f(t, y)}{\partial t} (ph) + \frac{\partial f(t, y)}{\partial y} (qh f(t, y))}_B. \quad (12.25)$$

Vstavimo sedaj izpeljani B nazaj v izraz za $y(t+h)_{\text{Runge-Kutta}}$:

$$y(t+h)_{\text{Runge-Kutta}} = y(t) + c_0 f(t, y) h + c_1 \left(f(t, y) + \frac{\partial f(t, y)}{\partial t} (ph) + \frac{\partial f(t, y)}{\partial y} (qh f(t, y)) \right) h. \quad (12.26)$$

Nadaljujemo z izpeljevanjem in enačbo preoblikujemo, da bo podobna zgoraj izpeljani s Taylorjevo vrsto $y(t+h)_{\text{Taylor}}$:

$$y(t+h)_{\text{Runge-Kutta}} = y(t) + (c_0 + c_1) f(t, y) h + \frac{1}{2} \left(\frac{\partial f(t, y)}{\partial t} 2c_1 p + 2c_1 q \frac{\partial f(t, y)}{\partial y} f(t, y) \right) h^2. \quad (12.27)$$

Primerjajmo sedaj z zgoraj izpeljanim izrazom:

$$y(t+h)_{\text{Taylor}} = y(t) + f(t, y) h + \frac{1}{2} \left(\frac{\partial f(t, y)}{\partial t} + \frac{\partial f(t, y)}{\partial y} f(t, y) \right) h^2. \quad (12.28)$$

Ugotovimo, da za enakost mora veljati:

$$c_0 + c_1 = 1, \quad 2c_1 p = 1, \quad 2c_1 q = 1. \quad (12.29)$$

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 na rešitev. Sicer pa velja omeniti, da tudi 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:

$$y_{i+1} = y_i + f \left(t_i + \frac{1}{2} h, y_i + \frac{1}{2} h f(t_i, y_i) \right) h. \quad (12.30)$$

12.4 Metoda Runge-Kutta četrtega reda

Podobno kot smo izpeljali metodo Runge-Kutta drugega reda, se izpelje metoda Runge-Kutta četrtega reda. Tudi pri metodi četrtega reda obstaja več različic in kot metoda Runge-Kutta četrtega reda razumemo naslednjo metodo:

$$y_{i+1} = y_i + \frac{1}{6}(k_0 + 2k_1 + 2k_2 + k_3), \quad (12.31)$$

kjer so:

$$\begin{aligned} k_0 &= h f(t_i, y_i) \\ k_1 &= h f\left(t_i + \frac{h}{2}, y_i + \frac{k_0}{2}\right) \\ k_2 &= h f\left(t_i + \frac{h}{2}, y_i + \frac{k_1}{2}\right) \\ k_3 &= h f(t_i + h, y_i + k_2). \end{aligned} \quad (12.32)$$

Koraki metode Runge-Kutta četrtega reda so:

1. Določitev $i = 0$ in $t_0, y_0 = y(t_0)$,
2. Izračun koeficientov: k_0, k_1, k_2, k_3 ,
3. Izračun vrednosti rešitve diferencialne enačbe pri $t_{i+1} = t_i + h$: $y_{i+1} = y_i + \frac{1}{6}(k_0 + 2k_1 + 2k_2 + k_3)$,
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:

$$y(t_n) = y_{n,h} + E_h, \quad (12.33)$$

kjer je $y_{n,h}$ numerični približek rešitve in E_h napaka metode. Ker je globalna napaka četrtega reda, lahko napako zapišemo tako:

$$E_h = k h^4. \quad (12.34)$$

Podobno lahko za velikost koraka $2h$ zapišemo:

$$y(t_n) = y_{n,2h} + E_{2h}, \quad (12.35)$$

kjer je $y_{n,2h}$ numerični približek rešitve in E_{2h} napaka metode:

$$E_{2h} = k (2h)^4 = 16 k h^4. \quad (12.36)$$

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:

$$y_{n,h} + k h^4 = y_{n,2h} + 16 k h^4, \quad (12.37)$$

sledi:

$$15 k h^4 = y_{n,h} - y_{n,2h} \quad (12.38)$$

in nato določimo oceno napake natančnejše rešitve:

$$E_h = \frac{y_{n,h} - y_{n,2h}}{15}. \quad (12.39)$$

12.4.2 Numerična implementacija

```
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:

```
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:

```
def f_zračni_upor(t, y, g=9.81, m=1., c=0.5):
    return g - c*y/m
```

Podajmo začetni pogoj in časovni vektor, kjer nas zanima rezultat:

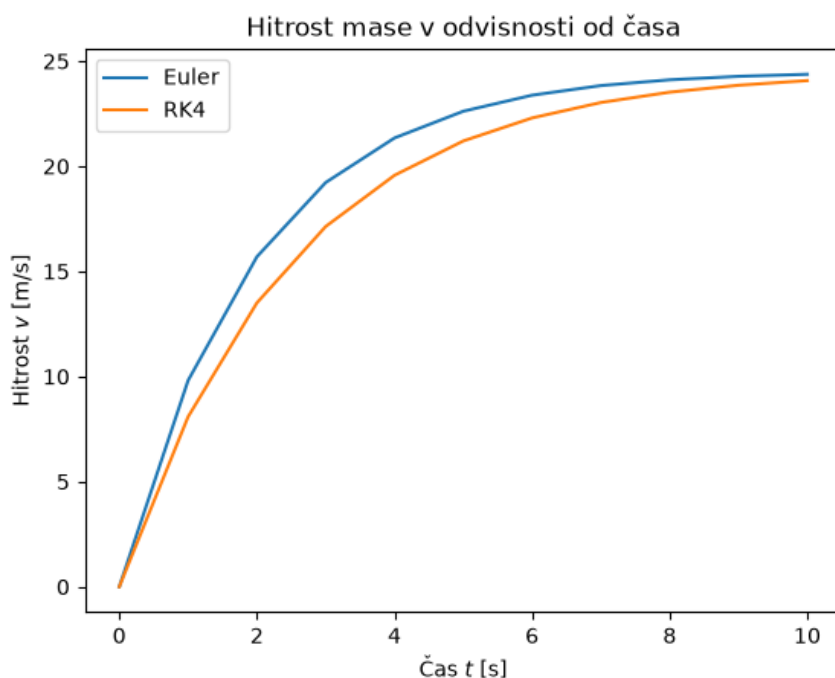
```
v0 = 0
t = np.linspace(0, 10, 11)
t
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.] )
```

Za primerjavo izračunajmo rešitev s funkcijo `euler` ter `runge_kutta_4`:

```
y_euler = euler(f_zračni_upor, t, y0=v0, c=0.4)
y_rk4 = runge_kutta_4(f_zračni_upor, t, y0=v0, c=0.4)
```

Prikažemo rezultat:

```
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:

```
n=101
t = np.linspace(0, 10, n)
runge_kutta_4_napaka(f_zračni_upor, t, y0=v0, c=0.4)
```

$$4.10657750175384 \cdot 10^{-8} \quad (12.40)$$

```
n=1001
t = np.linspace(0, 10, n)
runge_kutta_4_napaka(f_zračni_upor, t, y0=v0, c=0.4)
```

$$3.85990498822745 \cdot 10^{-12} \quad (12.41)$$

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. Tukaj si bomo ogledali funkcijo `scipy.integrate.solve_ivp`, ki je primerna za večino začetnih problemov.

12.5.1 `scipy.integrate.solve_ivp`

Sintaksa za uporabo (IVP - *angl.* Initial Value Problem):

```
scipy.integrate.solve_ivp(fun, t_span, y0, method='RK45',
                          t_eval=None, dense_output=False,
                          events=None, vectorized=False,
                          args=None, **options)
```

Pojasnilo vseh argumentov je v [dokumentaciji](#), tukaj bomo izpostavili nekatere:

- `fun` je desna stran (`func(t, y...)`),
- `t_span` terka (`t0, tf`), ki definira začetni `t0` in končni čas `tf`,
- `y0 seznam` začetne(ih) vrednosti,
- `method` definira numerično metodo (privzeta je eksplcitna 'RK45', za toge sisteme pa so boljše: 'Radau', 'BDF', 'LSODA'),
- `t_eval` uporabimo, če želimo rešitve ob določenih vrednosti neodvisne spremenljivke,
- `dense_output` ali se pripravi tudi zvezna rešitev, privzeto `False`,
- `events` za sledenje dogodkov, ki ustavijo reševanje začetnega problema (npr. ko se masa dotakne tal in je relativna razdalja do tal nič).
- `args` za posredovanje argumentov v funkcijo `fun` (glejte primer spodaj).

Rezultat klicanja `solve_ivp` je objekt z atributi (izbrani):

- `t` vrednosti neodvisne spremenljivke pri katerih je izračunan rezultat,
- `y` rezultat,
- `sol` funkcija zvezne rešitve (samo v primeru `dense_output=True`)
- `success` je `True`, če je bila rešitev uspešno izračunana ali prekinjena zaradi dogodka (`events`).

Numerični zgled

Definirajmo začetni pogoj in časovni vektor, kjer nas zanima rezultat (prikažemo samo prvih deset elementov):

```
v0 = 0
t = np.linspace(0, 10, 101)
t[:10]

array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

Argumente (`g=9.81`, `m=1.`, `c=0.4`) v funkcijo posredujemo prek parametra `args`:

```
from scipy.integrate import solve_ivp
y_ivp = solve_ivp(f_zračni_upor, t_span=(t[0], t[-1]), y0=[v0], args=(9.81, 1, 0.4))
```

Lahko bi pa uporabili tudi `lambda` izraza ([dokumentacija](#)):

```
y_ivp = solve_ivp(lambda t,y: f_zračni_upor(t, y, c=0.4), t_span=(t[0], t[-1]),
y0=[v0])
```

Zgoraj uporabljen `lambda` izraz `lambda t, y: f_zračni_upor(t, y, c=0.4)` je ekvivalenten:

```
def ime_funkcije(t, y):
    return f_zračni_upor(t, y, c=0.4)
```

Poglejmo rezultat:

```
y_ivp
```

```

message: The solver successfully reached the end of the integration interval.
success: True
status: 0
  t: [ 0.000e+00  1.000e -04  1.100e -03  1.110e -02  1.111e -01
      1.111e+00  2.982e+00  5.236e+00  7.948e+00  1.000e+01]
  y: [[ 0.000e+00  9.810e -04  1.079e -02  1.086e -01  1.066e+00
      8.800e+00  1.708e+01  2.150e+01  2.350e+01  2.407e+01]]
sol: None
t_events: None
y_events: None
nfev: 56
njev: 0
nlu: 0

```

Ker je `solve_ivp` že pripravljen za sistem navadnih diferencialnih enačb prvega reda, je rezultat podan v dveh dimenzijah. Iskan rezultat je torej: `y_ivp.y[0]`.

Za primerjavo še rezultat z lastno implementacijo Runge-Kutta 4. reda:

```

y_rk4 = runge_kutta_4(f_zračni_upor, t, y0=v0, c=0.4)

y_rk4[:10]

array([0.          , 0.96163898, 1.88557156, 2.77327623, 3.6261735 ,
      4.44562819, 5.23295161, 5.98940363, 6.71619474, 7.41448797])

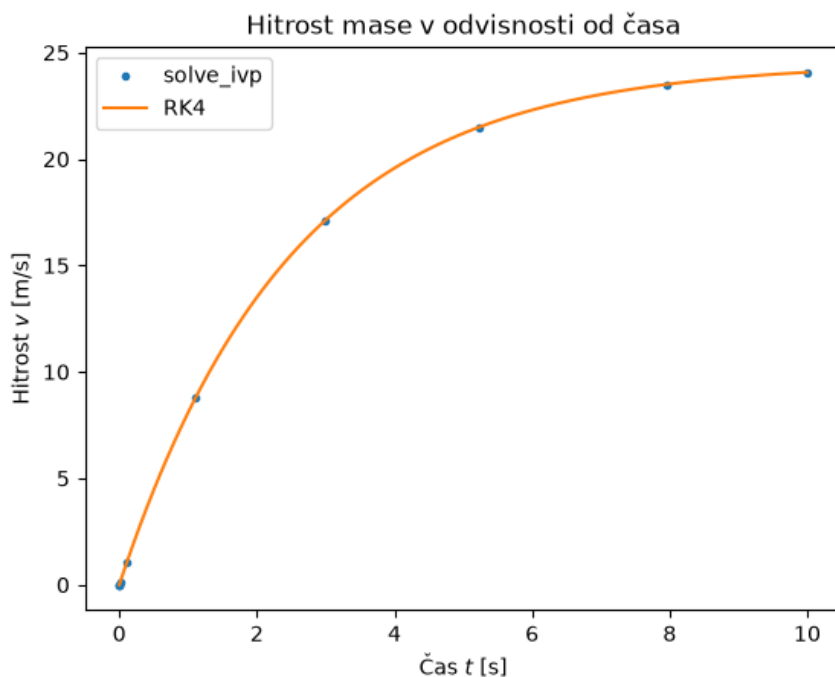
```

Prikažemo rezultat:

```

plt.plot(y_ivp.t, y_ivp.y[0], '.', label='solve_ivp')
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()

```



12.6 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:

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{y}), \quad (12.42)$$

kjer so podani dodatni (začetni) pogoji:

$$\mathbf{y}(t_0) = \mathbf{y}_0. \quad (12.43)$$

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 $a = t_0, t_1, \dots, t_{n-1} = b$, to je m funkcijskih vrednosti na vsakem koraku $y_k(t_i)$, $k = 0, 1, \dots, m-1$ in $i = 0, 1, \dots, n-1$. 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 potrebujeata zgolj malenkostne popravke (y_0 je numerično polje, $\mathbf{f}$ vrne seznam vrednosti odvodov):

```
def euler_sistem(f, t, y0, *args, **kwargs):
    """
    Eulerjeva metoda za reševanje sistema navadnih diferencialnih enačb prvega reda :
    y' = f(t, y)

    :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

def runge_kutta_4_sistem(f, t, y0, *args, **kwargs):
    """
    Metoda Runge -Kutta 4. reda za reševanje sistema navadnih diferencialnih enačb
    prvega reda: y' = f(t, y)

    :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(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((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

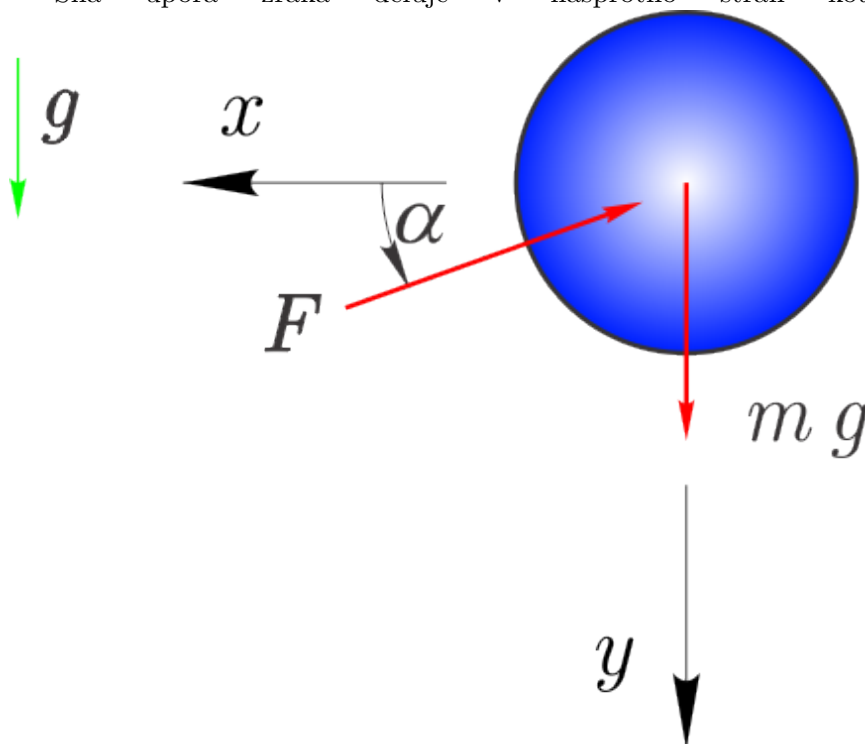
```

Numerični zgled

Padanje mase nadgradimo v ravninsko gibanje; velikost sile upora zraka naj bo definirana kot:

$$|\mathbf{F}(\mathbf{v})| = c |\mathbf{v}|^2. \quad (12.44)$$

Sila upora zraka deluje v nasprotno stran kot kaže vektor hitrosti.



Ob pomoči slike, definiramo silo v x smeri:

$$F_x = -c (v_x^2 + v_y^2) \cos(\alpha) = -c (v_x^2 + v_y^2) \frac{v_x}{\sqrt{v_x^2 + v_y^2}} = -c v_x \sqrt{v_x^2 + v_y^2}. \quad (12.45)$$

Podobno je sila v y smeri:

$$F_y = -c v_y \sqrt{v_x^2 + v_y^2}. \quad (12.46)$$

Glede na drugi Newtonov zakon zapišemo sistem dveh (vezanih) diferencialnih enačb prvega reda:

$$F_x = m v'_x, \quad (12.47)$$

$$m g + F_y = m v'_y, \quad (12.48)$$

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:

$$v_x(0) = v_y(0) = 5 \text{ m/s}. \quad (12.49)$$

Definirajmo seznam desnih strani:

```
def f_zračni_upor_sila(t, y, 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])
```

Definirajmo začetni pogoj in časovni vektor, kjer nas zanima rezultat:

```
y0 = np.array([5., 5.])
t = np.linspace(0, 5, 101)
t[:5]

array([0. , 0.05, 0.1 , 0.15, 0.2 ])
```

Za primerjavo izračunajmo rešitev s funkcijo `runge_kutta_4` ter `solve_ivp`:

```
y_RK4 = runge_kutta_4_sistem(f_zračni_upor_sila, t=t, y0=y0)
y_ivp = solve_ivp(f_zračni_upor_sila, t_span=(t[0], t[-1]), y0=y0, t_eval=t)
```

Poglejmo rezultat:

```
y_ivp.y[:, :5]

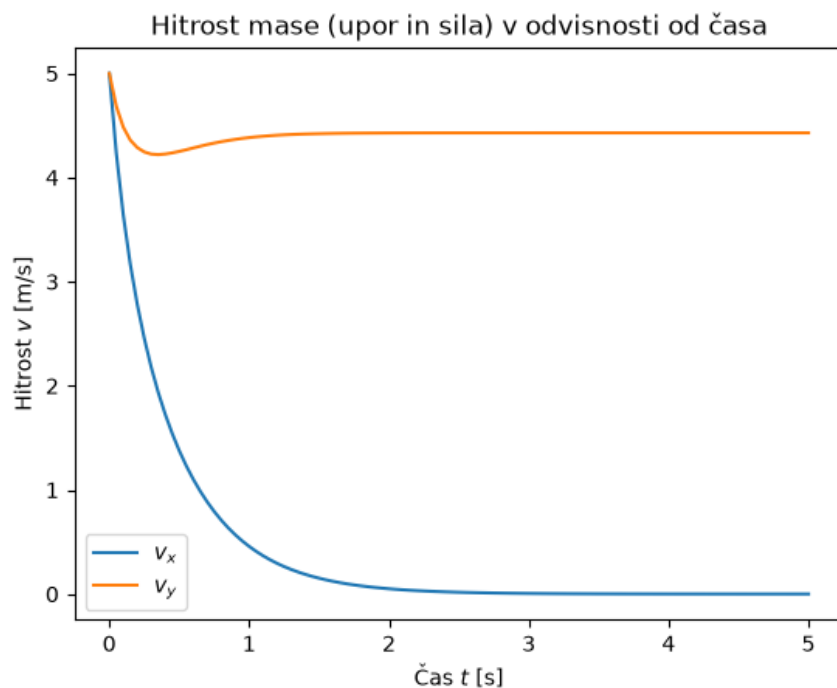
array([[5. , 4.23209907, 3.63858619, 3.16619484, 2.78087608],
       [5. , 4.68459713, 4.48359747, 4.35932897, 4.28742151]])

y_RK4[:5].T

array([[5. , 4.23211316, 3.6399137 , 3.16660748, 2.77780226],
       [5. , 4.68462761, 4.48494405, 4.35999931, 4.28466858]])
```

Prikažemo rezultat:

```
plt.plot(t, y_ivp.y[0], label='$v_x$')
plt.plot(t, y_ivp.y[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 :

$$y^{(n)} = f(t, y, y', y'', \dots, y^{(n-1)}), \quad (12.50)$$

pri začetnih pogojih:

$$y(t_0) = k_0, \quad y'(t_0) = k_1, \quad \dots \quad y^{(n-1)}(t_0) = k_{n-1} \quad (12.51)$$

preoblikujemo v sistem diferencialnih enačb prvega reda.

Najprej namesto odvodov vpeljemo nove spremenljivke:

$$y_i = y^{(i)}, \quad i = 0, 1, \dots, n-1. \quad (12.52)$$

Diferencialna enačba n -tega reda zapisana z novimi spremenljivkami je:

$$y'_{n-1} = f(t, y_0, y_1, y_2, \dots, y_{n-1}). \quad (12.53)$$

Nove funkcije odvajamo po neodvisni spremenljivki:

$$y'_i = y^{(i+1)} = y_{i+1}, \quad i = 0, 1, \dots, n-2 \quad (12.54)$$

in

$$y'_{n-1} = f(t, y_0, y_1, y_2, \dots, y_{n-1}). \quad (12.55)$$

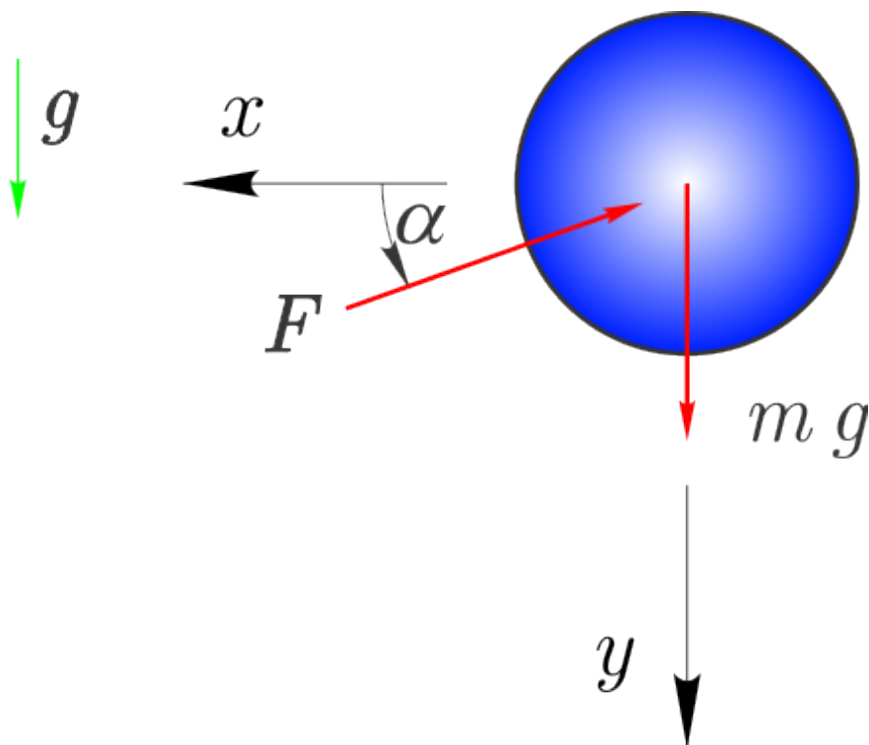
Dobili smo sistem navadnih diferencialnih enačb prvega reda:

$$\begin{aligned} y'_0 &= y_1 \\ y'_1 &= y_2 \\ &\dots \\ y'_{n-1} &= f(t, y_0, y_1, y_2, \dots, y_{n-1}) \end{aligned} \quad (12.56)$$

pri začetnih pogojih:

$$y_0(t_0) = k_0, \quad y_1(t_0) = k_1, \quad \dots \quad y_{n-1}(t_0) = k_{n-1}. \quad (12.57)$$

Numerični zgled



Vrnemo se k padajoči masi:

Vendar tokrat drugi Newtonov zakon zapišimo glede na pomik (diferencialna enačba drugega reda):

$$F_x = m x'', \quad (12.58)$$

$$m g + F_y = m y'', \quad (12.59)$$

m je masa, g gravitacijski pospešek, c koeficient zračnega upora ter x'' in y'' pospešek v izbranem koordinatnem sistemu. Začetni pogoji:

$$x(0) = y(0) = 0 \text{ m} \quad \text{in} \quad x'(0) = y'(0) = 5 \text{ m/s}. \quad (12.60)$$

Imamo sistem dveh diferencialnih enačb drugega reda. Z uvedbo novih spremenljivk y_i :

$$y_0 = x, \quad y_1 = x', \quad y_2 = y, \quad y_3 = y'. \quad (12.61)$$

Pripravimo sistem diferencialnih enačb prvega reda:

$$\begin{aligned} y_0' &= y_1 \\ y_1' &= F_x/m \\ y_2' &= y_3 \\ y_3' &= g + F_y/m \end{aligned} \quad (12.62)$$

Definirajmo Pythonovo funkcijo desnih strani / prvih odvodov:

```
def f_zračni_upor_vezana(t, y, 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:

```
y0 = np.array([0., 5., 0., 5.])
t = np.linspace(0, 5, 101)
t[:5]
```

```
| array([0.   , 0.05, 0.1  , 0.15, 0.2  ])
```

Izračunamo rešitev:

```
| y_ivp = solve_ivp(f_zračni_upor_vezana, t_span=(t[0], t[-1]), y0=y0, t_eval=t)
```

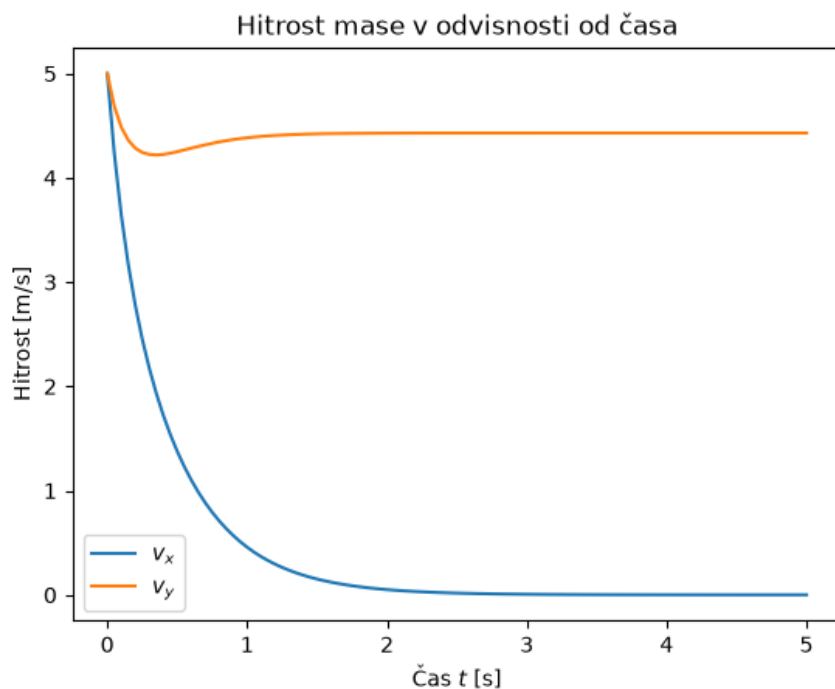
Poglejmo rezultat $([x, x', y, y'])$:

```
| y_ivp.y[:, :5]
```

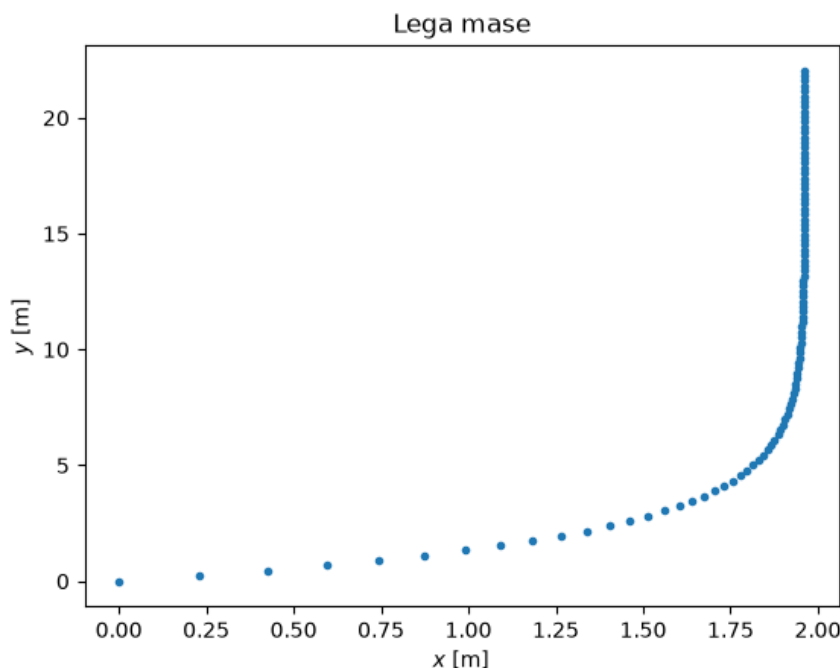
```
| array([[0.          , 0.22992477, 0.42611996, 0.59587483, 0.74424286],
|        [5.          , 4.23183658, 3.64002431, 3.16703014, 2.77775353],
|        [0.          , 0.24153615, 0.47038377, 0.69124852, 0.9072755  ],
|        [5.          , 4.68435696, 4.48497294, 4.36034218, 4.28444004]])
```

Prikažemo rezultate; najprej hitrost, nato lego (y koordinata je pozitivna navzdol)!

```
| plt.plot(t, y_ivp.y[1], label='$v_x$')
| plt.plot(t, y_ivp.y[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()
```



```
| plt.scatter(y_ivp.y[0], y_ivp.y[2], marker='.')
| plt.xlabel('$x$ [m]')
| plt.ylabel('$y$ [m]')
| plt.title('Lega mase')
| plt.show()
```



12.7 Stabilnost reševanja diferencialnih enačb*

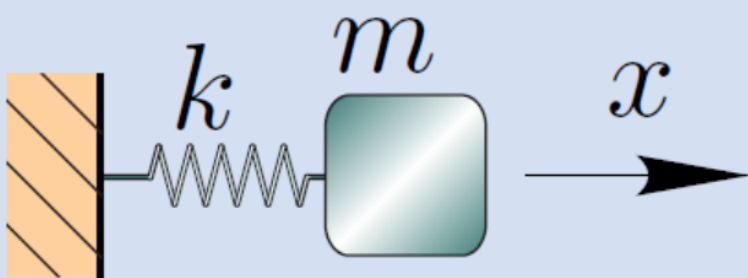
Numerično reševanje diferencialnih enačb je izpostavljeno *napaki metode* in *zaokrožitveni napaki*. Te napake so pri različnih numeričnih metodah različne.

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, 2017) prikazuje dinamski sistem (masa m , togost k), katerega diferencialna enačba je

$$m x'' + k x = 0. \quad (12.63)$$

Tako diferencialno enačbo preoblikujemo v standardno obliko lastnega nihanja:

$$x'' + \omega_0^2 x = 0, \quad (12.64)$$

kjer je lastna krožna frekvenca:

$$\omega_0 = \sqrt{\frac{k}{m}} \quad (12.65)$$

in pričakujemo odziv oblike:

$$x(t) = A \cos(\omega_0 t) + B \sin(\omega_0 t). \quad (12.66)$$

Če so začetni pogoji:

$$x(0\text{ s}) = x_0 \quad \text{in} \quad x'(0\text{ s}) = 0\text{ m/s}, \quad (12.67)$$

je rešitev začetnega problema:

$$x(t) = x_0 \cos(\omega_0 t). \quad (12.68)$$

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 $y_0 = x, y_1 = x'$:

```
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:

```
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
```

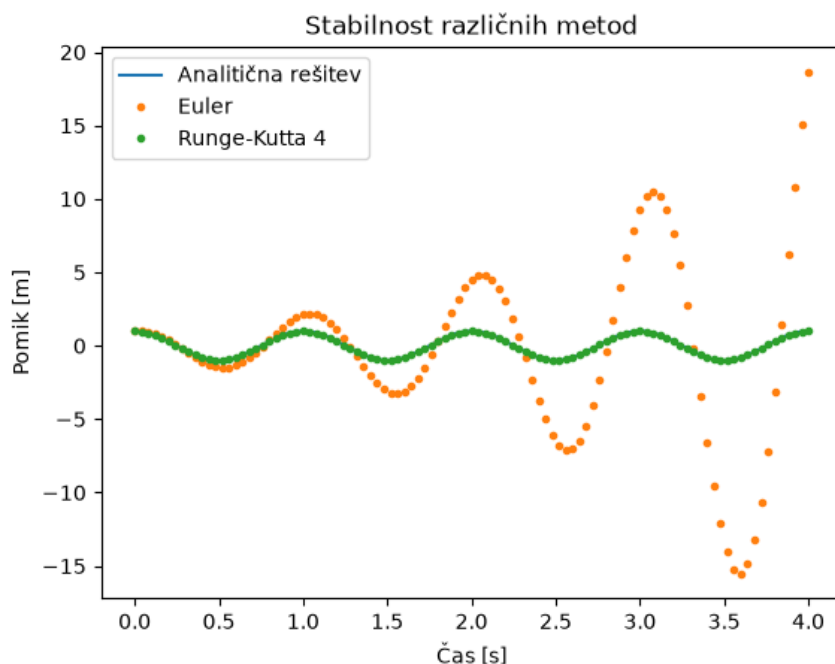
Rešitev s pomočjo metod Euler in Runge-Kutta četrtega reda:

```
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 = solve_ivp(f_nihalo, t_span=(t_RK4[0], t_RK4[-1]), y0=x_zacetni_pogoji,
t_eval=t_RK4).y
```

Prikažimo rezultate:

```
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?

```
| x_Eu[:10,0]
|
| array([ 1.          ,  1.          ,  0.93683453,  0.8105036 ,  0.62499707,
|         0.3882947 ,  0.1121141 , -0.18859332, -0.49638247, -0.79225904])
```

Spomnimo se Eulerjeve metode:

$$x(t+h) = x(t) + x'(t)h, \quad (12.69)$$

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:

$$x(t+h) = x(t) + x'(t)h = x_0 (\cos(\omega_0 t) - \omega_0 h \sin(\omega_0 t)). \quad (12.70)$$

Predpostavimo, da gledamo stanje ob takem času $t = \pi/(2\omega_0)$, ko velja $\cos(\omega_0 t) = 0$ in $\sin(\omega_0 t) = 1$:

$$x(t+h) = x_0 \underbrace{(-\omega_0 h)}_A. \quad (12.71)$$

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:

$$|A| < 1 \quad \rightarrow \quad h < \frac{1}{\omega_0}. \quad (12.72)$$

Opomba: v nekaterih knjigah boste videli tudi vrednost $h < 2/\omega_0$; enolične 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 Dodatno: 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](#).

Definirajmo seznam odvodov:

```
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]])

x_zacetni_pogoji = np.array([1.5, 0.])
dt = 0.1
t1 = 3000
```

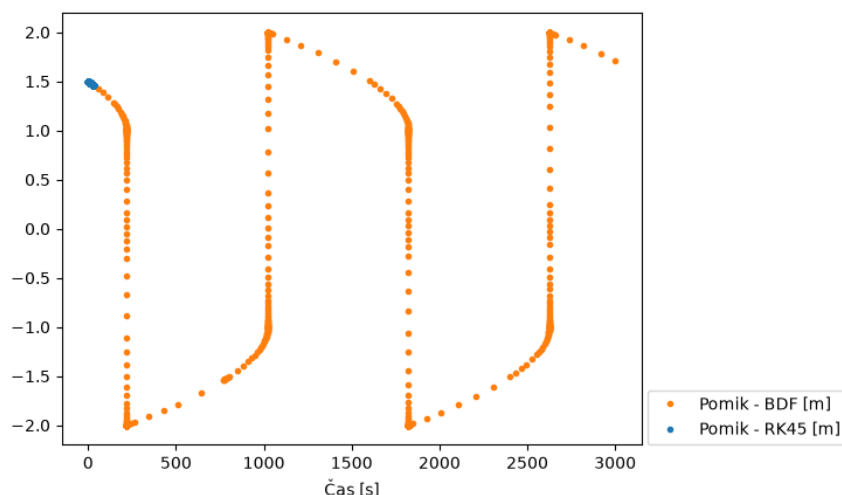
Rešitev po metodi RK45 (gre za eksplisitno shemo, ki ni primerna za toge sisteme diferencialnih enačb; reševanje je zelo počasno, zato rešitev računamo samo do $t1/100$):

```
vp_RK45 = solve_ivp(f_van_der_pol, t_span=(0., t1/100), y0=x_zacetni_pogoji,
                    method='RK45')
```

Implicitna shema BDF (*angl.* Backward Differentiation Formulas) se tukaj izkaže kot bistveno bolj primerna. Zaradi stabilnosti, so koraki lahko bistveno večji in zato je reševanje bistveno hitrejše:

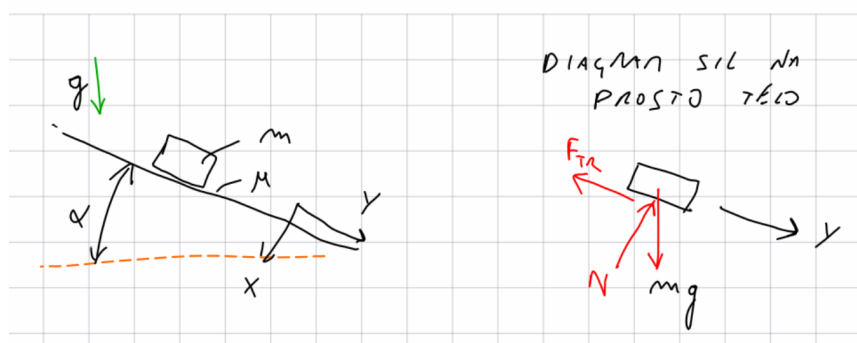
```
vp_BDF = solve_ivp(f_van_der_pol, t_span=(0., t1), y0=x_zacetni_pogoji, method='BDF')

plt.plot(vp_BDF.t, vp_BDF.y[0], 'C1.', label='Pomik - BDF [m]')
plt.plot(vp_RK45.t, vp_RK45.y[0], 'C0.', label='Pomik - RK45 [m]')
plt.xlabel('Čas [s]')
plt.legend(loc=(1.01, 0));
```



12.9 Dodatno: 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 $\dot{y}(0\text{s}) = v_0$, pomik $y(0\text{s}) = 0\text{m}$.



Gibalna enačba (samo za smer y) je definirana glede na II. Newtonov zakon (glejte diagram sil na prosto telo).

Izpeljava gibalne enačbe

```
y = sym.Function('y')
m, mu, g, alpha, t, v0 = sym.symbols('m, mu, g, alpha, t, v0')
eq = sym.Eq(m*y(t).diff(t,2), m*g*sym.sin(alpha) - m*g*sym.cos(alpha)*mu)
eq
```

$$m \frac{d^2}{dt^2} y(t) = -gm\mu \cos(\alpha) + gm \sin(\alpha) \quad (12.73)$$

Rešitev enačbe je:

```
dsol = sym.dsolve(eq, y(t))
dsol
```

$$y(t) = C_1 + C_2 t + \frac{gt^2(-\mu \cos(\alpha) + \sin(\alpha))}{2} \quad (12.74)$$

Da določimo C_1 in C_2 , vstavimo $t = 0\text{s}$:

```
dsol.args[1].subs(t, 0)
```

$$C_1 \quad (12.75)$$

Nato odvajamo po času in ponovno vstavimo $t = 0$ s:

```
dsol.args[1].diff(t).subs(t, 0)
```

$$C_2 \quad (12.76)$$

Glede na začetne pogoje smo torej določili konstante:

```
zacetni_pogoji = {'C1': 0, 'C2': v0}
```

Sledi rešitev:

```
resitev = dsol.args[1].subs(zacetni_pogoji)
resitev
```

$$\frac{gt^2(-\mu \cos(\alpha) + \sin(\alpha))}{2} + tv_0 \quad (12.77)$$

Pripravimo si funkciji za numerični klic:

```
podatki = {mu: 0.3, alpha: 15*np.pi/180, v0: 1., g: 9.81} #tukaj uporabimo np.pi, da
imamo numerično vrednost
```

```
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)))
```

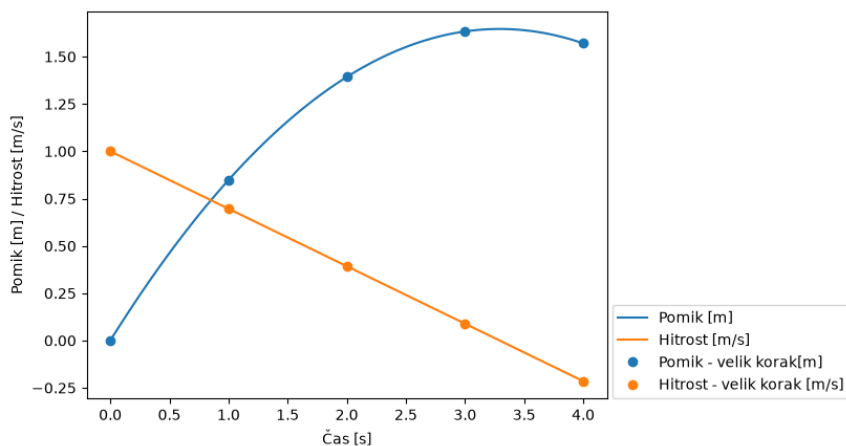
```
Pomik pri 0s: 0m
Hitrost pri 0s: 1m/s
```

Pripravimo prikaz:

```
cas = np.linspace(0, 4, 100)
cas2 = np.linspace(0, 4, 5)
```

```
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()
```

```
slika()
```



Simbolno preoblikovanje diferencialne enačbe v sistem diferencialnih enačb prvega reda

Spomnimo se izvirne diferencialne enačbe:

| eq

$$m \frac{d^2}{dt^2} y(t) = -gm\mu \cos(\alpha) + gm \sin(\alpha) \quad (12.78)$$

Definirajmo nove spremenljivke in pripravimo funkcijo f :

```
| y0 = sym.Function('y0')
| y1 = sym.Function('y1')
| f = sym.simplify(eq.args[1]/m)
| f
```

$$g(-\mu \cos(\alpha) + \sin(\alpha)) \quad (12.79)$$

Povežimo sedaj nove spremenljivke.

dy_0/dt naj bo enako y_1 :

```
| eq1 = sym.Eq(y0(t).diff(t), y1(t))
| eq1
```

$$\frac{d}{dt} y_0(t) = y_1(t) \quad (12.80)$$

Odvod dy_1/dt (v bistvu je to y'') naj bo enak funkciji f :

```
| eq2 = sym.Eq(y1(t).diff(t), f)
| eq2
```

$$\frac{d}{dt} y_1(t) = g(-\mu \cos(\alpha) + \sin(\alpha)) \quad (12.81)$$

Zgornje izraze zapišemo v vektorski obliki:

$$\mathbf{y}' = \mathbf{f}(t, \mathbf{y}). \quad (12.82)$$

```
| y_odvod = [y0(t).diff(t), y1(t).diff(t)]
| y_odvod
```

$$\left[\frac{d}{dt}y_0(t), \frac{d}{dt}y_1(t) \right] \quad (12.83)$$

```
f_vec = [y1(t), f]
f_vec
```

$$[y_1(t), g(-\mu \cos(\alpha) + \sin(\alpha))] \quad (12.84)$$

Spomnimo se sedaj **f_vec**:

```
f_vec
```

$$[y_1(t), g(-\mu \cos(\alpha) + \sin(\alpha))] \quad (12.85)$$

Če rešujemo numerično, potem funkcijo **f(t,y)** zapišemo:

```
pospesek = float((eq.args[1]/m).simplify().subs(podatki))
pospesek #raziščite zakaj smo tukaj tako definirali! namig: type(pospesek)
```

$$-0.303704874312999 \quad (12.86)$$

```
def F_klada(t, y):
    return np.array([y[1], pospesek], dtype=float)
```

Preverimo funkcijo pri začetnem času $t = 0$ s in pri začetnih pogojih $[y_0, y_1] = [0, v_0]$:

```
y_zacetni_pogoji = np.array([0, podatki[v0]])
y_zacetni_pogoji
```

```
array([0., 1.])
```

```
F_klada(0., y_zacetni_pogoji)
```

```
array([ 1., -0.30370487])
```

Uporabimo sedaj Eulerjevo metodo:

```
%%timeit
x_Eu = np.linspace(0, 4, 5)
y_Eu = euler_sistem(F_klada, x_Eu, np.array([0, 1.]))
y_Eu
```

```
array([[ 0., 1.],
       [ 1., 0.69629513],
       [ 1.69629513, 0.39259025],
       [ 2.08888538, 0.08888538],
       [ 2.17777075, -0.2148195 ]])
```

Prikažemo in primerjamo z analitično rešitvijo:

```
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), 'CO', label='Pomik - analitično [m]')
    plt.plot(cas, hitrost(cas), 'C1', label='Hitrost - analitično [m/s]')
    plt.plot(x_Eu, y_Eu[:, 0], 'CO.', 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();
```

```
| interact(narisi_euler, n=(3, 10, 1));  
  
| interactive(children=(IntSlider(value=5, description='n', max=10, min=3), Output()),  
| _dom_classes=('widget -int...
```

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:

$$\ddot{y} = f(t, y, \dot{y}), \quad (13.1)$$

ob predpisanih *robni*h pogojih:

$$y(a) = \alpha \quad \text{in} \quad y(b) = \beta. \quad (13.2)$$

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. **streška metoda**,
2. **metoda končnih razlik**.

13.2 Streška metoda

Rešujemo robni problem:

$$\ddot{y} = f(t, y, \dot{y}), \quad y(a) = \alpha, \quad y(b) = \beta, \quad (13.3)$$

ki ga prevedemo na začetni problem tako, da si izberemo:

$$\dot{y}(a) = u. \quad (13.4)$$

Problem rešimo z numeričnimi metodami reševanja začetnega problema in rešitev označimo z $\theta(u, t)$.

Robni problem rešimo, ko izberemo u tako, da velja:

$$r(u) = \theta(u, b) - \beta = 0. \quad (13.5)$$

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:

$$u_{i+1} = u_i - r(u_i) \frac{u_i - u_{i-1}}{r(u_i) - r(u_{i-1})}, \quad i = 2, 3, \dots \quad (13.6)$$

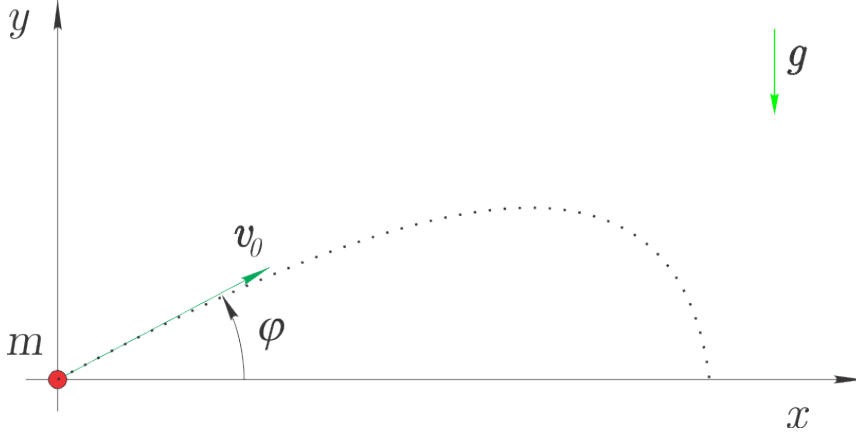
Zaključimo, ko je:

$$|r(u_{i+1})| < \epsilon. \quad (13.7)$$

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:

$$x''(t) = F_x/m \quad y''(t) = F_y/m - g. \quad (13.8)$$

Komponente sile so (glejte izpeljavo pri poglavju iz reševanja začetnega problema sistema diferencialnih enačb):

$$F_x = -c x' \sqrt{x'^2 + y'^2}, \quad F_y = -c y' \sqrt{x'^2 + y'^2}. \quad (13.9)$$

Vertikalni met

Najprej predpostavimo, da je $\varphi = 90^\circ$ in torej v x smeri nimamo gibanja. Zanima nas rešitev, ko izstrelek izstrelimo iz višine $y = 0$ m in mora pri času $t = b = 1$ s biti na višini $y(b) = 10$ m. Definirali smo robni problem:

$$y''(t) = F_y/m - g, \quad y(0) = 0, \quad y(1) = 10. \quad (13.10)$$

Najprej moramo enačbo drugega reda:

$$y'' = f(t, y, y') = F_y/m - g \quad (13.11)$$

preoblikovati na sistem dveh enačb prvega reda. Uporabimo $y_i = y^{(i)}$ ter upoštevamo $F_y = -c y' \sqrt{y'^2}$.

Odvajamo $y'_i = y_{i+1}$ in pripravimo sistem enačb prvega reda:

$$\begin{aligned} y'_0 &= y_1 \\ y'_1 &= -c y_1 \sqrt{y_1^2}/m - g \end{aligned} \quad (13.12)$$

Uvozimo modula `numpy` in `scipy.integrate.solve_ivp`:

```
import numpy as np
from scipy.integrate import solve_ivp
```

Pripravimo seznam funkcij desnih strani:

```
def f_vert(t, y, g=9.81, m=1., c=0.5):
    return np.array([y[1], -g -c*y[1]*np.sqrt(y[1]**2)/m])
```

ter pripravimo funkcijo za izračun mejnega preostanka pri času b v odvisnosti od začetne hitrosti v_0 (privzeti zračni upor je $c = 0.1$):

```
def r(v0=100., t=None, ciljna_lega=10., g=9.81, m=1., c=0.1):
    def f_vert_tmp(t, y): # pripravimo funkcijo z željenimi parametri
        return f_vert(t, y, g=g, m=m, c=c)
    sol = solve_ivp(f_vert_tmp, t_span=(t[0], t[-1]), y0=np.array([0., v0]))
    r = sol.y[0, -1] - ciljna_lega
    return r
```

Preverimo mejni preostanek pri začetnem pogoju $v_0 = y' = 50$ m/s:

```
t = np.linspace(0, 1, 100)
r(v0=50., t=t)

np.float64(5.63455093999735)
```

Opazimo, da je masa pri 1 sekundi 5,635 m nad ciljno višino.

Naš cilj je, da pri 1 sekundi masa doseže lego 10 m z natančnostjo $1e-6$:

```
ciljna_lega = 10
epsilon = 1e -6
```

Izvedimo sedaj sekantno metodo:

```
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 = r(v0=x2, t=t, ciljna_lega=ciljna_lega)
    print(f'Novi približek je v0={x2}, napaka je {err}.')
    x0 = x1
    x1 = x2
    if abs(err)<epsilon:
        rešitev = x2
        print(f'Rešitev v0={rešitev}')
        break
```

```
Novi približek je v0=5.644178759347113, napaka je -9.672536331804414.
Novi približek je v0=33.67258709458857, napaka je 2.2187017725280302.
Novi približek je v0=28.442965190719487, napaka je 0.8225839451367332.
Novi približek je v0=25.361704474139284, napaka je -0.10153714557803362.
Novi približek je v0=25.70025579721459, napaka je 0.004276366407131249.
Novi približek je v0=25.686573522825174, napaka je 2.16308919487318e -05.
Novi približek je v0=25.686503962731987, napaka je -4.623860405672531e -09.
Rešitev v0=25.686503962731987
```

Poglejmo si sedaj izračunano rešitev:

```
sol = solve_ivp(lambda t, y: f_vert(t, y, c=0.1),
                t_span=(t[0], t[-1]), y0=np.array([0., rešitev]),
                t_eval=t
                )
```

Zgoraj smo uporabili `lambda` izraz ([dokumentacija](#)). Izraz `lambda t, y: f_vert(t, y, c=0.1)` je ekvivalenten:

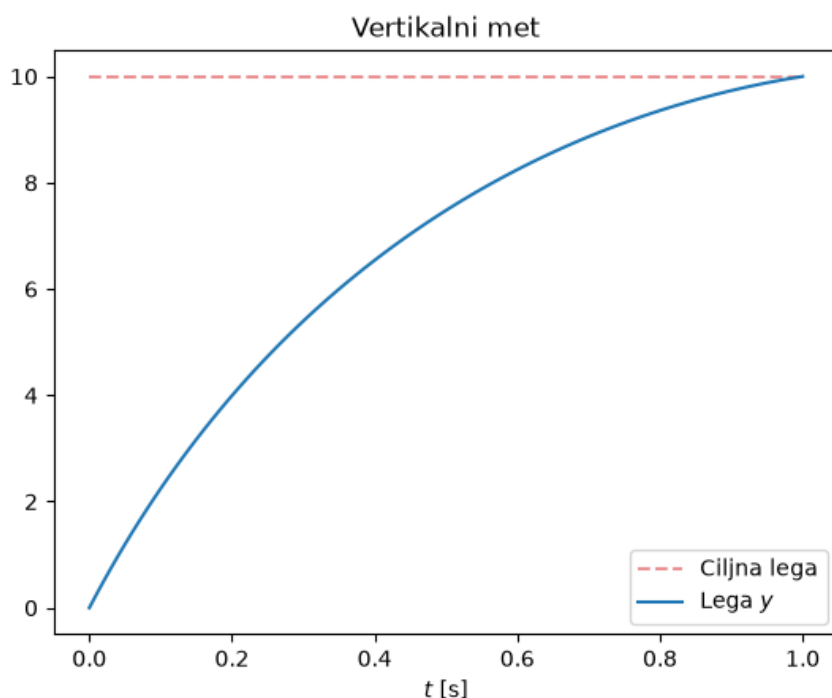
```
def ime_funkcije(t, y):
    return f_vert(t, y, c=0.1)
```

Uvozimo matplotlib:

```
import matplotlib.pyplot as plt
%matplotlib inline
```

Prikažemo rezultat:

```
plt.title('Vertikalni met')
plt.hlines(ciljna_lega, 0., 1, 'C3', linestyle='dashed', label='Ciljna lega',
alpha=0.5)
plt.plot(sol.t, sol.y[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:

$$y_0 = x, \quad y_1 = x', \quad y_2 = y, \quad y_3 = y' \quad (13.13)$$

in dobimo sistem diferencialnih enačb prvega reda:

$$\begin{aligned} y_0' &= y_1 \\ y_1' &= F_x/m \\ y_2' &= y_3 \\ y_3' &= F_y/m - g. \end{aligned} \quad (13.14)$$

Pripravimo seznam funkcij desnih strani:

```
def f_poševno(t, y, 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 $\mathbf{v}_0$:

```
def r_poševno(v0=[5., 100.], t=None, ciljna_lega=np.array([10, 5.]), g=9.81, m=1.,
c=0.1):
    sol = solve_ivp(lambda t, y: f_poševno(t, y, g=g, m=m, c=c),
                    t_span=(t[0], t[-1]),
                    y0=[0, v0[0], 0, v0[1]])
    r = sol.y[0:3:2, -1] - ciljna_lega
    return r
```

Preverimo mejni preostanek pri začetnem pogoju $\mathbf{v}_0 = [100., 100.]$ m/s:

```
r_poševno(v0=[100., 100.], t=t)
array([ 9.66195325, 11.84726075])
```

Za iskanje korena sistema nelinearnih funkcij smo že spoznali funkcijo `scipy.optimize.root` ([dokumentacija](#)):

```
root(fun, x0, args=(), method='hybr', jac=None,
     tol=None, callback=None, options=None)
```

Najprej jo uvozimo:

```
from scipy import optimize
```

Potem uporabimo z začetnim ugibanjem $\mathbf{v}_0$:

```
rešitev = optimize.root(r_poševno, np.array([100., 100.]), args=(t,))
```

Rešitev je:

```
rešitev
message: The solution converged.
success: True
status: 1
  fun: [ 1.155e -13 -3.260e -13]
   x: [ 1.938e+01  1.655e+01]
method: hybr
  nfev: 19
  fjac: [[ -9.339e -01  3.575e -01]
         [-3.575e -01 -9.339e -01]]
   r: [ -4.078e -01  2.409e -01 -3.754e -01]
  qtf: [ 1.067e -09 -1.311e -09]
```

Atribut `rešitev.x` vsebuje vektor izračunanih rešitev ($[19.37894314, 16.55482478]$). Preverimo mejni preostanek pri izračunani rešitvi:

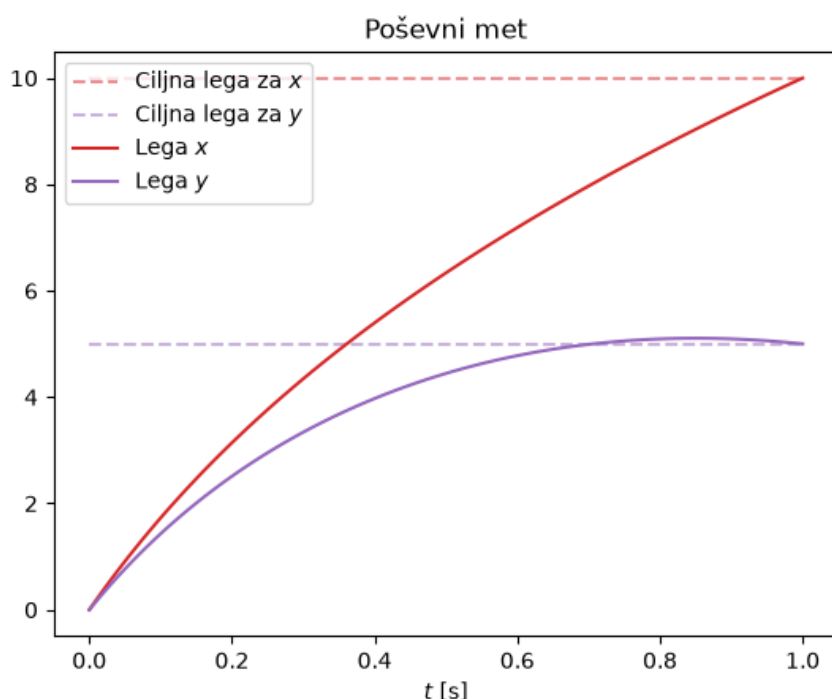
```
r_poševno(v0=rešitev.x, t=t)
array([ 1.15463195e -13, -3.25961480e -13])
```

Poglejmo si sedaj izračunano rešitev:

```
sol = solve_ivp(lambda t, y: f_poševno(t, y, g=9.81, m=1., c=.1),
                t_span=(t[0], t[-1]),
                y0=[0, rešitev.x[0], 0, rešitev.x[1]],
                t_eval=t)
```

Prikažemo rezultat:

```
plt.title('Poševni met')
plt.hlines(10,0., 1, 'C3', linestyle='dashed', label='Ciljna lega za $x$',
alpha=0.5)
plt.hlines(5,0., 1, 'C4', linestyle='dashed', label='Ciljna lega za $y$', alpha=0.5)
plt.plot(sol.t, sol.y[0], label='Lega $x$', color='C3')
plt.plot(sol.t, sol.y[2], label='Lega $y$', color='C4')
plt.xlabel('$t$ [s]')
plt.legend()
plt.show()
```



13.2.2 Uporaba `scipy.integrate.solve_bvp`

Namesto `scipy.integrate.solve_ivp` in `scipy.optimize` lahko uporabimo vgrajeno funkcijo za reševanje robnih problemov `scipy.integrate.solve_bvp` (BVP - *angl.* Boundary Value Problem):

```
scipy.integrate.solve_bvp(fun, bc, x, y,
                          p=None, S=None, fun_jac=None, bc_jac=None,
                          tol=0.001, max_nodes=1000, verbose=0)
```

Pojasnilo vseh argumentov je v [dokumentaciji](#), tukaj bomo izpostavili nekatere:

- `fun` je desna stran (`func(t, y...)`),
- `bc` je mejni preostanek: `bc(y(a), y(b)) = 0`
- `x` numerično polje (dimenzija (`m`)) neodvisne spremenljivke `x[0]=a` in `x[-1]=b`,
- `y` numerično polje (dimenzija (`n, m`)) začetnih vrednosti.

Rezultat klicanja `solve_bvp` je objekt z atributi (izbrani):

- `x` vrednosti neodvisne spremenljivke pri katerih je izračunan rezultat,

- `y` rezultat,
- `sol` rezultat v obliki kubičnega zlepk,
- `success` je `True`, če je rešitev konvergirala.

Poglejmo primer:

```
from scipy.integrate import solve_bvp
```

Definirajmo neodvisno spremenljivko

```
t = np.linspace(0, 1, 15)
```

In mejni preostanek (želimo, da je pri času `b` lega $x = 10$ m in $y = 5$ m):

```
def mejni_preostanek(ya, yb):
    return np.array([ya[0], ya[2], yb[0] - 10, yb[2] - 5])
```

Definirajmo še začetne vrednosti hitrosti (kot začetno ugibanje):

```
y0 = np.zeros((4, t.size))
y0[1] = 5 # hitrost v x smeri
y0[3] = 100 # hitrost v y smeri
```

```
y0[:, :3]
```

```
array([[ 0.,  0.,  0.],
       [ 5.,  5.,  5.],
       [ 0.,  0.,  0.],
       [100., 100., 100.]])
```

In rešimo robni problem in opis rezultata

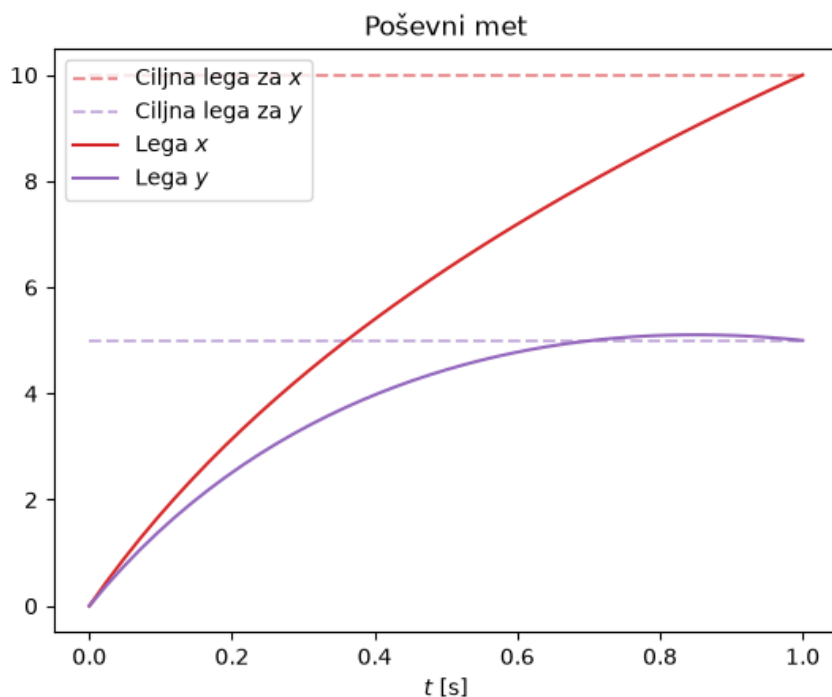
```
sol = solve_bvp(lambda t, y: f_poševno(t, y, g=9.81, m=1., c=.1),
                bc=mejni_preostanek, x=t, y=y0)
```

```
sol
```

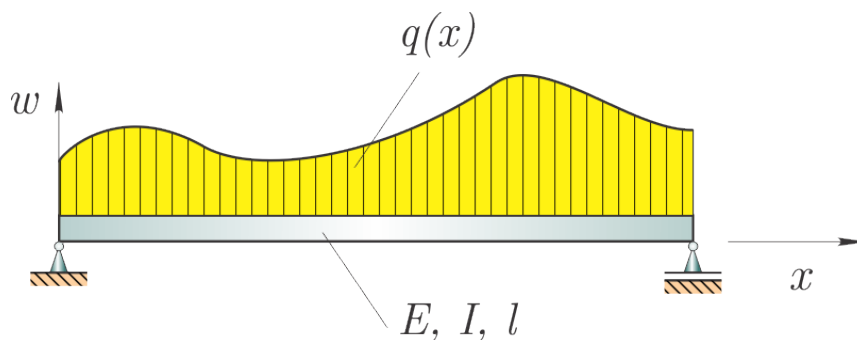
```
message: The algorithm converged to the desired accuracy.
  success: True
  status: 0
    x: [ 0.000e+00  2.381e -02 ...  9.762e -01  1.000e+00]
  sol: <scipy.interpolate._interpolate.PPoly object at 0x7e3e20de7a10>
    p: None
    y: [[ -1.588e -22  4.482e -01 ...  9.853e+00  1.000e+01]
       [ 1.939e+01  1.828e+01 ...  6.210e+00  6.117e+00]
       [ -3.176e -22  3.801e -01 ...  5.031e+00  5.000e+00]
       [ 1.656e+01  1.539e+01 ... -1.189e+00 -1.403e+00]]
  yp: [[ 1.939e+01  1.828e+01 ...  6.210e+00  6.117e+00]
       [ -4.944e+01 -4.369e+01 ... -3.926e+00 -3.839e+00]
       [ 1.656e+01  1.539e+01 ... -1.189e+00 -1.403e+00]
       [ -5.205e+01 -4.659e+01 ... -9.058e+00 -8.929e+00]]
  rms_residuals: [ 4.302e -05  3.582e -05 ...  1.297e -06  4.151e -06]
    niter: 2
```

Prikažimo rezultat:

```
plt.title('Poševni met')
plt.hlines(10, 0., 1, 'C3', linestyle='dashed', label='Ciljna lega za $x$',
alpha=0.5)
plt.hlines(5, 0., 1, 'C4', linestyle='dashed', label='Ciljna lega za $y$', alpha=0.5)
plt.plot(sol.x, sol.y[0], label='Lega $x$', color='C3')
plt.plot(sol.x, sol.y[2], label='Lega $y$', color='C4')
plt.xlabel('$t$ [s]')
plt.legend()
plt.show()
```



13.2.3 Numerični zgled: nosilec z obremenitvijo



Poglejmo si nosilec:

Poves $w(x)$ nosilca popiše diferencialna enačba četrtega reda:

$$-EI \frac{d^4}{dx^4} w(x) + q(x) = 0. \quad (13.15)$$

Znane konstante so E, I, l in je $q(x)$ porazdeljena obremenitev.

Robni pogoji (členkasto vpet nosilec):

$$w(0) = w(l) = 0 \quad \text{in} \quad w''(0) = w''(l) = 0 \quad (13.16)$$

Parametri so:

- $I = 2.1 \cdot 10^{-5} \text{ m}^4$,
- $E = 2.1 \cdot 10^{11} \text{ N/m}^2$,
- $l = 10 \text{ m}$.

Porazdeljena obremenitev $q(x)$ bo definirana pozneje.

Najprej moramo diferencialno enačbo četrtega reda preoblikovati v sistem diferencialnih enačb prvega reda. Uporabimo:

$$y_0 = w, \quad y_1 = w', \quad y_2 = w'', \quad y_3 = w''' \quad (13.17)$$

in dobimo sistem diferencialnih enačb prvega reda:

$$\begin{aligned}
 y'_0 &= y_1 \\
 y'_1 &= y_2 \\
 y'_2 &= y_3 \\
 y'_3 &= q(x)/(EI).
 \end{aligned}
 \tag{13.18}$$

Pripravimo različne porazdeljene obremenitve:

```
def q_konstanta(x, F_0=1e3, l=10):
    return -F_0*np.ones_like(x)

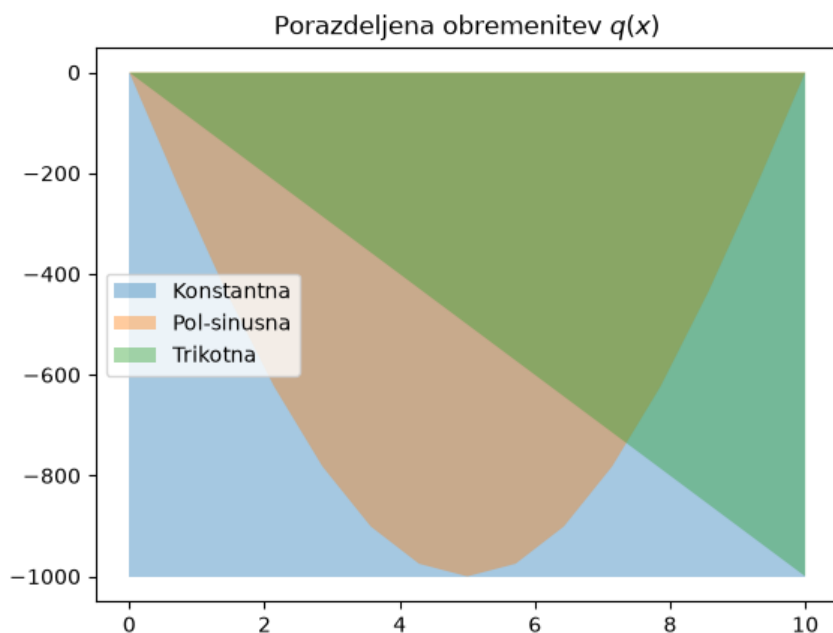
def q_trikotna(x, F_0=1e3, l=10):
    return -F_0*x/l

def q_pol_sinusna(x, F_0=1e3, l=10):
    return -F_0*np.sin(np.pi*x/l)
```

Definirajmo dolžino (ostale parametre, I, E , bomo uporabili privzete):

```
l = 10.
```

```
x = np.linspace(0, l, 15)
plt.fill_between(x, q_konstanta(x), alpha=0.4, label='Konstantna')
plt.fill_between(x, q_pol_sinusna(x), alpha=0.4, label='Pol -sinusna')
plt.fill_between(x, q_trikotna(x), alpha=0.4, label='Trikotna')
plt.title('Porazdeljena obremenitev $q(x)$')
plt.legend()
plt.show()
```



Pripravimo seznam funkcij desne strani:

```
def f_nosilec_konstanta(x, y, E=2.1e11, I=2.1e-5):
    return np.array([y[1], y[2], y[3], q_konstanta(x)/(E*I)], dtype=float)
def f_nosilec_pol_sinusna(x, y, E=2.1e11, I=2.1e-5):
    return np.array([y[1], y[2], y[3], q_pol_sinusna(x)/(E*I)])
def f_nosilec_trikotna(x, y, E=2.1e11, I=2.1e-5):
    return np.array([y[1], y[2], y[3], q_trikotna(x)/(E*I)])
```

Definirajmo sedaj robne pogoje oz. mejni preostanek (poves in moment sta na robovih enaka nič):

```
def mejni_nosilec(ya, yb):
    return np.array([ya[0], ya[2], yb[0], yb[2]])
```

Definirajmo še začetne vrednosti naklona (w') in strižne sile (w''') (kot začetno ugibanje):

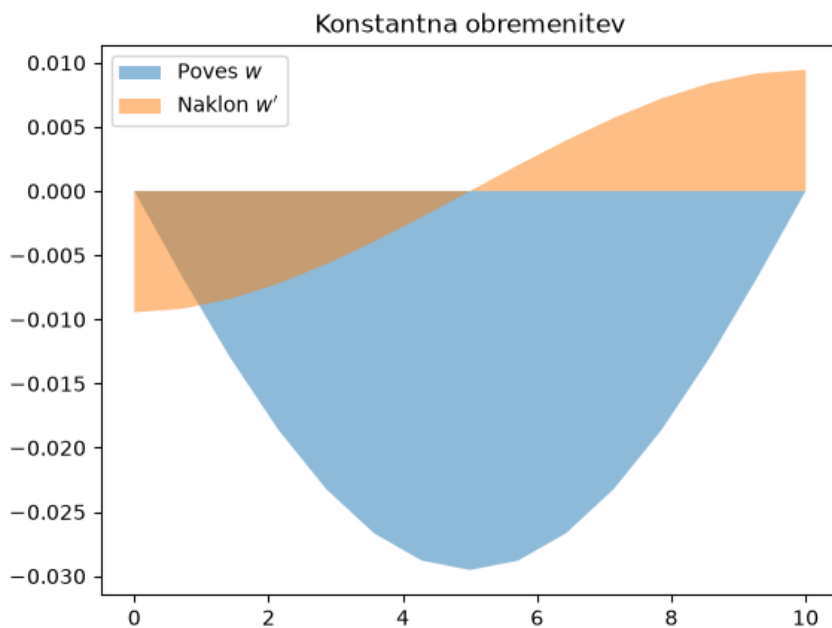
```
y0 = np.zeros((4, x.size), dtype=float)
# lahko tisti, ki niso znani ne bi bili nič:
# y0[1] = 1. # w'
# y0[3] = 1. # w'''
```

In rešimo robni problem (za vse tri tipe obremenitve):

```
sol_ko = solve_bvp(f_nosilec_konstanta, bc=mejni_nosilec, x=x, y=y0)
sol_pol_sin = solve_bvp(f_nosilec_pol_sinusna, bc=mejni_nosilec, x=x, y=y0)
sol_trikotna = solve_bvp(f_nosilec_trikotna, bc=mejni_nosilec, x=x, y=y0)
```

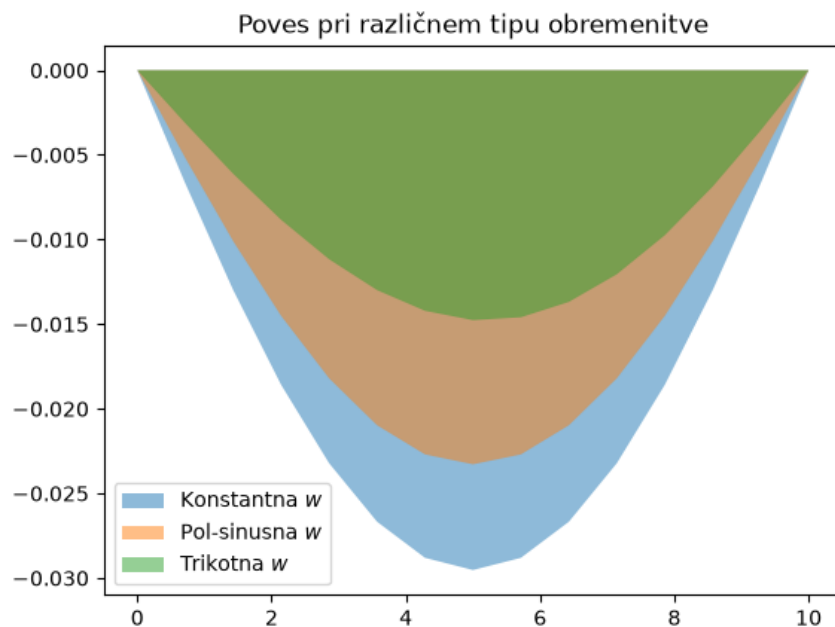
Rezultat prikažimo za konstantno obremenitev:

```
plt.fill_between(sol_ko.x, sol_ko.y[0], label='Poves $w$', alpha=0.5)
plt.fill_between(sol_ko.x, sol_ko.y[1], label='Naklon $w\$', alpha=0.5)
plt.title('Konstantna obremenitev')
plt.legend()
plt.show()
```



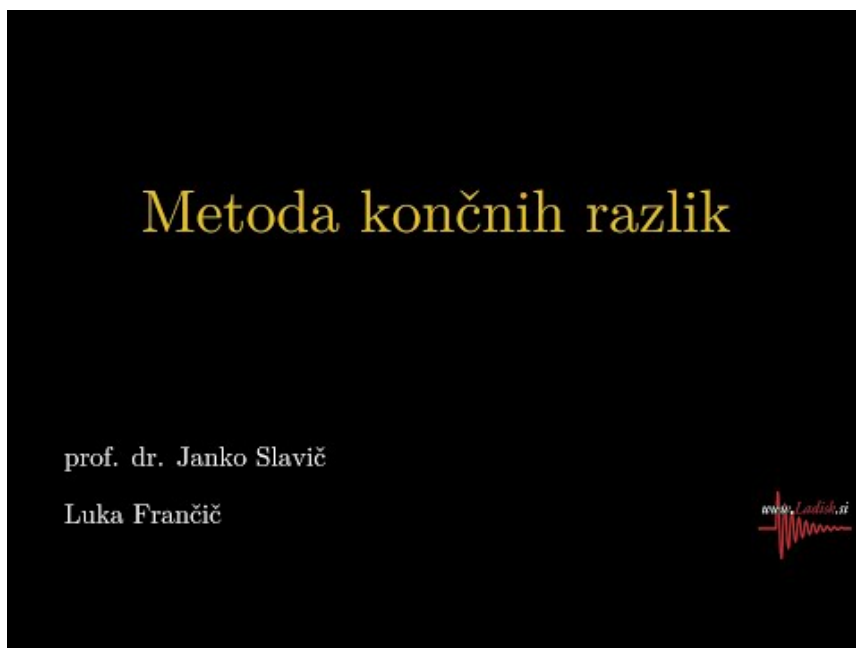
In primerjavo povesa za različne tipe obremenitve:

```
plt.fill_between(sol_ko.x, sol_ko.y[0], label='Konstantna $w$', alpha=0.5)
plt.fill_between(sol_pol_sin.x, sol_pol_sin.y[0], label='Pol -sinusna $w$', alpha=0.5)
plt.fill_between(sol_trikotna.x, sol_trikotna.y[0], label='Trikotna $w$', alpha=0.5)
plt.title('Poves pri različenem tipu obremenitve')
plt.legend()
plt.show()
```



13.3 Metoda končnih razlik

```
from IPython.display import YouTubeVideo
YouTubeVideo('YozKGsNYZ6Y', width=800, height=300)
```



Rešujemo robni problem:

$$y'' = f(t, y, y'), \quad y(a) = \alpha, \quad y(b) = \beta. \quad (13.19)$$

Velja torej:

$$a = t_0, \quad b = t_{n-1}. \quad (13.20)$$

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):

$$t = [t_0, t_1, \dots, t_{n-1}]. \quad (13.21)$$

Odhode nadomestimo s centralno diferenčno shemo:

Odvod $\downarrow \setminus$ Vrednosti $\rightarrow$	y_{i-2}	y_{i-1}	y_i	y_{i+1}	y_{i+2}
$y'_i = \frac{1}{h}$	0	-0.5	0	0.5	0
$y''_i = \frac{1}{h^2}$	0	1	-2	1	0
$y'''_i = \frac{1}{h^3}$	-0.5	1	0	-1	0.5
$y^{(4)}_i = \frac{1}{h^4}$	1	-4	6	-4	1

Prikazana centralna diferenčna shema ima napako drugega reda $\mathcal{O}(h^2)$.

V i -ti točki navadno diferencialno enačbo drugega reda s centralnimi diferencami zapišemo:

$$\frac{1}{h^2} (y_{i-1} - 2y_i + y_{i+1}) + \mathcal{O}(h^2) = f\left(t_i, y_i, \frac{1}{2h} (-y_{i-1} + y_{i+1}) + \mathcal{O}(h^2)\right). \quad (13.22)$$

Če zanemarimo napako metode:

$$\frac{1}{h^2} (y_{i-1} - 2y_i + y_{i+1}) = f\left(t_i, y_i, \frac{1}{2h} (-y_{i-1} + y_{i+1})\right), \quad i = 1, 2, \dots, n-2. \quad (13.23)$$

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:

$$y_0 = \alpha, \quad y_{n-1} = \beta. \quad (13.24)$$

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:

$$y(t_i) = y_{i,h} + E_{i,h}, \quad (13.25)$$

kjer je $y_{i,h}$ numerični približek in E_h napaka metode. Ker je globalna napaka drugega reda, lahko napako zapišemo:

$$E_{i,h} = k h^2, \quad (13.26)$$

Podobno lahko za velikost koraka $2h$ zapišemo:

$$y(t_j) = y_{j,2h} + E_{j,2h}, \quad (13.27)$$

kjer je $y_{j,2h}$ numerični približek in E_{2h} napaka metode in je:

$$E_{j,2h} = k (2h)^2 = 4k h^2 \quad (13.28)$$

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 $i = 2j$, $j = 1, 2, \dots$):

$$y_{i,h} + k h^2 = y_{j,2h} + 4k h^2 \quad (13.29)$$

sledi:

$$3 k h^2 = y_{2j,h} - y_{j,2h} \quad (13.30)$$

in nato izračunamo *oceno napake*:

$$E_{j,h} = \frac{y_{2j,h} - y_{j,2h}}{3}. \quad (13.31)$$

13.3.1 Numerični zgled: vertikalni met

Zgoraj smo vertikalni met rešili s strelesko metodo; uporabimo sedaj metodo končnih razlik. Najprej robni problem:

$$y''(t) = F_y/m - g, \quad y(0) = 0, \quad y(1) = 10. \quad (13.32)$$

zapisati s pomočjo centralne diferenčne sheme ($F_y = -c y'$):

$$\frac{1}{h^2} (y_{i-1} - 2y_i + y_{i+1}) = -c \left(\frac{1}{2h} (-y_{i-1} + y_{i+1}) \right) / m - g. \quad (13.33)$$

(Tukaj smo predpostavili, da je zračni upor linearno odvisen od hitrosti. V nasprotnem primeru bi imeli nelinearni robni problem, in posledično razvili sistem nelinearnih enačb.)

Zgornji izraz preoblikujemo:

$$(2 - c h/m) y_{i-1} - 4 y_i + (2 + c h/m) y_{i+1} = -2 g h^2, \quad i = 1, 2, \dots, n-2. \quad (13.34)$$

Robna pogoja sta:

$$y_0 = 0, \quad y_{n-1} = 10 \quad (13.35)$$

Robni problem smo torej preoblikovali na sistem n linearnih enačb. Poglejmo si sedaj konkreten izračun za $n = 11$; najprej definirajmo konstante, časovni vektor $\mathbf{t}$ in korak h :

```
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 $\mathbf{A}$.

Pomagamo si s funkcijo `numpy.diag()` ([dokumentacija](#)):

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

s parametroma:

- $\mathbf{v}$ vektor, ki bo prirejen diagonalni,
- k diagonalna, kateri se priredi $\mathbf{v}$. $k=0$ uporabimo za glavno diagonalno, $k<0$ oz. $k>0$ uporabimo za diagonale pod oz. nad glavno diagonalno.

```
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]
```

```
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:

```
b = -2*g*h**2 * np.ones(n)
```

Sedaj *popravimo* matriko koeficientov **A** in vektor konstant **b**, da zadostimo robnim pogojem:

$$y_0 = 0, \quad y_{n-1} = 10. \quad (13.36)$$

```
A[0,0] = 1
A[0,1] = 0
A[-1, -2] = 0
A[-1, -1] = 1
```

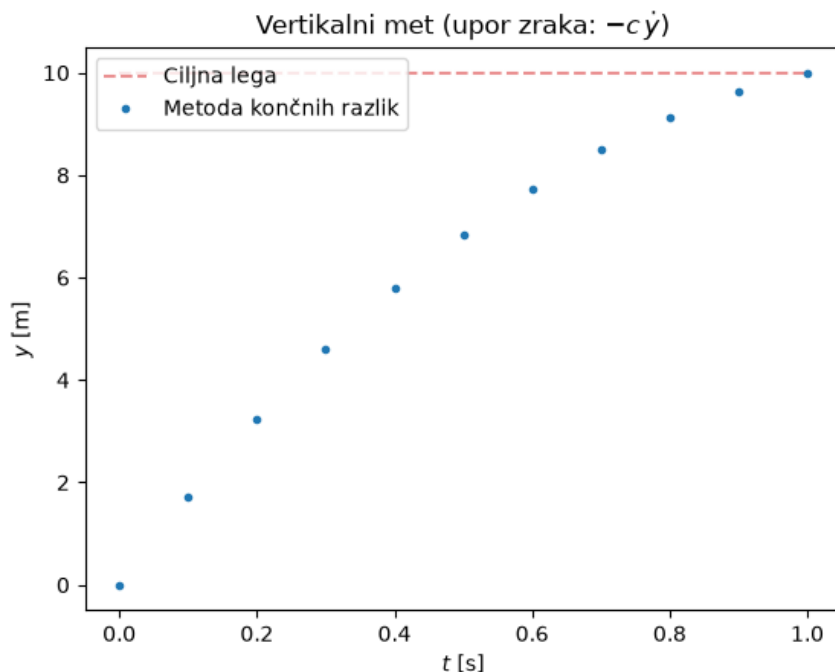
```
b[0] = 0.
b[-1] = 10.
```

Rešimo sistem linearnih enačb:

```
y_mkr = np.linalg.solve(A, b)
```

Prikažemo rezultat:

```
plt.title(r'Vertikalni met (upor zraka: $ -c\dot{y}$)')
plt.hlines(10, 0, 1, 'C3', label='Ciljna lega', linestyle='dashed', alpha=0.5)
plt.plot(t, y_mkr, '.', label='Metoda končnih razlik')
plt.xlabel('$t$ [s]')
plt.ylabel('$y$ [m]')
plt.legend()
plt.show()
```



S pomočjo funkcije `numpy.gradient()` izračunamo še hitrost in pospešek:

```
v_mkr = np.gradient(y_mkr,h, edge_order=2)
a_mkr = np.gradient(v_mkr,h, edge_order=2)
v_mkr[:4]

array([17.99109496, 16.20009044, 14.45276895, 12.79068266])
```

Izračunajmo sedaj še rezultat z dvojnimi koraki:

```
n2h = n//2+1
t2h = np.linspace(0, 1, n2h)
h2h = t2h[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)
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:

```
y2h_mkr[:6]

array([ 0.          ,  3.24082594,  5.7992875 ,  7.74037178,  9.12287659,
        10.          ])
```

z vsakim drugim rezultatom pri koraku h:

```
y_mkr[:12:2]

array([ 0.          ,  3.24001809,  5.79815462,  7.73931207,  9.12221538,
        10.          ])
```

Sedaj lahko ocenimo napako:

```
(y_mkr[:,2] - y2h_mkr)/3

array([ 0.          , -0.00026928, -0.00037763, -0.00035324, -0.0002204 ,
        0.          ])
```

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):

$$-\frac{EI}{h^4} (1 w_{i-2} - 4 w_{i-1} + 6 w_i - 4 w_{i+1} + 1 w_{i+2}) - F_0 \sin(\pi x_i/l) = 0. \quad (13.37)$$

Robni pogoji so štirje, najprej poves na robovih:

$$w_0 = w_{n-1} = 0 \quad (13.38)$$

Ker na robovih ni momenta, je drugi odvod nič. S centralno diferenčno shemo torej zapišemo dodatne enačbe:

$$w_{-1} - 2 w_0 + w_1 = 0 \quad w_{n-2} - 2 w_{n-1} + w_n = 0. \quad (13.39)$$

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 :

```

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, n)), v prvi in v zadnji dve vrstici bi lahko zapisali tudi vrednosti 0, to bomo pozneje popravili):

```

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]

```

```

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):

```

b = np.zeros(n+2)
b[2:-2] = - h**4 * F_0*np.sin(np.pi*x[1:-1]/l)/(E*I)
b[:4]

```

```

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 $w_0 = w_{n-1} = 0$:

```

A[0,:3]= np.array([0, 1, 0])
b[0] = 0
A[-1,-3:]= np.array([0, 1, 0])
b[-1] = 0

```

Nato $w_{-1} - 2w_0 + w_1 = 0$ in $w_{n-2} - 2w_{n-1} + w_n = 0$:

```

A[1,:4] = np.array([1, -2, 1, 0])
b[1] = 0
A[-2,-4:] = np.array([0, 1, -2, 1])
b[-2] = 0

```

```

A[:5,:5]

```

```

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.]])

```

```

A[-5:,-5:]

```

```

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.]])

```

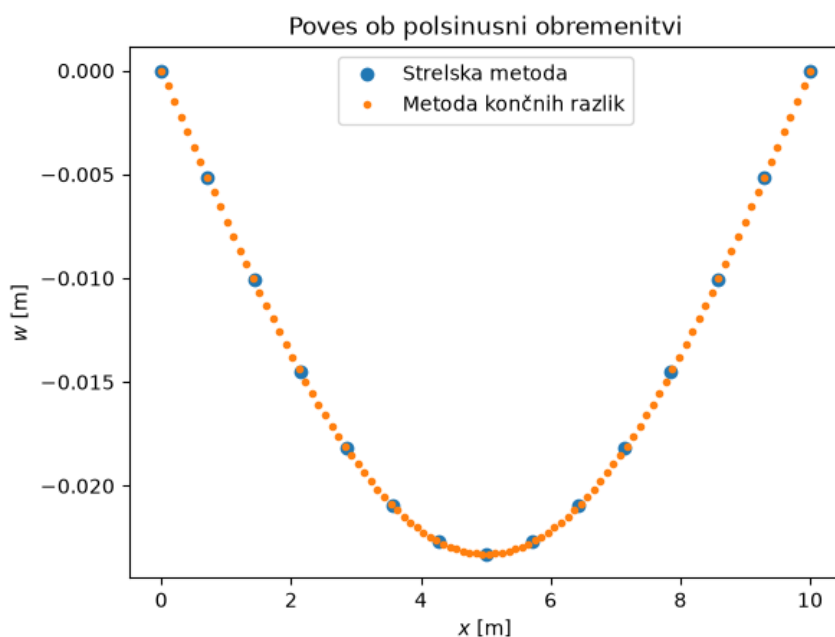
```
| b[:5]
|
| array([ 0.00000000e+00,  0.00000000e+00, -7.48966545e -10, -1.49717894e -09,
|        -2.24388381e -09])
|
| b[-5:]
|
| array([ -2.24388381e -09, -1.49717894e -09, -7.48966545e -10,  0.00000000e+00,
|        0.00000000e+00])
```

Rešimo sistem linearnih enačb:

```
| y_mkr = np.linalg.solve(A, b)
```

Prikažemo rezultat:

```
| plt.title('Poves ob polsinusni obremenitvi')
| plt.plot(sol_pol_sin.x, sol_pol_sin.y[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:

```
| import sympy as sym
| sym.init_printing()
```

Definirajmo spremenljivke:

```
w = sym.Function('w')
x, E, I, F_0, l = sym.symbols('x, E, I, F_0, l')
```

Definirajmo diferencialno enačbo (robne pogoje dodamo pozneje):

```
eq = sym.Eq( -E*I*w(x).diff(x,4) -F_0*sym.sin(sym.pi*x/l),0)
eq
```

$$-EI \frac{d^4}{dx^4} w(x) - F_0 \sin\left(\frac{\pi x}{l}\right) = 0 \quad (13.40)$$

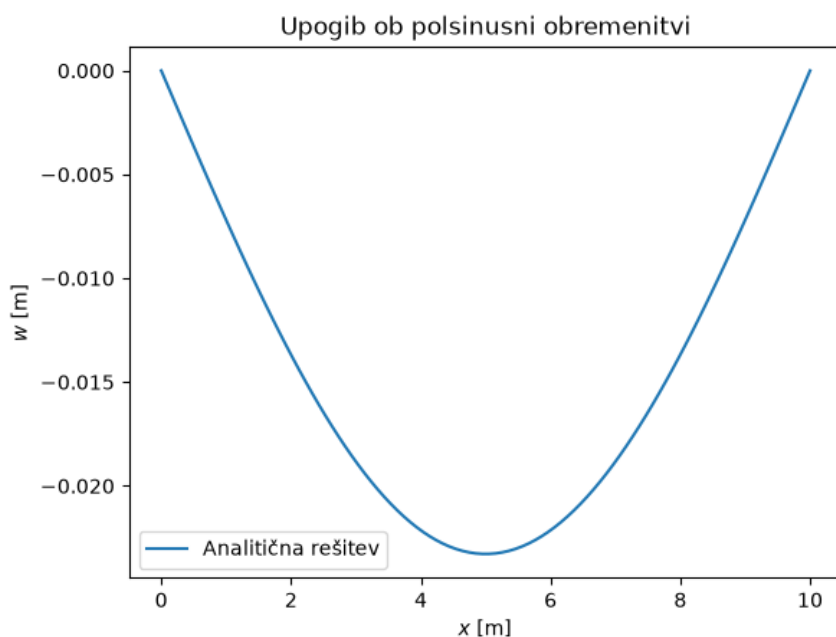
Rešimo:

```
sol = sym.dsolve(eq, func=w(x),
hint='nth_linear_constant_coeff_undetermined_coefficients',
ics={
    w(0):0, #poves pri x=0 je nič
    w(x).diff(x,2).subs(x, 0): 0, #moment pri x=0 je nič
    w(x).subs(x, l): 0, #poves pri x=l je nič
    w(x).diff(x, 2).subs(x, l): 0 #moment pri x=l je nič
})
sol
```

$$w(x) = -\frac{F_0 l^4 \sin\left(\frac{\pi x}{l}\right)}{\pi^4 EI} \quad (13.41)$$

```
podatki = {E: 2.1e11, I: 2.1e-5, l: 10, F_0:1e3}
resitev_np = sym.lambdify(x, sol.rhs.subs(podatki), modules='numpy')
x_ana = np.linspace(0, podatki[l], 100)
y_ana = resitev_np(x_ana)
```

```
plt.title('Upogib ob plosinusni 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 rešitvijo z metodo končnih razlik in streško metodo:

```
| [np.min(y_ana), np.min(y_mkr), np.min(sol_pol_sin.y[0])]
```

$$[-0.0232759411601314, -0.0232798479988575, -0.0232786462140205] \quad (13.42)$$

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](#) 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](#); če pogledate [vsebino](#) podmodula `numpy.linalg` za linearno algebro:

Branch: master ▾	numpy / numpy / linalg /
 eric-wieser MAINT: Remove unused isscalar import	
..	
 lapack_lite	DOC: Add changelog entry for new lapack_lite
 tests	TST: linalg: add basic smoketest for cholesky
 __init__.py	MAINT: Rearrange files in numpy/testing module.
 info.py	2to3: Apply `print` fixer.
 lapack_lite module.c	MAINT: Include the function name in all argument
 linalg.py	MAINT: Remove unused isscalar import
 setup.py	MAINT: Prefer to load functions from lapack/install

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 uporabljata sledeči:

- `unittest` (je vgrajen v Python),
- `pytest`.

Oba pristopa imata podobno funkcionalnost. Najpogosteje se uporablja `pytest` (npr. NumPy/SciPy), katerega osnove si bomo tukaj pogledali.

14.1.1 `pytest`

Modul `pytest` namestimo z ukazom `pip`.

Nekatere lastnosti:

- vrne opis testa, ki ni bil uspešen,
- samodejno iskanje testnih modulov in funkcij,
- lahko poganja tudi `unittest` 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](#)).

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

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](#)), ki nudi podporo za testiranje numeričnih polj.

Izbrane funkcije so:

- `assert_allclose(dejansko, pričakovano[, rtol, ...])` ([dokumentacija](#)) preveri enakost do zahtevane natančnosti,
- `assert_array_less(x, y[, err_msg, verbose])` ([dokumentacija](#)) preveri, ali so elementi `x` manjši od elementov `y`,
- `assert_string_equal(dejansko, pričakovano)` ([dokumentacija](#)) 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)
```

Če funkcija `orodja.zamenjaj_stolpca()` deluje pravilno, se klicanje `pytest` v ukazni vrstici uspešno zaključi. Za preostale teste pogledajte datoteko `test_orodja.py` in [dokumentacijo](#) paketa `pytest`.

14.2 Grafični uporabniški vmesnik

Uporabniški vmesnik se uporablja za interakcijo uporabnika s programsko kodo. Uporabniški vmesniki so lahko preko ukazne vrstice ali grafični. Za preproste uporabniške vmesnike preko ukazne vrstice je najbolje, da uporabimo kar [argparse](#), sicer pa se bomo tukaj predvsem osredotočili na **grafični uporabniški vmesnik**.

14.2.1 Grafični uporabniški vmesnik znotraj brskalnika

V zadnjem obdobju je opaziti veliko napora pri podpori razvoja grafičnih vmesnikov znotraj okolja *Jupyter notebook* ali *Jupyter lab*. Takšen pristop ima predvsem to prednost, da je nadgradnja v obliki spletne aplikacije relativno enostavna, glejte npr.: [Dashboarding with Jupyter Notebooks, Voila and Widgets | SciPy 2019 | M. Breddels and M. Renou](#).

14.2.2 Grafični uporabniški vmesnik znotraj operacijskega sistema

v okviru tega poglavja si bomo pogledali klasične uporabniške vmesnike, ki jih poganjamo v okviru določenega operacijskega sistema. Tudi za programiranje grafičnega uporabniškega vmesnika obstaja veliko različnih načinov/modulov. Nekaj najpogostejše uporabljenih:

- [PySide6](#) je uradna Python vezava (*Qt for Python*) za [Qt](#), ki predstavlja najbolj razširjeno platformo za uporabniške vmesnike; licenca [LGPL](#) dovoljuje tudi komercialno uporabo,
- [PyQt6](#) je starejša, neodvisna vezava za Qt s praktično enakim vmesnikom (razlikuje se le uvoz), vendar z licenco GPL (za komercialno uporabo je plačljiva),
- [Kivy](#) za hiter razvoj modernih uporabniških vmesnikov (ni tako zrel kakor npr. Qt),
- [wxWidgets](#) široko uporabljena prosta platforma za izdelavo uporabniških vmesnikov.

Med vsemi naštetimi si bomo podrobneje pogledali *PySide6*, ki je uradna in danes najbolj dozorela izbira. Namestimo ga z ukazom `pip install PySide6`. (Če spodnjo kodo želite uporabiti s *PyQt6*, zamenjajte le *PySide6* v uvozih s *PyQt6*.)

Velja omeniti, da uporabniški vmesnik lahko:

- *rišemo* s [QtDesignerjem](#) (glejte [video](#)), ali
- *kodiramo*.

Tukaj bomo uporabniški vmesnik *kodirali*.

14.2.3 Zgled

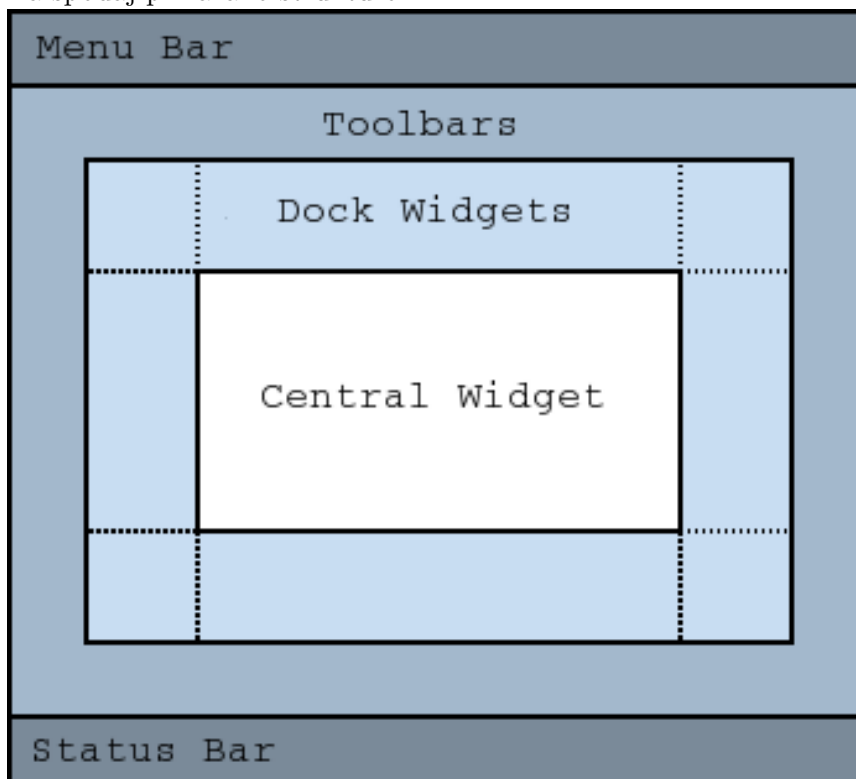
Najprej uvozimo paket za interakcijo in z grafičnimi objekti:

```
from PySide6 import QtCore      # za interakcijo
from PySide6 import QtWidgets  # grafični objekti
```

Potrebovali bomo tudi modul `sys` za poganjanje programa:

```
import sys
```

Uporabniški vmesnik tipično gradimo na razredu `QtWidgets.QMainWindow` ([dokumentacija](#)), ki ima spodaj prikazano strukturo:



Bistveni grafični elementi, ki jih pri takem uporabniškem vmesniku uporabimo, so:

- *Menu Bar*,
- *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:

- [Qt6 galeriji](#).

Zelo enostaven uporabniški vmesnik (z vrstičnimi komentarji kode) je prikazan spodaj.

```
import sys
from PySide6 import QtCore
from PySide6 import QtGui
from PySide6 import QtWidgets

class GlavniOkno(QtWidgets.QMainWindow):
```

```

""" Glavno okno podeduje `QtWidgets.QMainWindow`
"""

def __init__(self):
    """ Konstruktor GlavnOkno 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
    predpripravljenemu v QMainWindow

    # central widget
    self.gumb1 = QtWidgets.QPushButton('Gumb 1') # Gumb z napisom 'Gumb 1'
    self.gumb1.pressed.connect(self.akcija_pri_pritisku_gumbal) # povežemo
    pritisk s funkcijo za akcijo
    self.setCentralWidget(self.gumb1) # dodamo gumb v `CentralWidget`

    # menuji
    self.moj_izhod_akcija = QtGui.QAction('&Izhod',
                                          self, shortcut=QtGui.QKeySequence.Stand_
                                          ardKey.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 akcije `self.moj_izhod_akcija`

def akcija_pri_pritisku_gumbal(self):
    QtWidgets.QMessageBox.about(self,
                                'Naslov :)',
                                'Tole se sproži pri pritisku gumba.')

```

Program potem poženemo znotraj stavka try:

```

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())

```

Za naprednejši zgled glejte datoteki:

- [preprosti_uporabniski_vmesnik.py](#),
- [uporabniski_vmesnik.py](#)

Nekaj komentarjev na uporabniski_vmesnik.py:

1. Poglejte prepis dogodka `mouseDoubleClickEvent` in prepisite 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](#).

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 VisualStudioCode 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 `preprosti_uporabniski_vmesnik.py` inicializirski 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(...)` 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](#).

Poglavje 15

Literatura

1. Slavič J.: Programiranje in numerične metode s Pythonom, [izvršljiva knjiga](#) ali [spletna knjiga](#), 2015-
2. Petrišič J.: Uvod v Matlab za inženirje, 2011
3. Demšar J. Python za programerje, 2012 (glejte [spletno učilnico FRI](#))
4. Nicolas P. Rougier: From Python to Numpy, 2017 [spletna knjiga](#)
5. Kiusalaas J: Numerical Methods in Engineering with Python 3, 2013
6. Fangohr H. Python for Computational Science and Engineering, 2014 ([dosegljiva na spletu](#))
7. Bucky R. Python Programming Tutorials ([videi na YouTube](#))
8. [Projekt TOMO](#) (več vsebin, glejte predvsem predmete na FMF in FRI)
9. [Scientific Python Lectures](#) (prej *Scipy Lecture Notes*)
10. Uporaba paketov Numpy, Scipy itd za inženirje [Mr. P Solver \(YouTube\)](#)