Python za fizičare

Distribucija 2.0

Krešimir Kumerički

21 June, 2016

Sadržaj

1 Uvod **3**
- 1.1 Python vs. Jupyter vs. Sage . . . 3
- 1.2 Zašto Python? . . . 3

2 Sučelje **5**
- 2.1 Bilježnice i ćelije . . . 5
- 2.2 Elementarno računanje . . . 6
- 2.3 Help sustav . . . 9
- 2.4 Poruke o greškama . . . 9

3 Programiranje **11**
- 3.1 Liste i drugi spremnici . . . 11
- 3.2 Kontrola toka izvršavanja . . . 19
- 3.3 Funkcije . . . 20
- 3.4 Crtanje grafova . . . 21

4 Matematika **33**
- 4.1 Simbolički izrazi . . . 33
- 4.2 Jednadžbe . . . 37
- 4.3 Matematička analiza . . . 42
- 4.4 Linearna algebra . . . 45
- 4.5 Diferencijalne jednadžbe . . . 47
- 4.6 Statistika . . . 51
- 4.7 Prilagodba funkcije podacima . . . 54

5 Fizika **59**
- 5.1 Mehanika . . . 59
- 5.2 Elektrodinamika . . . 62
- 5.3 Termodinamika i statistička fizika . . . 66
- 5.4 Kvantna fizika . . . 67
- 5.5 Kozmologija . . . 70

6 Dodaci **77**
- 6.1 Testiranje hipoteze . . . 77
- 6.2 Funkcionalno programiranje . . . 81

Djelo *Python za fizičare*, čiji je autor Krešimir Kumerički, ustupljeno je pod međunarodnom licencom Creative Commons "Imenovanje-Dijeli pod istim uvjetima" (*Attribution-ShareAlike*) 4.0. Za uvid u tu licencu, posjetite http://creativecommons.org/licenses/by-sa/4.0/deed.hr.

POGLAVLJE 1

Uvod

1.1 Python vs. Jupyter vs. Sage

- Python je računalni jezik opće namjene. Zahvaljujući, s jedne strane, dobroj čitljivosti koda, a s druge brojnim kvalitetnim bibliotekama za numeriku, simboliku, crtanje itd., Python se često koristi u istraživanjima i edukaciji iz fizike i srodnih područja znanosti i tehnike.
- Jupyter je sučelje za Python (i brojne druge računalne jezike) namjenjeno interaktivnom radu. Postoje inačice za terminalski rad, grafičko sučelje i, najpopularnije, sučelje putem WWW preglednika tzv. *Jupyter notebook*.
- Anaconda je besplatna distribucija Pythona koja uključuje Jupyter i sve pakete koji se koriste u ovom dokumentu, pa je tako najjednostavniji put k instalaciji svega što nam je potrebno.
- Sage je besplatni matematički softver otvorenog koda koji udružuje niz postojećih matematičkih (i drugih) biblioteka u zajedničko sučelje zasnovano na Pythonu, s ciljem kreiranja alternative komercijalnim softverima poput Mathematice, Maplea ili Matlaba. Također omogućuje izvršavanje svog koda iz ovog dokumenta. Ne postoji verzija za Microsoft Windows, ali je na tom operacijskom sustavu moguća relativno bezbolna upotreba putem virtualnog Linuxa. SageMathCloud je besplatni WWW servis koji omogućuje online korištenje Sage ili Jupyter okruženja. Starija verzija ovog dokumenta koristila je Sage i dostupna je ovdje.

1.2 Zašto Python?

1. Python je izrazito elegantan za upotrebu. Popularan je u znanstvenoj zajednici, pa postoji velik broj korisnih Python biblioteka od općih pa do vrlo specijaliziranih znanstvenih namjena.
2. Znanost mora biti reproducibilna i u načelu vječna. Računalni kod koji ovisi o softveru "zatvorenog" koda to onemogućuje.
3. Sage je najrazvijeniji sustav za računalnu algebru (CAS - *Compute Algebra System*) otvorenog koda (*open source*). Ukoliko zahtjevi korisnika za složenijim matematičkim i simboličkim strukturama nisu preveliki, čisti Python u kombinaciji s nekoliko važnih dodatnih Python paketa (numpy, scipy, matplotlib, sympy) predstavlja dovoljnu, a značajno univerzalniju i portabilniju alternativu Sage-u.

Za detaljniju diskusiju o izboru softvera za računanje u znanosti, vidi Johansson, Introduction to scientific computing with Python, te, također, Koepke, 10 Reasons Python Rocks for Research (And a Few Reasons it Doesn't).

POGLAVLJE 2

Sučelje

2.1 Bilježnice i ćelije

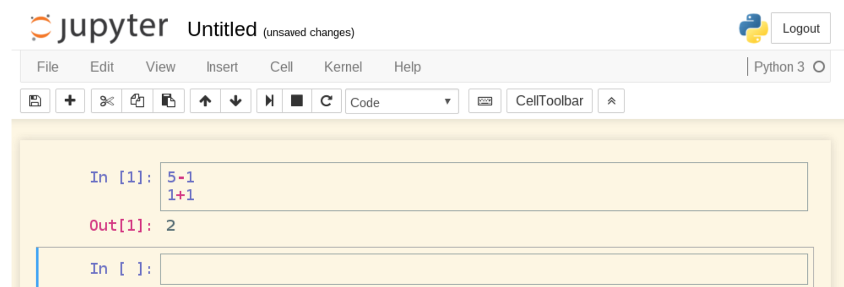
Bilježnica (*notebook*) je skup računalnog kôda s rezultatima i tekstualnim opisom koji je

- prikazan na jednoj stranici u WWW pregledniku
- može se pohraniti u jednu datoteku (ekstenzija `.ipynb`)

Račun, tj. računalni kôd radnog lista je organiziran u tzv. ćelije (*cell*). Svaka računska (*input*) ćelija je ograđena pravokutnikom i izvršava se *kao cjelina* na nekoliko načina:

1. pritiskom na kombinaciju tipki `shift-Enter` dok je kursor bilo gdje u ćeliji (Pritisak samo na `Enter` otvara novi redak u ćeliji),
2. pritiskom na kombinaciju tipki `alt-Enter`. To onda dodatno i otvara novu ćeliju.
3. klikom mišem na gumb s ikonom `Play`
4. klikom mišem na izbornik `Cell` pa `Run Cells`

Rezultat se nakon izvršavanja ispisuje neposredno ispod ćelije. Ako se eksplicitno ne zatraži drugačije (npr. naredbom `print`), bit će ispisan samo rezultat zadnjeg računa/komande/retka u toj ćeliji (ali izvršit će se naravno cijela ćelija).



Otvaranje novih ćelija se izvodi putem ikone `Plus` ili izbornika `Insert`. Nove ćelije su po defaultu izvršne (*Code*) tj. sadrže računalni kôd. Ukoliko želimo u bilježnicu dodati slobodni tekst i komentare, ćeliju prvo pretvaramo u tekstualnu putem izbornika na traci s alatima (`Code` pretvorimo u `Markdown` ili `Heading` ukoliko želimo staviti naslov odjeljka) ili putem tipke `M`.

Zadatak 1

Kreirajte novu input ćeliju i izračunajte `2+3`. Kreirajte zatim tekstualnu ćeliju s nekim tekstom iznad ove. Izbrišite obje ćeije.

Važno svojstvo Jupyter sučelja je da svaku ćeliju možemo i naknadno editirati i onda ponovno izvršiti. To je onda sve skupa sličnije radu u tabličnom kalkulatoru (*spreadsheet*) nego standardnom programiranju. Osim samog sadržaja ćelija bilježnice, trenutno stanje je određeno i stanjem python jezgre (*kernela*), a ono općenito ovisi o redosljedu kojim su ćelije izvršene. Resetiranje tog stanja se izvodi putem izbornika `Kernel` pa `Restart`. (Što je korisna operacija u trenucima kad se Jupyter počne neočekivano ponašati.) Zbog toga je po završetku rada dobro prekontrolirati konzistentnost i reproducibilnost cijelog računa tako da resetiramo jezgru i izvršimo sve ćelije odjednom pomoću izbornikai `Kernel` pa `Restart & Run All`.

Svakoj bilježnici pripada nezavisni kernel proces s nezavisnim varijablama. Ovisno o instalaciji Jupytera, osim Python jezgri na raspolaganju mogu biti i jezgre drugih programskih jezika.

2.2 Elementarno računanje

Python se može koristiti kao obični kalkulator proizvoljne preciznosti:

```
>>> 3*2+1
7

>>> 13**23
41753905413413116367045797

>>> 13.**23
4.175390541341312e+25
```

Uočite različit tretman cijelih (*integer*) i brojeva s pomičnom točkom (*floating point*). Cijeli brojevi se tretiraju egzaktno, bez zaokruživanja ili odbacivanja nekih znamenaka.

Napomena: Pythonov prompt znak >>> u gornjim primjerima nije potrebno unositi u Jupyter ćelije (makar ne smeta). On se u ovom tekstu pojavljuje samo zato da se omogući automatsko testiranje prikazanog koda.

Napomena: Jupyter automatski ispisuje samo rezultat posljednje komande u datoj ćeliji. Ukoliko trebamo i ispis nekih unutarnjih komandi, treba upotrijebiti funkciju `print`.

```
print(2+3)
2-3   # Zadnja linija pa je print ovdje nepotreban
```

```
5
-1
```

Za zapis velikih brojeva može se koristiti standardni Fortran/C zapis po kojem se npr. $3.2 \cdot 10^4$ zapisuje ovako

```
>>> 3.2e4
32000.0
```

Mnoge standardne matematičke funkcije mogu se pozivati tek nakon uključenja dodatnog Python paketa

```
>>> from scipy import *
>>> sqrt(9)
3.0

>>> sin(radians(90))
1.0
```

Umjesto `scipy` paketa mogli bi koristiti i pakete `math` ili `cmath`, (koji su manji i brži i također sadrže osnovne matematičke funkcije), ili paket `numpy` koji je srednje rješenje, vidi tablicu, ali `scipy` pruža unificiraniji pristup kompleksnim i realnim brojevima, a koristiti ćemo ga ionako i za brojne druge svrhe. U slijedećoj tablici uspoređujemo ove pakete. Redak *brzina* je vrijeme u mikrosekundama za izvrijednjavanje logaritma sto slučajnih brojeva, a redak *distributivnost* označava da li funkcije mogu primiti liste brojeva kao argumente (pa se automatski distribuiraju po elementima).

Paket	math	numpy	scipy
brzina	37.4	89.1	968
distributivnost	NE	DA	DA
sqrt(-1)	ValueError	nan	1j
log(-1)	ValueError	TypeError	3.14159j

Kod učitavanja funkcija na način kao gore (`from X import *`) treba paziti da ne dođe do kolizije istoimenih funkcija iz različitih paketa. Pravilnije je učitavanje i korištenje modula na slijedeći način

```
>>> import scipy.special
>>> scipy.special.gamma(6)    # = (6-1)!
120.0
```

(Eulerova gama funkcija je poopćeni faktorijel.)

Paketu `numpy` se tradicionalno pri učitavanju skraćuje ime.

```
>>> import numpy as np
>>> np.random.random()
0.3745401188473625
```

(Tako se dobiva slučajni broj između 0 i 1.)

Pridruživanje vrijednosti varijablama izvodi se znakom jednakosti "=".

```
>>> x = 4
>>> x
4

>>> x + 3
7
```

Primijetite da sama operacija pridruživanja ne rezultira nikakvim ispisom. Pridruživanje i naknadni ispis se mogu napraviti i u jednom redu korištenjem simbola `;` koji odvaja naredbe:

```
>>> x = 4; x
4
```

Standardne matematičke funkcije i konstante imaju uobičajena imena i ponašanje:

```
>>> log(e)
1.0

>>> sin(pi)
1.2246467991473532e-16
```

Uočite da su ovi objekti *floating point* tipa i zadržavaju preciznost od oko 14-15 značajnih znamenki. Ukoliko trebamo veću preciznost možemo koristiti paket `mpmath`, a ukoliko želimo raditi s "apsolutno" točnim simboličkim objektima, koristimo paket *SymPy* ili Sage okruženje.

```
>>> import mpmath as mp
>>> mp.mp.dps = 30
>>> mp.sin(mp.pi)
mpf('1.69568553207377992879174029387737e-31')
```

S kompleksnim brojevima radimo jednostavno. Samo treba imati na umu da se imaginarni broj konstruira dodavanjem znaka `j` pa je tako imaginarna jedinica `1j`.

```
>>> 1j**2
(-1+0j)

>>> sqrt(-4)
2j
```

Zadatak 2

Izračunajte $\sqrt{2\sqrt{e^{\pi}}}$.

Zadatak 3

Uvjerite se numerički da za proizvoljni realni x vrijedi

$$\sin(ix) = i\sinh(x).$$

Zadatak 4

Izvrijednite Eulerovu gama funkciju za z=1/2, dakle $\Gamma(1/2)$. Uvjerite se da je rezultat jednak $\sqrt{\pi}$.

Za upis matematičkih izraza u tekstualne čelije stavljamo LaTeX kôd unutar dolarskih znakova. Tako se upis

```
$\alpha$
```

unutar retka prikazuje kao α. Veće jednadžbe koje trebaju stajati u posebnom redu upisuju se između parova dolarskih znakova u novom redu. Tako se:

```
$$ E = \gamma m c^2 $$
```

prikazuje kao

$$E = \gamma m c^2$$

Za precizno formatiranje ispisa rezultata računa koristimo pretvorbu brojeva u stringove te standardno Python formatiranje. Npr. slijedeći kod formatira ispis broja π na 7 mjesta ukupno i 4 mjesta iza decimalne točke.

```
>>> print('{} = {:7.4f}'.format('Ludolfov broj', pi))
Ludolfov broj =  3.1416
```

2.3 Help sustav

Da bismo pronašli potrebnu funkciju te način i primjere njene upotrebe služimo se slijedećim pristupima Python dokumentaciji. Kao prvo, tu je `TAB-nastavljanje` (*TAB-completion*): započnemo li pisati ime neke funkcije, pritisak na tipku TAB dovršava pisanje njenog imena ako je nastavak jedinstven, a ako nije dobivamo popis svih mogućnosti (pa željenu odaberemo mišem ili kursorskim tipkama i tipkom `Enter`).

Dokumentaciju konkretne funkcije dobijemo tako da nakon imena funkcije stavimo upitnik i onda pritisnemo `ctrl-ENTER`. Iz dobivene dokumentacije možemo *cut-and-paste*-ati primjere u input ćelije. (Ukoliko umjesto jednog stavimo dva upitnika dobijemo cijeli ispis kôda koji definira tu funkciju.)

Za pretraživanje dokumentacije najefikasnije je koristiti Google.

2.4 Poruke o greškama

Ako se ogriješimo o matematička ili sintaktička pravila Python će nam uzvratiti porukom o grešci. Klik mišem lijevo od vrha te poruke daje opširnije informacije (inače, drugi klik potpuno skriva poruku što se može koristiti i za skrivanje svih nepregledno dugačkih ispisa rezultata računa.) Za interpretaciju opširnije poruke potrebno je znanje Python programskog jezika, no ključna informacija je obično u zadnjem retku, koji je odmah vidljiv.

```
>>> 1/0
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
```

```
>>> sin[2.3]
Traceback (most recent call last):
  ...
TypeError: 'numpy.ufunc' object is not subscriptable
```

Početniku će te poruke izgledati nejasno, ali s vremenom će poprimati sve više smisla i treba ih uvijek čitati. Npr. gornja greška *"object is unsubscriptable"* je posljedica toga što smo za poziv funkcije `sin` umjesto okruglih zagrada upotrijebili uglate, koje služe za pristup elementima (tj. indeksima, *subskriptima*) polja i matrica.

POGLAVLJE 3

Programiranje

3.1 Liste i drugi spremnici

3.1.1 Liste

Liste su središnji objekti programiranja u Pythonu. Srodne su poljima (*array*) iz standardnih programskih jezika (C, Fortran), ali imaju bitno više svojstava. Elementi lista mogu biti praktički bilo koji objekti.

```
>>> t1 = [1, 3, 5, 7, 9, 11]; t1
[1, 3, 5, 7, 9, 11]

>>> t2 = [[], ["jedan"], t1]; t2
[[], ['jedan'], [1, 3, 5, 7, 9, 11]]
```

Dakle, elementi liste u načelu mogu biti različitog tipa, kao u slučaju gornje liste `t2`, ali to je rijetko korisno.

Liste se između ostalog mogu zbrajati, množiti i može im se mjeriti duljina:

```
>>> t1+t2
[1, 3, 5, 7, 9, 11, [], ['jedan'], [1, 3, 5, 7, 9, 11]]

>>> 7*[3]
[3, 3, 3, 3, 3, 3, 3]

>>> len(t1)
6
```

Kako je Python objektno orijentirani računalni jezik, mnoge operacije se izvode putem tzv. *metoda* objekta, a to su objektu pridružene funkcije koje se pozivaju sintaksom `objekt.metoda()`. I liste su objekti pa imaju niz metoda (vidi popis), od kojih je daleko najvažnija metoda `append()` koja služi za dodavanje elemenata na kraj liste:

```
>>> t1.append(13); t1
[1, 3, 5, 7, 9, 11, 13]
```

Pojedinim elementima liste se pristupa pomoću indeksiranja, gdje prvi element liste ima indeks 0, drugi ima indeks 1 itd. Pojedine elemente možemo promijeniti običnim pridruživanjem pomoću znaka jednakosti:

```
>>> t1[1] = "tri"
>>> t1
[1, 'tri', 5, 7, 9, 11, 13]
```

Ukoliko želimo pristupiti većem broju elemenata od koristi su tzv. *rezovi* liste (engl. *slice*). Općenita sintaksa reza je `lista[početak:kraj:korak]`, gdje je `početak` uključen u rez, a `kraj` nije:

```
>>> t1[2:6]
[5, 7, 9, 11]

>>> t1[2:6:2]
[5, 9]
```

Ukoliko rez ide od samog početka ili sasvim do kraja liste, odgovarajuće indekse možemo izostaviti:

```
>>> t1[2:]
[5, 7, 9, 11, 13]
```

Negativni indeksi nam omogućuju da brojimo od kraja liste ...

```
>>> t1[-2:]    # zadnja dva elementa
[11, 13]
```

... a negativan korak da rez ide unatrag tj. da se izvrne redosljed elemenata:

```
>>> t1[::-1]  # lista natraške
[13, 11, 9, 7, 5, 'tri', 1]
```

Zadatak 1

Promijenite u gornjoj listi t1 element `"tri"` (ne element s indeksom=3!) natrag u broj `3`, ali ne eksplicitnom upotrebom indeksa `1`, već tako da "pronađete" indeks elementa `"tri"` pomoću metode `index()`. (Nije dozvoljeno upotrijebiti znamenku `1`).

Za konstrukciju liste cijelih brojeva pogodna je konstrukcija `list(range(početak, kraj, korak))` gdje treba imati na umu da će konstruirana lista početi s elementom `početak`, ali će završiti s elementom `kraj-1`.

```
>>> list(range(3))   # rezultira listom od tri elementa
[0, 1, 2]
```

`range()` je *niz*, što je poseban tip Python objekta, a `list` je funkcija koja konvertira taj niz u listu. Za konstrukciju liste *realnih* brojeva, vidi dolje odjeljak *NumPy polja*.

3.1.2 Obuhvaćanje liste

U praksi je kirurško mijenjanje pojedinih elemenata liste, kao u gornjem zadatku, vrlo rijetko. Listama se u pravilu rukuje kao cjelinama i najčešće se djeluje na sve njihove elemente. U standardnim računalnim jezicima u tu svrhu se koriste petlje, ali u Pythonu je najelegantnije koristiti tzv. *obuhvaćanje liste* (engl. *list comprehension*):

```
>>> t3 = [1, 3, 7, 11]
>>> [n**2 for n in t3]
[1, 9, 49, 121]
```

Obuhvaćanjem `range()`-a možemo konstruirati najrazličitije liste:

```
>>> from scipy import *
>>> pi
3.141592653589793

>>> [round(pi, k) for k in range(6)]
[3.0, 3.1, 3.14, 3.142, 3.1416, 3.14159]
```

```
>>> import numpy as np
>>> [np.random.random() for k in range(4)]
[0.3745401188473625, 0.9507143064099162, 0.7319939418114051, 0.5986584841970366]
```

Primijetite da varijablu "petlje" `k` ne moramo eksplicitno koristiti pri konstrukciji elemenata liste.

Često se javlja potreba za izborom elemenata liste koji zadovoljavaju neki kriterij. To se može izvesti obuhvaćanjem liste uz dodatni uvijet:

```
>>> t4 = range(1, 11)
>>> [n for n in t4 if (n % 2) == 0]     # izbor parnih elemenata
[2, 4, 6, 8, 10]
```

(Kao i u jeziku C, operator `%` daje ostatak cjelobrojnog dijeljenja.)

Napomena: Mjerenje vremena potrebnog da se izvrši neka ćelija izvodi se stavljanjem `%%time` u prvi red, a prekid računa koji traje predugo izvodi se putem izbornika `Kernel` pa `Interrupt` ili pritiskom na ikonu `Stop` u traci s alatima.

`%%time` naredba ne pripada jeziku Python već je to tzv. "magic" naredba koja modificira ponašanje IPython/Jupyter ćelije.

```
>>> import sympy
>>> %time sympy.factorint(1627222223994444443322222221)
CPU times: user 2.73 s, sys: 0 ns, total: 2.73 s
Wall time: 2.73 s

{10100000011: 1, 161111111111111111: 1}
```

Zadatak 2

Koristeći funkciju `sympy.isprime` i obuhvaćanje liste, konstruirajte listu svih prim-brojeva manjih od 100. Koliko ima prim-brojeva manjih od $10^4, 10^5, 10^6, \ldots$? Usporedite rezultate (i brzinu njegovog dobivanja) s funkcijom sympy.primepi().

Zadatak 3

Izračunajte funkciju $\pi(x)$ (broj prim-brojeva manjih od x) za $x = 10^{10}$ statističkom metodom: Izaberite uzorak od n slučajnih cijelih brojeva između 1 i x i testirajte koliko ima prim-brojeva među njima. Iz tog udjela odredite $\pi(x)$. Kolika veličina uzorka vam treba da bi relativna greška prema $\pi(10^{10}) = 455052511$ bila otprilike 1% ili manje?

Zadatak 4

Kreirajte listu od 100 slučajnih brojeva iz intervala `[0, 10)`, a zatim u toj listi ostavite samo brojeve koji se za manje od `0.02` razlikuju od cijelog broja. *Naputak*: Testirajte `abs(x-round(x))`.

3.1.3 Ostali spremnici

Osim lista, postoje i drugi spremnici (*containers*) za objekte. Kao prvo tu su tzv *tuplovi* (*tuple*) koji "izvana" izgledaju isto kao liste samo u okruglim zagradama.

```
>>> tpl = (1, 3, 5, 7)
>>> tpl[3]
7
```

Glavna razlika prema listama je da su tuplovi nepromjenjivi. Nije moguće niti promijeniti neki njihov element niti im dodati nove:

```
>>> tpl[3] = 2
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

Razlika u upotrebi između tuplova i lista je da su tuplovi obično heterogeni strukturirani skupovi u kojima pojedina mjesta u tuplu nose različita značenja, dok su liste obično homogeni skupovi istovrsnih objekata. Npr, koordinate neke točke je prirodno staviti u tupl (x, y), a ne u listu [x,y], ali niz točaka je prirodno staviti u listu takvih tuplova [(x1, y1), (x2, y2), ...].

Kad varijablama pridružujemo vrijednosti iz kraćih lista ili tuplova zgodno je koristiti tehniku *raspakiravanja*

```
>>> x, y, z = (1, 2, 3)
>>> print("Norma vektora  = {}".format(sqrt(x**2 + y**2 + z**2)))
Norma vektora  = 3.7416573867739413
```

Raspakiravanje se često koristi kod procesiranja listi čiji su elementi liste ili tuplovi. U slijedećem primjeru pretvaramo listu 2D kartezijevih vektora u listu njihovih normi. Vidimo kako možemo kombinirati raspakiravanje po unutarnjoj s obuhvaćanjem 2D liste po vanjskoj dimenziji:

```
>>> [sqrt(x**2 + y**2) for x,y in [(1,2), (3,4), (5,6), (7,8)]]
[2.23606797749979, 5.0, 7.810249675906654, 10.63014581273465]
```

Zadatak 5

Koristeći raspakiravanje tuplova u kombinaciji s obuhvaćanjem liste pretvorite ovu listu vektora

$$\mathrm{tt} = [(r_1, \theta_1), (r_2, \theta_2), \ldots]$$

```
>>> tt = [(2.23, 1.11), (5.0, 0.93), (7.81, 0.88), (10.63, 0.85)]
```

iz 2D polarnog sustava u kartezijev:

$$[(x_1, y_1), (x_2, y_2), \ldots]$$

Nakon toga transformirajte nastalu listu kartezijevih vektora u listu kartezijevih vektora s cjelobrojnim koordinatama, zaokruživanjem vrijednosti x i y koordinata pomoću funkcije `round()`.

Daljnji spremnici koji nam stoje na raspolaganju su *skupovi* (*set*), koji su skupovi različitih(!) objekata i s kojima možemo raditi standardne stvari poput unije, presjeka, komplementa ...

```
>>> s1 = set([10, 9, 10, 8, 7, 7, 7, 4]); s1
{8, 9, 10, 4, 7}
```

(Uočite da se duplikati automatski izbacuju.)

```
>>> s1.union(t1)
{1, 3, 4, 5, 7, 8, 9, 10, 11, 13}
>>> s1.intersection(t1)
{9, 7}
>>> s1.difference(t1)
{4, 8, 10}
```

Zadnji, vrlo važan, spremnik kojeg ćemo spomenuti je *rječnik* (engl. *dictionary*). Riječ je o preslikavanju `key` -> `value`, gdje je `value` bilo koji objekt, a `key` može biti tipično broj ili string (premda su i druge stvari dopustive kao key, npr tuplovi).

```
>>> d1 = {'a':1, 'c':3, 'b':2}
>>> d2 = {'ime': 'pero', 'prezime': 'peric', 'spol': 'M'}

>>> d1
{'b': 2, 'a': 1, 'c': 3}
```

Redosljed elemenata u rječniku nema značenja. Pristup pojedinim elementima rječnika je moguć "indeksiranjem" pomoću ključa:

```
>>> d1['c']
3
```

Na isti način je moguće dodavati nove elemente u rječnik:

```
>>> d2['telefon'] = '123456'
```

Iteriranje po rječniku ne daje elemente već samo ključeve

```
>>> [it for it in d1]
['b', 'c', 'a']
```

Dok za elemente treba koristiti metodu `values`

```
>>> [val for val in d1.values()]
[3, 2, 1]
```

A metoda `items` i tehnika raspakiravanja tuplova daje oboje

```
>>> fmt = '{:>8s} {:s} {:10s}'
>>> print(fmt.format('ključ', '|', 'vrijednost'))
>>> print(fmt.format('-----', '+', '----------'))
>>> aux = [print(fmt.format(key, '|', val)) for key,val in d2.items()]

   ključ | vrijednost
   ----- + ----------
    spol | M
 telefon | 123456
 prezime | peric
     ime | pero
```

Zadatak 6

Izračunajte prosjek godina ljudi iz slijedećeg rječnika (rezultat treba biti decimalni broj). Koristite funkciju `mean`. Vaš kod neka ne sadrži ni jedan broj.

```
>>> ages = {'john' : 76, 'paul': 73, 'george': 73, 'ringo': 75}
```

Za kraj, i *stringove* možemo interpretirati kao liste znakova i tretirati ih kao takve

```
>>> s1 = 'Samobor'
>>> s1[-3:]
'bor'
```

3.1.4 NumPy polja

Obične liste se rabe za razne namjene, uključujući simboličke račune, ali za numeriku su pogodnija tzv. NumPy polja. NumPy (*Numerical Python*) je modul za python namjenjen baratanju s multidimenzionalnim brojčanim poljima kakva se često sreću u svim područjima znanosti. Riječ je o velikom modulu koji nije automatski prisutan u Pythonu, već ga treba učitati pomoću naredbe `import`

```
>>> import numpy as np
```

(Gore smo već učitali taj modul. Višestruko učitavanje ne stvara probleme.)

Numpy polja su u računalu zapisana u neprekinutom slijedu memorijskih lokacija što omogućuje brži pristup no zbog toga svi elementi polja moraju biti istog tipa (`integer`, `float`, `complex` ...).

```
>>> a = np.array(range(1,11)); a    # konverzija obične liste u NumPy polje
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

Vidimo da je NumPy polje ispisano na drugačiji način nego obična lista, ali za razlikovanje tih dvaju objekata izgled ispisa nije pouzdani kriterij. Neke operacije, poput `print`, ispisuju NumPy polje da izgleda kao obična lista

```
>>> print(a)
[ 1  2  3  4  5  6  7  8  9 10]
```

Ako postoji dvojba, možemo ispitati tip objekta komandom `type()`

```
>>> type(a)
<class 'numpy.ndarray'>
```

Pri konstrukciji NumPy polja izbor tipa objekata je automatski, ali moguće ga je i odrediti opcionalnim argumentom dtype:

```
>>> b = np.array(range(1,11), dtype=np.float64); print(b)
[  1.   2.   3.   4.   5.   6.   7.   8.   9.  10.]
```

Tip objekata u NumPy polju je zapisan u `dtype` *atributu* polja [1].

```
>>> a.dtype
dtype('int64')
>>> b.dtype
dtype('float64')
```

Numpy polja imaju dosta više metoda od običnih lista ...

```
>>> dir(a)[-66:]
['argpartition', 'argsort', 'astype', 'base', 'byteswap', 'choose', 'clip', 'compress',
```

... no zbog efikasnosti implementacije nema upravo onih metoda koje imaju obične liste. To je zato jer je npr. umetanje elementa u listu vrlo “skupa” operacija koja uključuje brisanje i pisanje svih elemenata koji u memoriji dolaze nakon tog mjesta umetanja. Ako želimo raditi takve stvari upotreba NumPy polja nam ionako neće donijeti nikakve prednosti i treba koristiti obične liste. (Za konverziju NumPy liste u običnu koristimo funkciju `list()`.)

Jedno od vrlo korisnih svojstava NumPy lista je da se većina matematičkih operacija prirodno distribuira po elementima liste (obične liste nemaju to svojstvo):

```
>>> xs = np.linspace(1,10,3); xs
array([  1. ,   5.5,  10. ])
>>> xs + 1
array([  2. ,   6.5,  11. ])
>>> sin(xs)
array([ 0.84147098, -0.70554033, -0.54402111])
```

NumPy liste mogu biti i višedimenzionalne, no i one su u memoriji računala zapisane u jednodimenzionalnom (1D) slijedu memorijskih adresa. Stoga je preoblikovanje elemenata 1D NumPy u 2D polje proizvoljnog oblika računalno “jeftina” operacija, koja može biti od koristi

```
>>> b=a.reshape(2,5); b
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])
```

```
>>> b.shape   # "oblik" polja - broj redaka i stupaca
(2, 5)
```

Pristupanje pojedinom elementu višedimenzionalnog polja je moguće očitim indeksiranjem a[i][j]..., ali je efikasnije koristiti skraćeno indeksiranje: a[i, j, ...]:

```
>>> b[1,1] = 42.
```

[1] Atributi objekta su sve ono čemu se pristupa operatorom točkice `.`. Ranije spomenute *metode* su atributi koji se mogu pozivati kao funkcije. `dtype` je atribut koji je naprosto tip elementa polja. Za bolje razumijevanje svega ovog dobro je pročitati nešto o objektno-orijentiranom programiranju u Pythonu.

```
>>> print(b)
[[ 1  2  3  4  5]
 [ 6 42  8  9 10]]
```

Pažnja: radi efikasnosti, preoblikovanja i rezovi kroz NumPy polja ne kopiraju originalne elemente već samo manipuliraju pokazivačima na ista mjesta. Tako se npr. ova netom načinjena zamjena u listi `b` odražava i u listi `a` čijim preoblikovanjem je lista `b` nastala:

```
>>> print(a)
[ 1  2  3  4  5  6 42  8  9 10]
```

Obične liste nemaju takvo ponašanje:

```
>>> t5 = t1[2:4]; t5     # t1 je obična lista
[5, 7]
```

```
>>> t5[0] = 'novi'; t5
['novi', 7]
>>> t1
[1, 3, 5, 7, 9, 11, 13]
```

Inače, rezovi kroz dvodimenzionalne NumPy liste su elegantan način ekstrakcije redaka ili stupaca:

```
>>> b[:,1]
array([ 2, 42])
```

Ukoliko trebamo liste realnih (*float*) brojeva, možemo koristiti NumPy funkciju `arange()` (sintaksa je ista kao za `range()`)

```
>>> np.arange(2., 9.9, 1.1)
array([ 2. ,  3.1,  4.2,  5.3,  6.4,  7.5,  8.6,  9.7])
```

Funkcija `np.arange()` je pogodna kad želimo specificirati korak liste. Kad želimo specificirati broj elemenata liste koristimo funkciju `np.linspace()`.

```
>>> np.linspace(0, 10, 5)
array([  0. ,   2.5,   5. ,   7.5,  10. ])
```

Za daljnje detalje o NumPy poljima, vidi NumPy Tutorial.

Zadatak 7

Za spajanje dvije jednako duge 1D liste u 2D listu (zapravo listu tuplova) koristimo naredbu `zip`

```
>>> xs = [11, 12, 13, 14, 15]
>>> ys = [-1, -2, -3, -4, -5]
>>> points = list(zip(xs, ys)); points
[(11, -1), (12, -2), (13, -3), (14, -4), (15, -5)]
```

Rastavite listu `points` natrag na originalne dvije 1D liste i to dvjema različitim metodama:

1. Dva obuhvaćanja liste `points` (ova metoda nema veze s NumPy-em)
2. Dva reza kroz listu `points`, koju prvo pretvorite u NumPy polje pomoću `np.array`.

3.2 Kontrola toka izvršavanja

3.2.1 Grananje if-then

Sintaksa je

```
if <condition>:
    <block 1>
elif <condition>:
    <block 2>
else:
    <block 3>
```

Treba uočiti dvotočke prije blokova koda koji se uvjetno izvršavanju. Također je važno paziti na konzistentno uvlačenje blokova koda (tamo gdje bi u C-u bile vitičaste zagrade), jer je to jedina vodilja parseru da prepozna gdje su granice bloka. (Tradicionalno se uvlači za 4 mjesta. Editor u Sage ćelijama automatski radi to uvlačenje.) Npr:

```
>>> if 2>3:
...     print("veći je")
... elif 2==3:
...     print("jednak je")
... else:
...     print("manji je")
manji je
```

U uvjetima se mogu koristiti standardni logički operatori `and`, `or`, `not`, ..., a važno je uočiti da se test usporedbe jednakosti radi s dvostrukim znakom jednakosti ==. (Jednostruka jednakost = je rezervirana za operacije pridruživanja poput `x=1`.) Operatori uspoređivanja su standardni: `==`, `!=`, `<`, `<=`, `>`, `>=`.

```
>>> if 2>3 or 2<3:
...     print("nisu isti")
nisu isti
```

3.2.2 Petlje

`while` petlja ima standardni oblik

```
while <condition>:
    <body>
```

gdje je `condition` logički uvjet. Npr.

```
>>> k = 1
>>> while k < 10:
...     print(k, end='')
...     k += 1
123456789
```

Opcionalnim argumentom `end` sprečavamo automatski prelazak u novi red. (Defaultna vrijednost je `end='\n'` što je kontrolni znak za prelazak u novi red.) Mnogo Python funkcija ima opcionalne argumente koji se nužno zadaju nakon obaveznih argumenata. Ako ih ne zadamo, oni imaju svoje defaultne vrijednosti koje možemo saznati uvidom u dokumentaciju ili definiciju funkcije.

`For` petlja ima specifičnu sintaksu

```
for x in <iterable>:
    <body>
```

gdje je `<iterable>` lista, range, tuple, skup, rječnik, ... Za prijevremeno iskakanje iz petlje postoji komanda `break`. Npr. slijedeći algoritam pronalazi prvi cijeli broj čiji rastav sadrži više od pet različitih prostih faktora.

```
>>> import sympy
>>> for i in range(1,10**5):
...     if len(sympy.factorint(i))>5:
...         break
>>> print(i)
30030
```

Alternativni i nešto elegantniji algoritam bi bio

```
>>> i = 1
>>> while len(sympy.factorint(i))<6:
...     i +=1
>>> print(i)
30030
```

Zadatak 1

Isprogramirajte petlju koja će pomnožiti sve brojeve od 1 do 7 (dakle, izračunati će 7!). Riješite zadatak jednom koristeći `for` petlju, a drugi put `while` petlju.

3.3 Funkcije

Mogućnost definiranja novih funkcija je osnovna stepenica k naprednijem programiranju. U Pythonu funkcije se definiraju korištenjem ključnih riječi `def` i `return`, te blokova kôda koji su konzistentno uvučeni

```
>>> def h(x):
...     "Kvadriraj broj x."
...     return x**2

>>> print(h(3))
9
```

Kao prvi red tijela funkcije može se, kao gore, staviti dokumentacijski string (tzv. *docstring*) kojem se kasnije može pristupiti standardnim metodama pristupa dokumentaciji. (Dakle, `h?` će ispisati dokumentacijski string.)

Funkcija može imati i opcionalne argumente s defaultnom vrijednošću:

```
>>> def fun(x, n=1, b=0):
...     return x**n + b

>>> fun(3, 4, 5)
86
>>> fun(3, b=5, n=4)
86
```

```
>>> fun(3)
3
```

Uočite da zahvaljujući eksplicitnom imenovanju opcionalnih argumenata ne moramo paziti na njihov poredak.

Bilo što može biti argument funkcije. Najmoćnija stvar, obilato korištena u funkcionalnom pristupu programiranju, je da i same funkcije mogu biti argumenti funkcija:

```
>>> def twice(f, x):
...     "Komponiraj dvaput funkciju sa samom sobom"
...     return f(f(x))

>>> from scipy import *
>>> print(twice(sqrt, 16))
2.0
```

Zadatak 1

Isprogramirajte funkciju `fib(n)` koja računa n-ti član Fibonaccijevog niza (niz kod kojeg je svaki član definiran kao zbroj prethodna dva, a javlja se pri analizi idealizirane populacije zečeva). `fib(4) = 3`.

Zadatak 2

Isprogramirajte funkciju `fibBinet(n)` koja računa n-ti član Fibonaccijevog niza putem Binetove formule

$$F_n = \frac{\varphi^n - \cos(n\pi)\varphi^{-n}}{\sqrt{5}}$$

gdje je $\varphi = 1.618\ldots$ tzv. zlatni omjer

```
>>> from scipy.constants import golden_ratio
>>> golden_ratio
1.618033988749895
```

Uvjerite se da dobivate dobre vrijednosti za neke `n`.

3.4 Crtanje grafova

Glavna Python biblioteka za crtanje grafova je `matplotlib`. Ona se u Jupyter okruženju može elegantno koristiti tako da se na početku bilježnice pomoću *magic* komande

```
%matplotlib inline
```

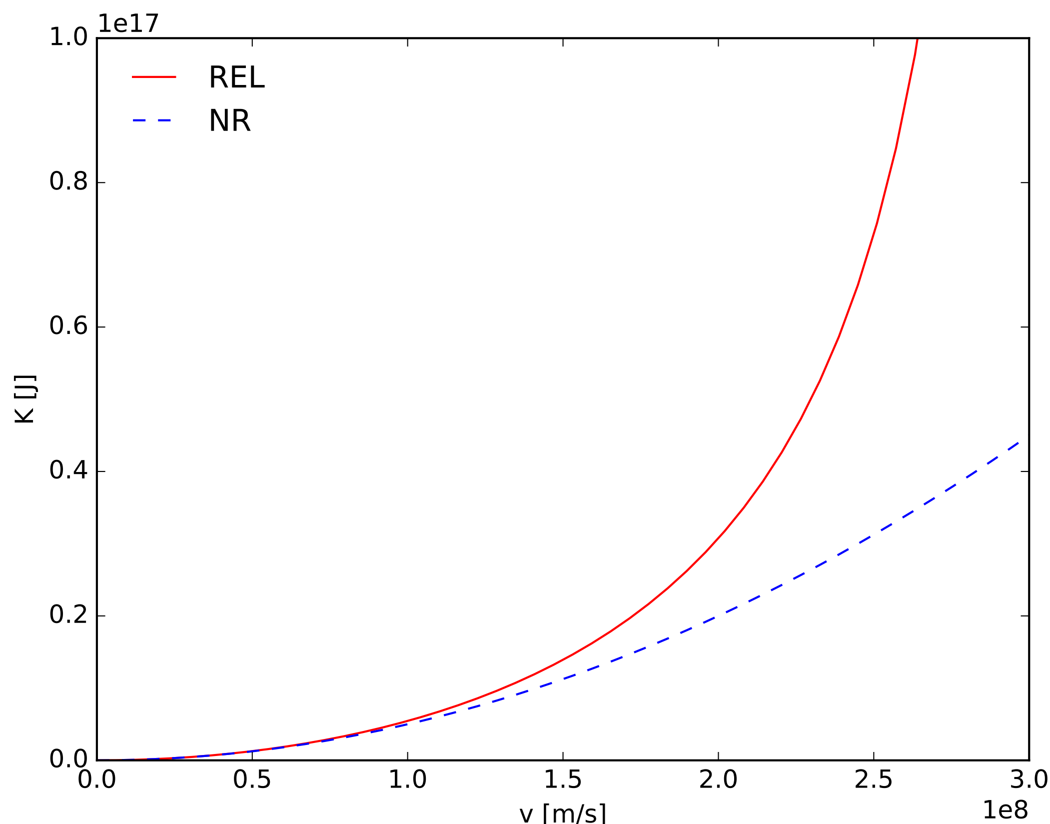
zatraži automatsko iscrtavanje slika ispod Jupyter ćelija. Zgodna alternativa je

```
%matplotlib notebook
```

koja rezultira time da iscrtane slike postanu interaktivne.

Matplotlib se onda koristi na slijedeći način. Prvo učitavamo potrebne pakete

```
from scipy import *
import numpy as np
import matplotlib.pyplot as plt
```

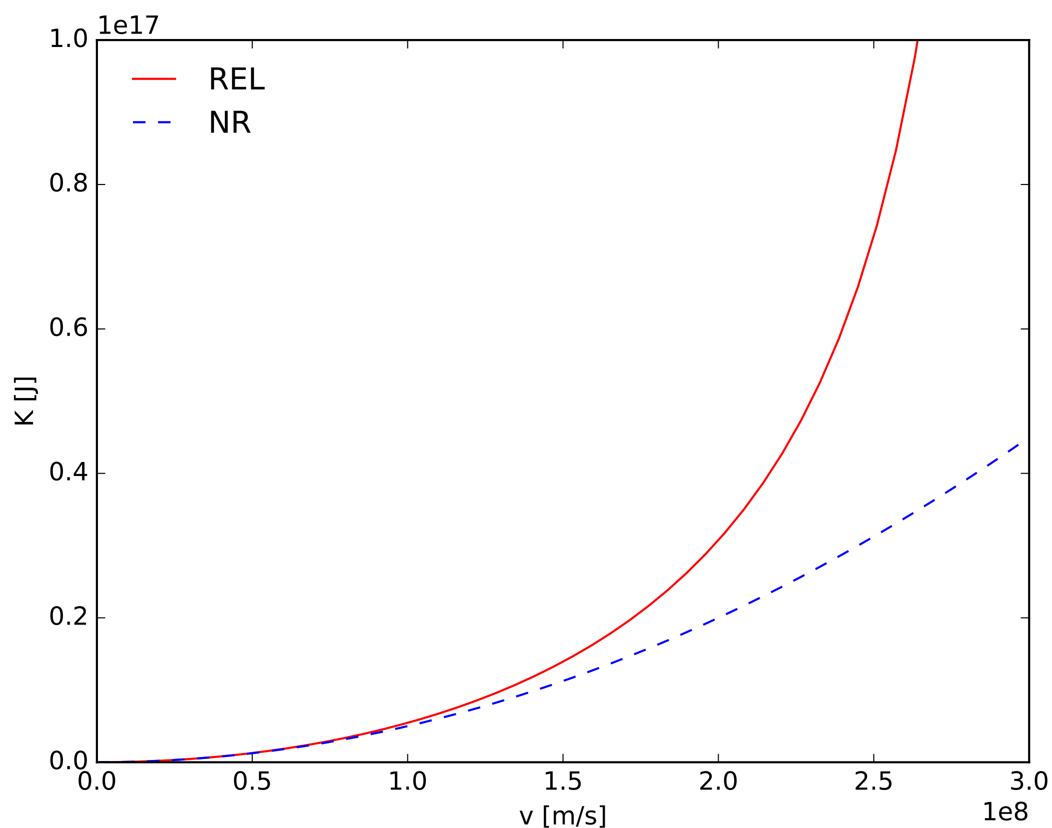


Za komunikaciju s Matplotlib bibliotekom koristimo modul `pyplot` (kojeg smo preimenovali u `plt`) koji omogućuje pristupačno sučelje [1].

Glavna funkcija za crtanje je `plt.plot` čiji svaki poziv rezultira obično jednom linijom grafa. Ta funkcija kao svoja prva dva argumenta traži liste `x` i `y` koordinata točaka koje definiraju liniju grafa. Točke će biti spojene ravnim crtama pa ih treba biti dovoljno da se dobiju glatke krivulje:

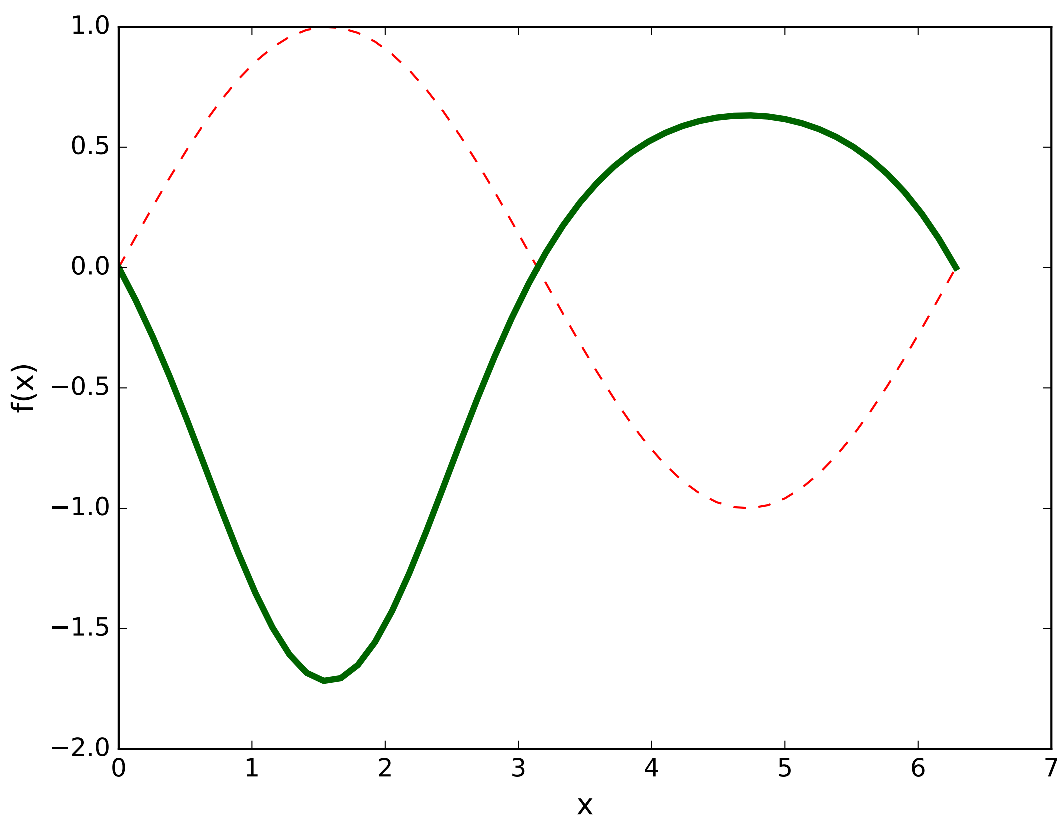
```
xs = np.linspace(0, 2*pi)
plt.plot(xs, sin(xs))
```

[1] Postoji i modul `pylab` sa sučeljem sasvim sličnim komercijalnom programu Matlab. Vidi usporedbu raznih Matplotlib sučelja.



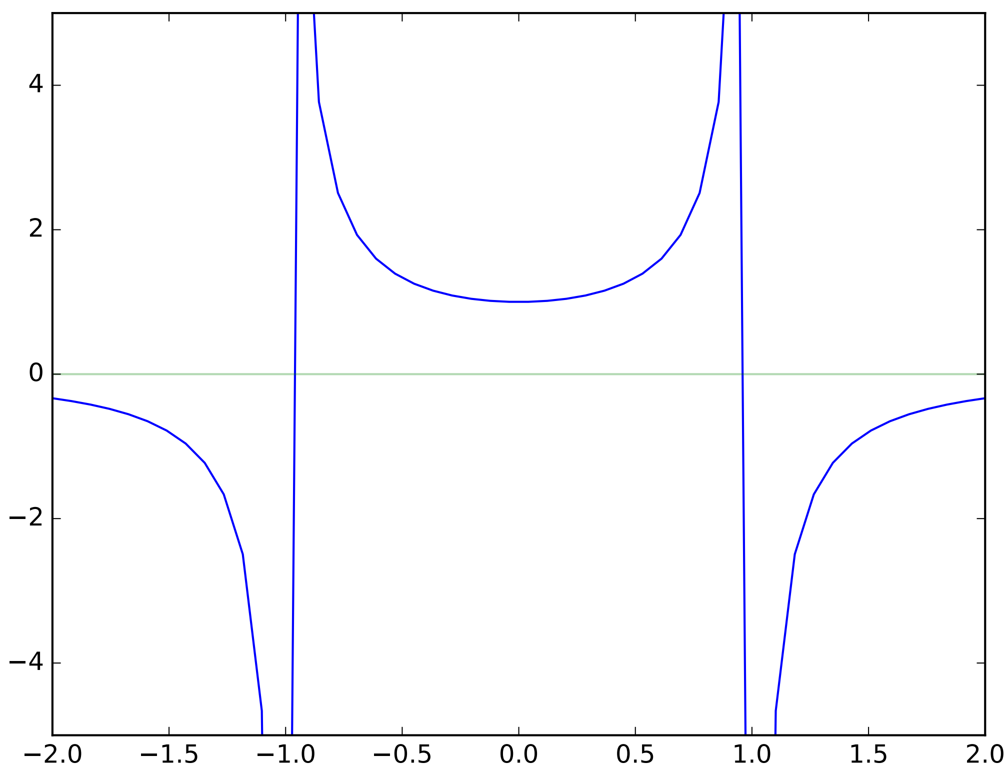
Svojstva linije kontroliramo opcionalnim argumentima funkcije `plot`, a za boju i stil postoji i skraćeni oblik:

```
plt.plot(xs, sin(xs), 'r--')
plt.plot(xs, 1-exp(sin(xs)), color='darkgreen',
                        linestyle='-', linewidth='3')
plt.xlabel('x', fontsize=14)
plt.ylabel('f(x)', fontsize=14)
```



Matplotlib sam određuje raspon vrijednosti ordinate pogodan za crtanje zadanih funkcija. Ponekad, npr. ukoliko funkcija ima singularitet u području crtanja, to može ispasti loše. No, uvijek možemo i sami specificirati raspon ordinate `ylim`:

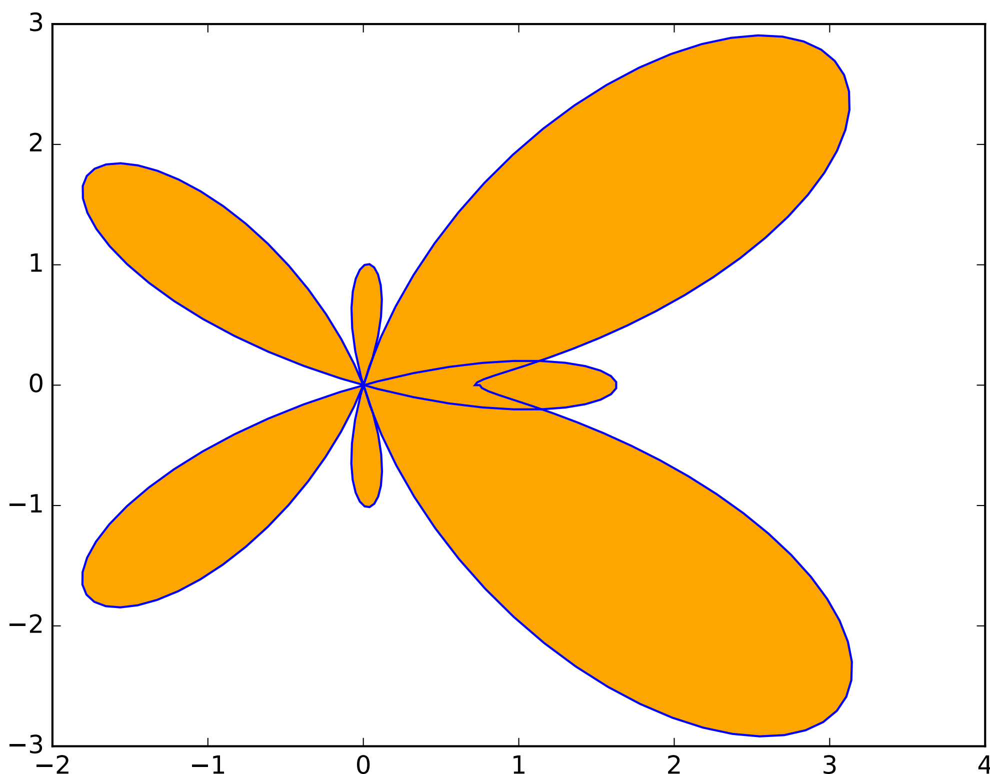
```
xs = np.linspace(-2, 2)
plt.plot(xs, 1/(1-xs**2))
plt.axhline(0, color='green', alpha=0.3) # transparent line at y=0
fig = plt.ylim(-5, 5)
```



Grafove često želimo upotrijebiti i zasebno, npr. u nekom članku ili prezentaciji. Izvoz grafova i drugih objekata postiže se uporabom funkcije `plt.savefig(<ime_fajla.ext>)`. Format grafičke datoteke određen je ekstenzijom. Neke od dopuštenih ekstenzija su `.png`, `.pdf`, `.ps`, `.eps`, `.svg`, `and` `.sobj`.

Osim funkcija eksplicitno zadanih u obliku $y = y(x)$ možemo crtati i funkcije zadane parametarski u obliku $y = y(t)$, $x = x(t)$. Npr:

```
ts = np.linspace(0, 2*pi, 200)
amps = [(exp(cos(t)) - 2*cos(4*t) + sin(t/12)**5) for t in ts]
plt.fill(amps*cos(ts), amps*sin(ts),
         facecolor='orange', edgecolor='blue')
```



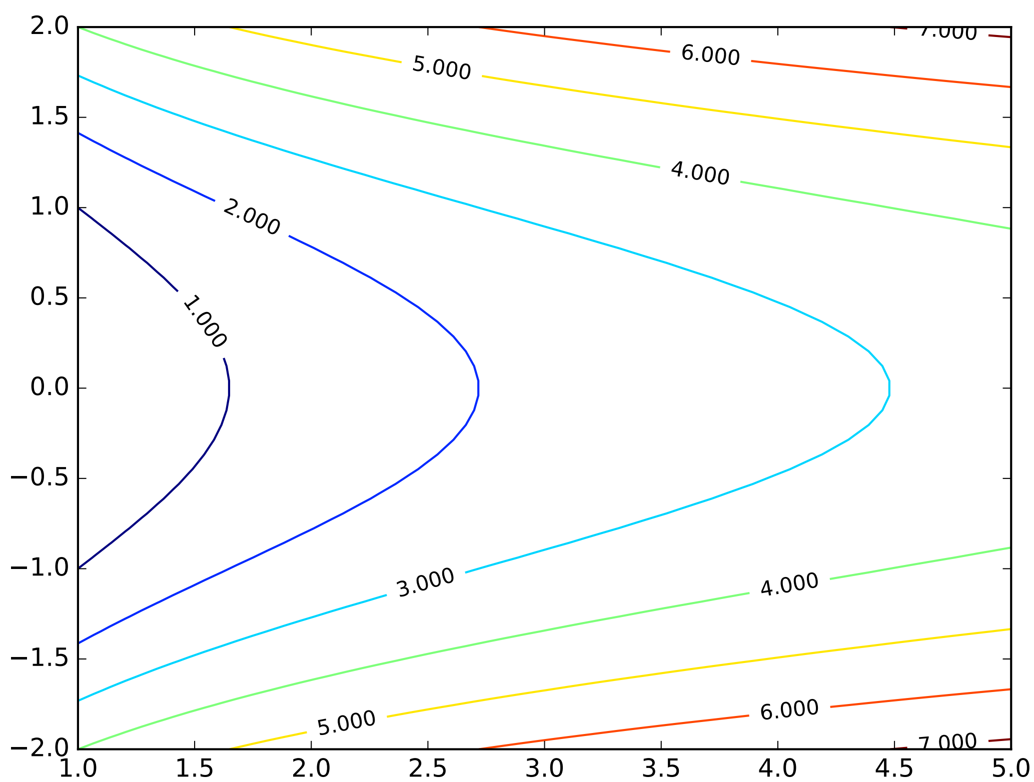
Za prikaz raznih potencijala i srodnih funkcija, korisna je funkcija `plt.contour`. Njena tri prva argumenta su tri dvodimenzionalna numpy polja koja po redu odgovaraju vrijednostima x koordinata točaka u ravnini, y koordinata te vrijednostima $f(x, y)$ funkcije koju prikazujemo. Npr. za prikaz funkcije $f(x, y) = \log(x^2) + y^2$, ta polja priređujemo na slijedeći način

```
>>> import numpy as np
>>> from scipy import *
>>> xs = np.linspace(1, 5, 60)
>>> ys = np.linspace(-2, 2)
>>> X, Y = np.meshgrid(xs, ys)
>>> Z = log(X**2) + Y**2
```

```
>>> xs.shape
(60,)
>>> ys.shape
(50,)
>>> X.shape
(50, 60)
>>> X.shape == Y.shape == Z.shape
True
```

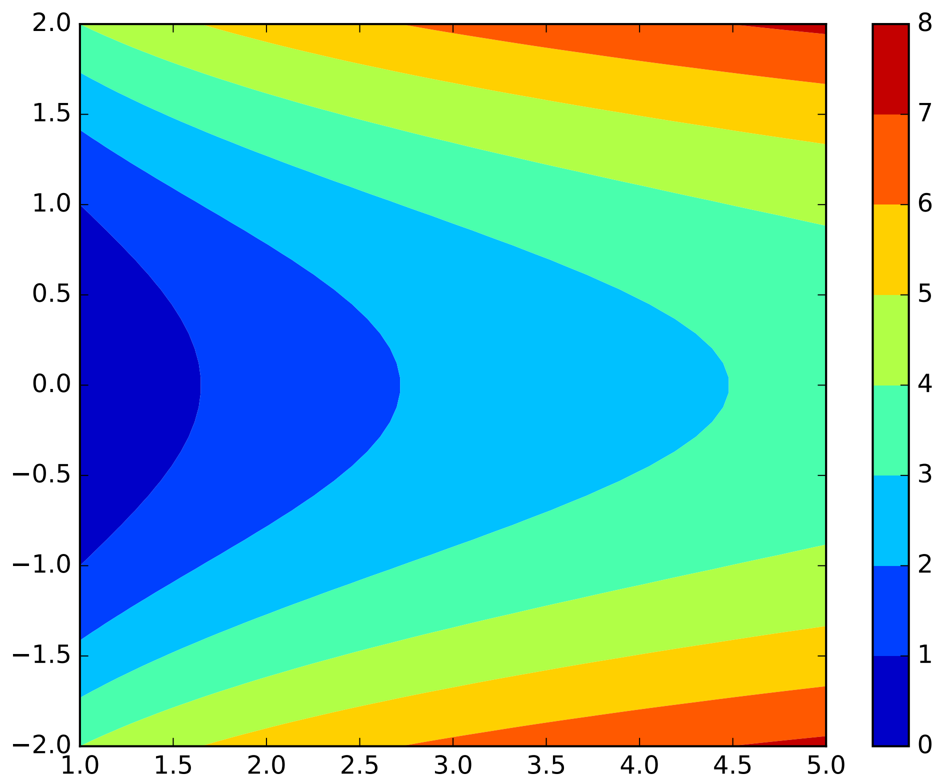
Sada crtamo ekvipotencijalne konture:

```
CS = plt.contour(X, Y, Z)
fig = plt.clabel(CS, inline=1, fontsize=10, colors='black')
```



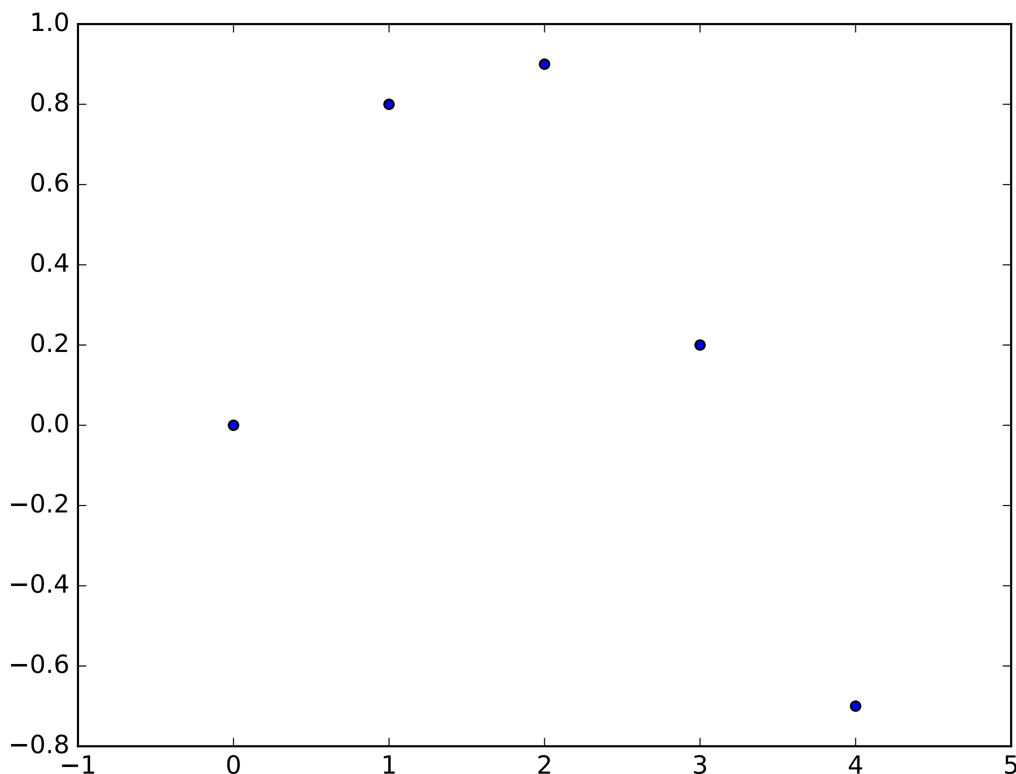
Alternativni prikaz kontura iste ove funkcije:

```
CS = plt.contourf(X, Y, Z)
plt.colorbar(CS)
```



Za crtanje podataka organiziranih u listu parova $[[x_1, y_1], [x_2, y_2], \ldots]$ rabimo `plt.scatter`:

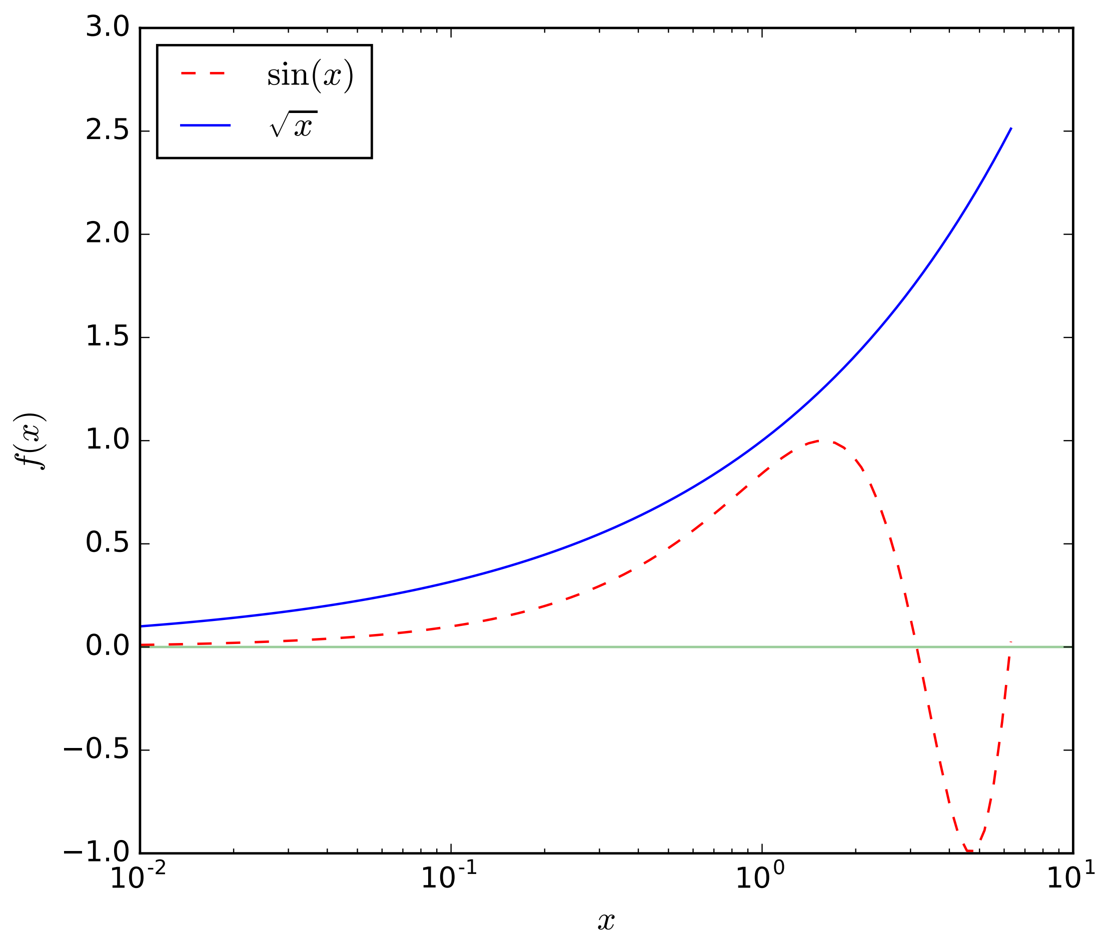
```
data = np.array([[0,0], [1,0.8], [2, 0.9], [3, 0.2],[4, -0.7]])
plt.scatter(data[:,0], data[:,1])
```



Gornji primjeri funkcioniraju unutar Jupyter okruženja. U čistom Pythonu potrebno je još dodatno na početku inicijalizirati sliku (za to je dobra funkcija `plt.subplots`) i na kraju zatražiti njeno iscrtavanje na ekran (`plt.show`) ili u datoteku (`plt.savefig`).

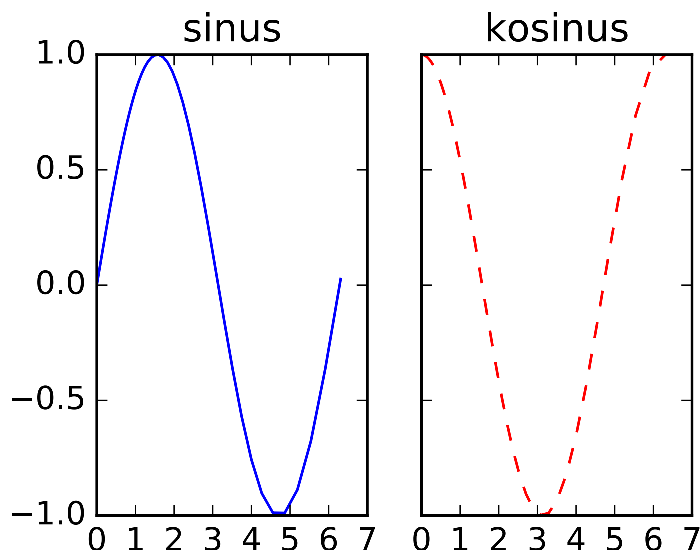
Slijedeći primjer pokazuje graf s logaritamskom osi, legendom i LaTeX oznakama.

```
xs = np.logspace(-2.0, 0.8, 100)  # granice su log_10(x)
fig, ax = plt.subplots(figsize=[7,6])
ax.plot(xs, sin(xs), color='red', linestyle='--', label='$\sin(x)$')
ax.plot(xs, sqrt(xs), 'b-', label='$\sqrt{x}$')
ax.set_xscale('log')
ax.axhline(0, color='g', linewidth=1, alpha=0.4)
ax.set_xlabel('$x$', fontsize=14)
ax.set_ylabel('$f(x)$', fontsize=14)
ax.legend(loc="upper left")
#fig.show()  # ovo bi trebalo izvan Jupytera
```



Funkcija `plt.subplots` stvara dva objekta: sliku (*figure*) i panel (*axis*). Složene slike mogu imati više panela, kao u slijedećem primjeru:

```
fig, (ax1, ax2) = plt.subplots(1, 2, sharey=True, figsize=[4,3])
ax1.plot(xs, sin(xs))
ax1.set_title('sinus')
ax2.plot(xs, cos(xs), color='red', linestyle='--')
ax2.set_title('kosinus')
#fig.show()
```



Zadatak 1

Uočite da se funkcija `fibBinet(x)` iz *prošlog odjeljka* može izvrijedniti i za vrijednosti argumenta x koje nisu cjelobrojne. Nacrtajte graf te funkcije za $-4 < x < 7$ i superponirajte na njega (u drugoj boji) točke koje odgovaraju vrijednostima funkcije za pozitivne cjelobrojne argumente `[(1, F_1), (2, F_2), ... (7, F_7)]`.

Zadatak 2

Koristeći trigonometrijske funkcije nacrtajte kružnicu kao parametarsku krivulju.

Zadatak 3

Nacrtajte 100 slučajno raspoređenih točaka (hint: `np.random.rand()`), gdje su x i y koordinate tih točaka u intervalu (-1,1), a nacrtane su na dijagramu s x i y osima koje se protežu u intervalu (-2, 2).

Zadatak 4

Nacrtajte kružnicu u kompleksnoj ravnini zadanu kompleksnim brojevima

$$\{z = \rho e^{i\eta}(1 + e^{i\phi}) - 1 \quad | \quad \phi \in [0, 2\pi), \rho = 1.3, \eta = 0.2\}$$

te krivulju koja se dobije kad se ova kružnica podvrgne transformaciji Žukovskog:

$$z \to w = \frac{1}{2}\left(z + \frac{1}{z}\right)$$

Matematika

4.1 Simbolički izrazi

Za manipulacije simboličkim izrazima u Pythonu, na raspolaganju nam je paket `sympy`

```
>>> from sympy import *
```

Kao prvo, potrebno je na slijedeći način deklarirati varijable koje namjeravamo koristiti u simboličkim izrazima

```
>>> a, b, c, x, y, z, t = symbols('a, b, c, x, y, z, t')
>>> alpha, beta = symbols('alpha, beta')
```

(Simboli s lijeve i desne strane ovih deklaracija se ne moraju nužno slagati, što može biti korisno za npr. skraćeni unos grčkih slova, ali općenito je to recept za probleme.)

Pomoću ovih varijabli sad izgrađujemo simboličke izraze:

```
>>> i1 = (beta+alpha)**3; i1
(alpha + beta)**3
```

Ukoliko želimo ljepši ispis možemo ga uključiti s

```
>>> init_printing()
>>> i1
               3
(alpha + beta)
```

(U Jupyter notebooku bi ovo bilo prikazano još ljepše, s pravim grčkim slovima.) U ostatku ovog dokumenta nećemo koristiti ovu mogućnost.

Sympy ne provodi skoro nikakve operacije na izrazima dok to eksplicitno ne zatražimo. Recimo da želimo razviti gornji izraz koristeći binomni teorem. Za to služi funkcija `expand()`

```
>>> expand(i1)
alpha**3 + 3*alpha**2*beta + 3*alpha*beta**2 + beta**3
```

Funkcija `expand()`, kao i mnoge druge, se može alternativno upotrijebiti i kao *metoda* simboličkog izraza.

```
>>> i1.expand()
alpha**3 + 3*alpha**2*beta + 3*alpha*beta**2 + beta**3
```

U prvom pristupu `expand` doživljavamo kao funkciju ili operaciju, dok je izraz `i1` njen argument odnosno operand. To je način razmišljanja svojstven standardnom *proceduralnom* ili pak tzv. *funkcionalnom programiranju*.

U drugom pristupu izraz `i1` treba pak doživljavati kao *objekt* , u smislu tzv. *objektno-orijentiranog* (OO) programiranja, a `expand()` je tzv. *metoda* što je naziv za funkciju koja je pridružena tipu objekta na koji djeluje [1].

Da bi saznali što pojedina metoda radi, upišemo je nakon odgovarajućeg objekta i operatora točkice `.`, dodamo upitnik i stisnemo TAB. Pri upotrebi metode ne smije se zaboraviti na zagrade, koje su često prazne, ali nekad sadrže opcionalne argumente kojima modificiramo ponašanje metode. Ukoliko zaboravimo zagrade Python ne poziva funkciju već samo ispisuje njeno ime poput "<metoda expand pridružena objektu tom i tom>":

```
>>> i1.expand
<bound method Expr.expand of (alpha + beta)**3>
```

Tek zagrade daju zahtjev interpreteru da dotičnu metodu i pozove tj. izvrši.

Naravno ovakve jednostavne izraze možemo razviti i na ruke, dok računalo blista kad radi s velikim izrazima (sve dok stanu u memoriju računala)

```
>>> i2 = (a +2*b + 3*c)**3 * (x+y)**3
>>> i2.expand()
a**3*x**3 + 3*a**3*x**2*y + 3*a**3*x*y**2 + a**3*y**3 + 6*a**2*b*x**3 + 18*a**2*b*x**2*y
```

Potenciranjem i razvijanjem gornjeg izraza dobivamo izraz od 550 članova ...

```
>>> i3 = expand(i2**3)
>>> len(i3.args)
550
```

... kojeg sympy s lakoćom faktorizira:

```
>>> factor(i3)
(x + y)**9*(a + 2*b + 3*c)**9
```

Napomena: Svaki simbolički izraz ima strukturu `funkcija(arg1, arg2, ...)` gdje argumenti mogu opet biti druge funkcije sa svojim argumentima, tvoreći tako drvoliku strukturu. Funkcija i argumenti nekog izraza pohranjeni su u atributima `func` i `args`, što smo gore iskoristili za brojanje članova izraza `i3` gdje je funkcija naprosto operacija zbrajanja

```
>>> i3.func
<class 'sympy.core.add.Add'>
```

Cijelu strukturu izraza možemo ispisati pomoću funkcije `srepr`. Vještom upotrebom `func` i `args` možemo proizvoljno manipulirati simboličkim izrazima.

Često je korisno izraz organizirati kao polinom u nekoj varijabli. Za to služi funkcija `collect()`:

[1] To onda omogućuje da metode istog imena rade različite stvari s različitim objektima (tzv. *polimorfizam*). Kako je python OO jezik, takva sintaksa se obilato koristi i brojne funkcije se ni ne mogu koristiti na prvi način. Jedna od prednosti takvog pristupa je da elegantno možemo saznati popis svih funkcija koje rade nešto smisleno sa zadanim objektom, i to tako da nakon što stavimo točkicu "." poslije objekta stisnemo TAB tipku. Dobit ćemo popis svih metoda tog objekta. Ovo međutim ne funkcionira s netom upisanim izrazom u trenutnoj ćeliji, već samo s ranije definiranim izrazima (objektima) kojima smo pridjelili ime. Pridjeljivanje imena objektima se izvodi znakom jednakosti i korisno je ne samo zbog navedenog razloga već i inače radi lakšeg baratanja izrazima i kasnijeg referiranja na iste.

```
>>> i4=i2.expand().collect(y)
>>> i4
a**3*x**3 + 6*a**2*b*x**3 + 9*a**2*c*x**3 + 12*a*b**2*x**3 + 36*a*b*c*x**3 + 27*a*c**2*x
```

```
>>> len(i4.args)
13
```

(Uočite da `collect()` ne pojednostavljuje koeficijente [2].) Za dobiti koeficijent uz neku potenciju neke varijable koristi se funkcija `coeff()`. Npr, koeficijent uz a^9 jest

```
>>> i3.coeff(a, 9)
x**9 + 9*x**8*y + 36*x**7*y**2 + 84*x**6*y**3 + 126*x**5*y**4 + 126*x**4*y**5 + 84*x**3*
```

Najsveobuhvatnija funkcija za pojednostavljivanje simboličkih izraza je `simplify()`:

```
>>> i5 = a/(1-a) + a/(1+a); i5
a/(a + 1) + a/(-a + 1)
```

```
>>> i5.simplify()
-2*a/(a**2 - 1)
```

Funkcija `simplify()` je kompozicija elementarnijih funkcija za pojednostavljivanje izraza. Jedna od tih elementarnijih funkcija je `trigsimp()` koja pri pojednostavljivanju rabi samo trigonometrijske identitete.

```
>>> (sin(x)**4 + 2*sin(x)**2*cos(x)**2 + cos(x)**4).trigsimp()
1
```

Još neke funkcije za pojednostavljivanje simboličkih izraza su `expand_trig`, `powsimp`, `expand_log`, `logcombine`.

Uočite da sympy neće naivno "pojednostaviti" izraze koji uključuju multifunkcije, kao na slijedećem primjeru, u kojem upoznajemo i važnu metodu `subs()` koja služi za uvrštavanje vrijednosti varijabli i druge supstitucije u izrazima

```
>>> i6 = log(a) + log(b)
>>> i6.simplify()
log(a) + log(b)
>>> i6.subs({a:-1, b:-1})
2*I*pi
>>> i7 = log(a*b)
>>> i7.subs([(a,-1),(b,-1)])
0
```

(Vidimo da argument od `subs` može biti rječnik, ali i lista koja definira zamjene.) Inače, simboličke varijable se mogu definirati i s dodatnim svojstvima koja onda mogu omogućiti željena pojednostavljenja izraza:

```
>>> m, n = symbols('m, n', positive=True)
>>> (log(m) + log(n)).simplify()
log(m*n)
```

Napomena: Za bolju kontrolu sympy koristi svoje vlastite klase brojeva, a ne one Pythonove. To omogućuje npr. upotrebu racionalnih brojeva bez gubitka točnosti i njihov prirodni ispis:

[2] To se može postići npr. ovako:

```
>>> i1 = (x/7).subs(x, 3); i1
3/7
>>> srepr(i1)
'Rational(3, 7)'
>>> srepr(7*i1)
'Integer(3)'
```

Da bi očuvali ta svojstva trebamo pripaziti da ne unosimo u izraze Pythonove brojeve s pomičnom točkom (float). Problem obično nastaje kad unesemo omjer dva cijela broja koja Python pretvori u float prije nego sympy napravi konverziju u svoje tipove.

```
>>> 2**(1/2)
1.4142135623730951
```

Da bi to spriječili dovoljno je napraviti eksplicitnu konverziju ("sympyfikaciju") jednog od brojeva funkcijom `S`

```
>>> 2**(S(1)/2)
sqrt(2)
```

Napomena: Slično, sympy ima svoj skup simboličkih matematičkih konstanti: bazu prirodnog logaritma `E`, Ludolfov broj `pi` (sic!), imaginarnu jedinicu `I`, beskonačnost `oo` itd. Usporedimo ponašanje broja π iz scipy i simpy paketa:

```
>>> import scipy
>>> scipy.exp(1j*scipy.pi)
(-1+1.2246467991473532e-16j)

>>> exp(I*pi)
-1
>>> pi.is_irrational
True
```

Vidimo da je čuvena Eulerova relacija sa scipy konstantama zadovoljena samo na konačnu točnost standardnih brojeva s pomičnom točkom. (Ovdje je bilo nužno ponovno učitati scipy paket i to na način da ne dođe do kolizije `pi` i `exp` sa istoimenim objektima iz sympy paketa.)

Napomena: Ako na kraju poželimo vidjeti rezultat u obliku broja s pomičnom točkom koristimo funkciju `N`:

```
>>> N(sqrt(2))
1.41421356237310
>>> N(pi, n=50)
3.1415926535897932384626433832795028841971693993751
```

Alternativno ime ove funkcije je `evalf` i u tom obliku se ona obično koristi kao metoda

```
>>> E.evalf()
2.71828182845905
```

Zadatak 1

Uzmite izraz $(a+b)((c+yx)x+tx^2)$ i algebarskim manipulacijama postignite prikaz u slijedećim ekvivalentnim oblicima

1. $x(a+b)(c+tx+xy)$
2. $acx+atx^2+ax^2y+bcx+btx^2+bx^2y$

Zadatak 2

Koristeći algebarske manipulacije pokažite da vrijedi

$$\frac{\sin^3 x + \cos^3 x}{\sin x + \cos x} = 1 - \sin x \cos x$$

Pazite na sintaksu: $\sin^3 x$ se unosi kao sin(x)**3!

Zadatak 3

Za izraz $f(x)$ kažemo da je *paran*, odnosno *neparan* u varijabli x ako vrijedi $f(x) = f(-x)$, odnosno $f(x) = -f(-x)$. Isprogramirajte funkciju `parnost(izraz, varijabla)` koja će testirati to svojstvo i vraćati 1 ako je izraz paran u varijabli, -1 ako je neparan i 0 ako nije ni paran ni neparan.

Zadatak 4

Isprogramirajte funkciju `leg(n, x)` koja vraća Legendreov polinom $P_n(x)$ u simboličkoj varijabli *x*, koristeći rekurzijsku relaciju

$$P_n(x) = \frac{2n-1}{n} x P_{n-1}(x) - \frac{n-1}{n} P_n(x)$$

Funkcija treba davati identične rezultate kao sympy funkcija legendre(n, x).

Bilješke

```
>>> sum(i4.coeff(y, a).factor()*y**a for a in range(4))
x**3*(a + 2*b + 3*c)**3 + 3*x**2*y*(a + 2*b + 3*c)**3 + 3*x*y**2*(a + 2*b + 3*c)**3 + y*
```

```
>>> from sympy import *
>>> x, y, z, a, b, c = symbols('x, y, z, a, b, c')
```

4.2 Jednadžbe

Simboličko rješavanje jednadžbi se izvodi funkcijom `solveset` [1]. Riješimo jednadžbu

$$2x^2 = 1$$

[1] Postoji i stara funkcija `solve` koja je inferiorna ovoj, osim za nelinearne susustave jednadžbi. Vidi ovdje.

```
>>> sol = solveset(2 * x**2 - 1, x); sol
{-sqrt(2)/2, sqrt(2)/2}
```

Valja primijetiti slijedeće:

1. Jednadžba se postavlja kao izraz kojem se traži nultočka.
2. Treba eksplicitno naznačiti po kojoj varijabli se traži rješavanje.
3. Pronađena su oba rješenja kvadratne jednadžbe.
4. Rezultat je ispisan u obliku skupa. To je najfleksibilniji Sympy objekt koji omogućuje zapis svih vrsta rješenja kakva se mogu pojaviti.

Provjera gornjih rješenja može ići npr ovako:

```
>>> [2 * x**2 - 1 for x in sol]
[0, 0]
```

Za linearne sustave jednadžbi koristimo `linsolve`. Npr. sustav:

$$x + 2y + 3z = 1 \tag{4.1}$$
$$3x + y + z = -6 \tag{4.2}$$
$$2x + 4y + 9z = 2 \tag{4.3}$$

rješavamo ovako:

```
>>> eqns = [x + 2*y + 3*z - 1, 3*x + y + z + 6, 2*x + 4*y + 9*z - 2]
>>> sol = linsolve(eqns, x, y, z); sol
{(-13/5, 9/5, 0)}
```

Da bi dobili numeričke vrijednosti ovih rješenja ne možemo jednostavno primijeniti funkciju `N()` na ovo rješenje, jer `N()` odnosno `evalf` nije metoda skupa (a ni tupla), već samo primitivnijih objekata:

```
>>> N(sol)
Traceback (most recent call last):
...
AttributeError: 'Tuple' object has no attribute '_eval_evalf'
```

Stoga je potrebno nekako primijeniti `N()` izravno na brojeve unutar tupla rješenja:

```
>>> [(N(x), N(y), N(z)) for x,y,z in sol]
[(-2.60000000000000, 1.80000000000000, 0)]
```

Zadatak 1

Riješite sustav jednadžbi

$$x + y = 3 \tag{4.4}$$
$$2x + 2y = 6 \tag{4.5}$$

Koliko ima rješenja?

Evo drugog primjera s beskonačnim skupom rješenja:

```
init_printing()
solveset(sin(x) - 1, x)

          pi
{2*n*pi + -- | n in Integers()}
          2
```

Neka rješenja npr. jednadžbi viših stupnjeva nije moguće analitički zapisati i `solveset` daje rezultat u slijedećem obliku:

```
>>> sol = solveset(9*x**6 + 4*x**4 + 3*x**3 + x - 17, x); sol
{1, CRootOf(9*x**5 + 9*x**4 + 13*x**3 + 16*x**2 + 16*x + 17, 0), CRootOf(9*x**5 + 9*x**4
```

Vidimo samo jedno realno rješenje i 5 kompleksnih ("Complex root of ..."). Ovo je sad moguće numerički izvrijedniti:

```
>>> [N(x) for x in sol]
[1.00000000000000, -1.10301507262981, -0.491102035999093 - 0.988331495372071*I, -0.4911
```

Na kraju, neke jednadžbe se uopće ne mogu analitički egzaktno riješiti. Npr.

$$2 \arctan y = y^2$$

```
>>> eq = 2 * atan(y) - y**2
>>> solveset(eq, y)
ConditionSet(y, Eq(-y**2 + 2*atan(y), 0), Complexes((-oo, oo) x (-oo, oo), False))
```

(Inverzne trigonometrijske funkcije u sympyu (i Pythonovom paketu math) se zovu `atan`, `asinh` itd. dok se u scipy i numpy paketima koriste `arctan`, `arcsinh` itd.)

U gornjem slučaju moramo pribjeći pravom numeričkom rješavanju. Napuštamo sympy i koristimo `scipy.optimize` paket i njegovu funkciju `root`.

Mala komplikacija s `root` je da je potrebno dati početnu vrijednost varijable od koje se traži rješenje i da će pronaći samo jedno rješenje. Također, funkcija čiju nultočku tražimo treba biti definirana kao Python funkcija.

```
>>> from scipy import *
>>> from scipy.optimize import root

>>> def fun(y):
...     return 2 * arctan(y) - y**2

>>> sol = root(fun, 0); sol
    fjac: array([[-1.]])
     fun: array([ 0.])
 message: 'The solution converged.'
    nfev: 3
     qtf: array([ 0.])
       r: array([-1.99999999])
  status: 1
 success: True
       x: array([ 0.])
```

Vidimo da je rješenje koje daje `root` složeni objekt, a samoj vrijednosti nultočke pristupamo putem atributa `x` (koji se zove `x` bez obzira na to kako se zove varijabla funkcije, u našem slučaju `y`!).

Eventualno drugo rješenje treba tražiti dalje zadavanjem drugih početnih vrijednosti.

```
>>> sol = root(fun, 1); sol.x[0]
1.371774342015089
```

Nastavljamo dalje ...

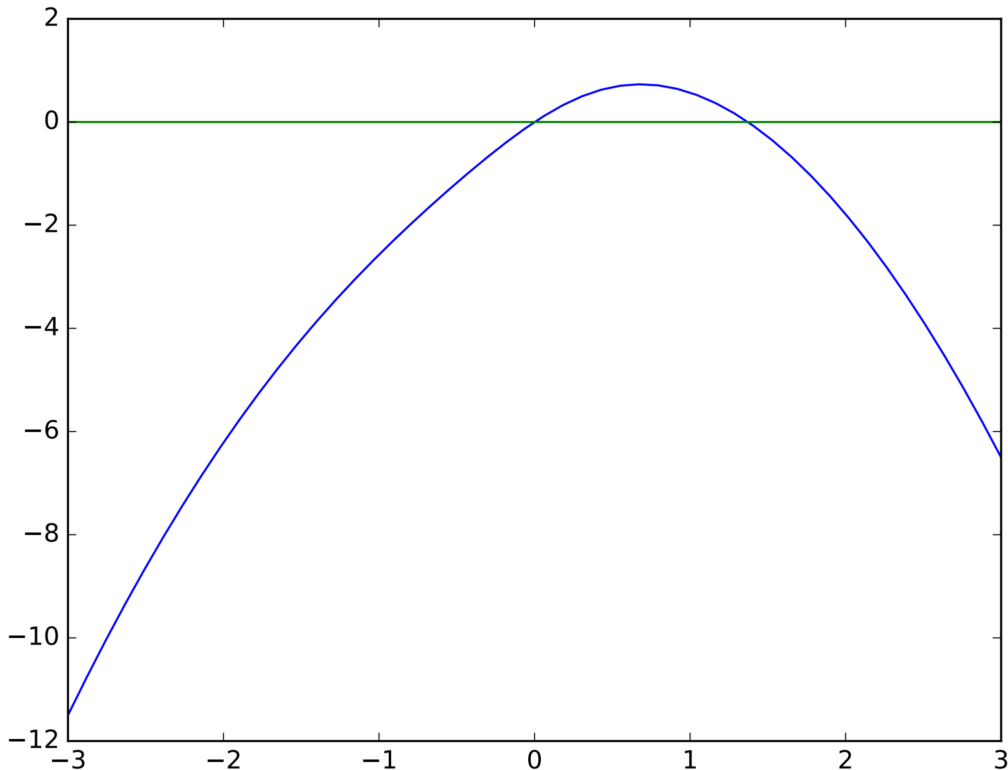
```
>>> sol = root(fun, 10); sol.x[0]
1.371774342015089

>>> sol = root(fun, -10); sol.x[0]
0.0
```

Sada se možemo uvjeriti da su dva pronađena rješenja zaista jedina, i to tako da skiciramo graf funkcije i uočimo da siječe apscisu na samo dva mjesta.

```
>>> from scipy import *
>>> from scipy.optimize import root
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

```
>>> def fun(y):
...     return 2 * arctan(y) - y**2
```

```
>>> xs = np.linspace(-3, 3)
>>> fig = plt.plot(xs, fun(xs))
>>> fig = plt.axhline(0, color='green')
```



Inače, kad nam treba jednostavna “funkcija za jednokratnu upotrebu”, poput funkcije `fun` gore, možemo koristiti tzv. *lambda-funkciju*

```
>>> sol = root(lambda x: 2 * arctan(x) - x**2, 1); sol.x[0]
1.371774342015089
```

Također, za brzu pretvorbu simboličkih izraza u Python funkcije, korisna je funkcija `lambdify`. Tako smo gore, prilikom numeričkog rješavanja jednadžbe, umjesto definiranja nove funkcije mogli "lambdificirati" već definirani simbolički izraz.

```
>>> root(lambdify(y, eq), 1).x[0]
1.371774342015089
```

Ovo omogućuje i crtanje simboličkog izraza matplotlibom (opcija `numpy` dolje omogućuje lambda funkciji da se distribura po numpy polju):

```
plt.plot(xs, lambdify(y, eq, 'numpy')(xs))
fig = plt.axhline(0, color='green')
```

No, za jednostavne skice simbličkih izraza možemo koristiti i sympy funkciju `plot`:

```
plot(eq, (y, -3, 3))
```

Za usporedbu raznih pristupa pretvorbi simboličkih izraza u numeričke, vidi tablicu na kraju ove stranice.

Zadatak 2

Riješite jednadžbu

$$\tan x - \frac{x}{10} = 0\,.$$

Zadatak 3

Pronađite pozicije lokalnog minimuma te lokalnog maksimuma gama funkcije $\Gamma(x)$ koji su najbliži točki $x = 0$

```
>>> from scipy.special import gamma
>>> from scipy.optimize import minimize
```

Zadatak 4

Pronađite pozicije lokalnih minimuma i maksimuma Besselove funkcije $J_1(x)$ koji su najbliži točki $x = 0$.

```
>>> from scipy.special import j1
>>> from scipy.optimize import minimize
```

Zadatak 5

Krivulje

$$y = 2x^2 + \alpha x + 3\alpha$$

za svaki α prolaze kroz istu točku (x_0, y_0). (U to se možete i grafički uvjeriti.) Definirajte funkciju $y(x, \alpha)$, zatim rješavanjem sustava jednadžbi odredite tu točku i na kraju ispišite broj $x_0 + y_0$.

Bilješke

```
>>> from sympy import *
>>> x, y, z, t, a, b, c = symbols('x, y, z, t, a, b, c')
```

4.3 Matematička analiza

Za simbolički pristup standardnim temama iz matematičke analize, poput limesa, nizova, redova, derivacija, integrala i diferencijalnih jednadžbi, koristimo sympy paket.

4.3.1 Limesi

Limesi se izvrijednjavaju funkcijom `limit`:

```
>>> limit(sin(x)/x, x, 0)
1

>>> limit((1+1/x)**x, x, oo)
E
```

4.3.2 Razvoj u red

Taylorov razvoj neke funkcije po nekoj varijabli oko neke točke do nekog reda radi funkcija `series`:

```
>>> ser = series(sin(x), x, 0, 6); ser
x - x**3/6 + x**5/120 + O(x**6)
```

gdje je upotrijebljena standardna Landauova oznaka zanemarenog ostatka. S ovakvim se izrazom može dalje računati i ostatak apsorbira članove tog i višeg reda:

```
>>> ser + 1 + x**7
1 + x - x**3/6 + x**5/120 + O(x**6)
```

a ukoliko nam taj ostatak smeta, možemo ga maknuti metodom `removeO`:

```
>>> ser.removeO()
x**5/120 - x**3/6 + x
```

Zadatak 1

Odredite Taylorov razvoj funkcije $f(x) = \arctan(x^2 + 1)$ oko točke $x = \infty$ do petog reda.

Zadatak 2

Kolika je *relativna* pogreška koju radimo ako za izvrijednjavanje $f(2) = \arctan(5)$ koristimo red dobiven u gornjem zadatku? Da li bi greška bila manja da smo upotrebljavali razvoj do petog reda, ali oko $x = 0$?

Zadatak 3

Za koji x greška razvoja funkcije $f(x) = \arctan(x^2 + 1)$ do petog reda oko točke $x = 0$ postaje jednaka greški istog razvoja, ali oko točke $x = \infty$?

4.3.3 Derivacije

Deriviranje se radi funkcijom `diff`

```
>>> diff(exp(2*x)*cos(3*x), x)
-3*exp(2*x)*sin(3*x) + 2*exp(2*x)*cos(3*x)
```

Višestruko deriviranje:

```
>>> diff(exp(2*x)*cos(3*x), x, 4)
(120*sin(3*x) - 119*cos(3*x))*exp(2*x)
```

Deriviranje izraza koji uključuje opću simboličku funkciju `f(x)`, korištenjem Leibnitzovog lančanog pravila:

```
>>> f = Function('f')  # simboličke funkcije treba deklarirati
>>> df = diff(x**2 * f(x), x); df
x**2*Derivative(f(x), x) + 2*x*f(x)

>>> df.subs(f, sin)
x**2*Derivative(sin(x), x) + 2*x*sin(x)

>>> df.subs(f, sin).doit()
x**2*cos(x) + 2*x*sin(x)
```

4.3.4 Simboličko integriranje

Simboličko neodređeno integriranje (integriramo izraz koji smo gore dobili deriviranjem):

```
>>> integrate(-3*exp(2*x)*sin(3*x) + 2*exp(2*x)*cos(3*x), x)
exp(2*x)*cos(3*x)
```

Primijetite da se konstanta integracije podrazumijeva bez navođenja.

Simboličko rješavanje određenih integrala:

```
integrate(x*log(x), (x, 0, 1))
-1/4
```

Zadatak 4

Izračunajte neodređeni integral

$$\int \frac{x^2+3}{x^5+x^4-x-1}dx$$

i onda provjerite dobiveni rezultat deriviranjem i *algebarskim manipulacijama*.

Zadatak 5

Izračunajte (simbolički) dvostruki određeni integral

$$\int_0^1 dx \int_0^{1-x} dy\, xy^2$$

Zadatak 6

Izračunajte dvostruki neodređeni integral

$$\int dx \int dy\, (x+y)\sqrt{xy^3}$$

i provjerite rezultat deriviranjem.

4.3.5 Numeričko integriranje

Neki integrali su preteški ili se naprosto ne daju prikazati u zatvorenoj formi pa sympy vraća zadani izraz:

```
>>> integrate(atan(gamma(x)), (x, 0, 1))
Integral(atan(gamma(x)), (x, 0, 1))
```

Tada nam preostaje numerička integracija. U paketu `scipy.integrate` ima nekoliko rutina za numeričku integraciju. Npr. `quad` daje rezultat kao tupl. Prvi element tupla je rezultat integracije, a drugi procijenjena greška.

```
>>> import scipy.integrate
>>> scipy.integrate.quad(lambdify(x, atan(gamma(x))), 0, 1)
(1.1020657425550973, 2.3166346669288508e-14)
```

Zadatak 7

Izračunajte integral

$$\int_0^\infty dt\, e^{-t}\sqrt{t}$$

simbolički i numerički i usporedite rezultate. Naputak: `oo` je simbolička beskonačnost, a za potrebe numeričke integracije, beskonačnost je reprezentirana kao `np.inf`.

4.4 Linearna algebra

Kao vektore i matrice koristit ćemo jednodimenzionalna i dvodimenzionalna NumPy polja te rutine NumPy modula za linearnu algebru.

```
>>> import numpy as np
>>> import numpy.linalg as la


>>> v1 = np.array([1,1,2])
>>> v2 = np.array((2,2,4))
```

Množenje skalarom je prirodno:

```
>>> 3*v1
array([3, 3, 6])
```

Skalarni i vektorski produkt vektora:

```
>>> np.dot(v1, v2)
12
```

```
>>> np.cross(v1, v2)
array([0, 0, 0])
```

Norma ("duljina") vektora:

```
>>> la.norm(v2)
4.8989794855663558
```

Množenje matrica te množenje matrice i vektora ide na prirodan način:

```
>>> A = np.array([[1, 2, 1], [4, 3, 3], [9, 1, 7]]); A
array([[1, 2, 1],
       [4, 3, 3],
       [9, 1, 7]])


>>> np.dot(A, A)
array([[18,  9, 14],
       [43, 20, 34],
       [76, 28, 61]])
```

```
>>> np.dot(A, v1)
array([ 5, 13, 24])
```

Determinanta i inverz matrice:

```
>>> la.det(A)    # = 7
-6.9999999999999947
```

```
>>> la.inv(A)
array([[-2.57142857,  1.85714286, -0.42857143],
       [ 0.14285714,  0.28571429, -0.14285714],
       [ 3.28571429, -2.42857143,  0.71428571]])
```

Numerika po defaultu ide s *double precision* točnošću, pa kad provjeravamo da li je $A^{-1}A = 1$ ne dobijemo egzaktno jediničnu matricu:

```
>>> should_be_unit = np.dot(la.inv(A), A); should_be_unit
array([[  1.00000000e+00,   1.60982339e-15,   4.44089210e-16],
       [ -2.22044605e-16,   1.00000000e+00,  -2.22044605e-16],
       [  0.00000000e+00,  -1.33226763e-15,   1.00000000e+00]])
```

Moguće je zatražiti od NumPyja da ispisuje ovakve male brojeve kao nule:

```
>>> np.set_printoptions(suppress=True)
>>> should_be_unit
array([[ 1.,  0.,  0.],
       [-0.,  1., -0.],
       [ 0., -0.,  1.]])
```

Zadatak 1

Za datu matricu A definiramo svojstvene vektore (eigenvectors) $\mathbf{v}$ i njima pripadajuće svojstvene vrijednosti λ (eigenvalues) kao rješenja matrične jednadžbe

$$A\mathbf{v} = \lambda\mathbf{v} \ .$$

Pomoću funkcije la.eig() odredite svojstvene vrijednosti i svojstvene vektore matrice

$$\begin{pmatrix} 2.3 & 4.5 \\ 6.7 & -1.2 \end{pmatrix} ,$$

i provjerite da dobivena rješenja zaista zadovoljavaju gornju jednadžbu.

Zadatak 2

Kreirajte 3x3 matricu sa slučajnim realnim brojevima između 0 i 10. Invertirajte je i pomnožite s originalnom matricom te se uvjerite da dobijete jediničnu matricu.

Dijagonalizacija matrice A je pronalaženje njenog rastava oblika

$$A = PDP^{-1}$$

gdje je D dijagonalna matrica. Takav rastav se također izvodi fukcijom la.eig(), koja daje svojstvene vrijednosti i svojstvene vektore matrice. Naime, stupci matrice P su upravo svojstveni vektori od A, a dijagonalna matrica D na dijagonali ima upravo odgovarajuće svojstvene vrijednosti:

```
>>> A = np.array([[3, 1], [1, 3]]); A
array([[3, 1],
       [1, 3]])

>>> evs, P = la.eig(A)

>>> D = np.diag(evs); D
array([[ 4.,  0.],
       [ 0.,  2.]])

>>> np.dot(np.dot(la.inv(P), A), P)   # provjeravamo P^-1 A P = D
array([[ 4.,  0.],
       [ 0.,  2.]])
```

Neke matrice nije moguće dijagonalizirati, u slučaju čega će matrice P i D koje vraća `la.eig()` i dalje zadovoljavati

$$AP = PD$$

ali P neće biti invertibilna:

```
>>> A = np.array([[1, 1], [0, 1]]); A
array([[1, 1],
       [0, 1]])

>>> evs, P = la.eig(A)
```

```
>>> np.dot(A, P) - np.dot(P, np.diag(evs))
array([[ 0.,  0.],
       [ 0.,  0.]])
```

Neinvertibilnost od P se ogleda u ogromnim brojevima koje dobijemo kad "invertiramo" tu matricu:

```
>>> print(la.inv(P))
[[  1.00000000e+00   4.50359963e+15]
 [  0.00000000e+00   4.50359963e+15]]
```

Za rad sa simboličkim matricama možemo koristiti odgovarajuće objekte iz sympyja

4.5 Diferencijalne jednadžbe

4.5.1 Simboličko rješavanje

Python sympy funkcija koja traži analitička rješenja običnih diferencijalnih jednadžbi je `desolve()`. Riješimo pomoću nje jednadžbu gušenog harmoničkog oscilatora

$$\frac{d^2y}{dt^2} + 2\frac{dy}{dt} + 4y = 0\,.$$

Prilikom definiranja diferencijalne jednadžbe treba eksplicitno deklarirati simboličku nezavisnu varijablu (obično je to vrijeme `t`) i nepoznatu simboličku funkciju (ovdje je to `y(t)`), pomoću opcije funkcije `symbols`:

```
>>> from sympy import *
>>> t = symbols('t')
>>> y = symbols('y', cls=Function)

>>> gsol = dsolve(y(t).diff(t, t) + 2*y(t).diff(t) + 4*y(t), y(t)); gsol
Eq(y(t), (C1*sin(sqrt(3)*t) + C2*cos(sqrt(3)*t))*exp(-t))
```

Napomene:

1. Kao što je kod običnih jednadžbi trebalo eksplicirati po kojoj varijabli se traži rješenje, tako se kod diferencijalnih specificira po kojoj funkciji se traži rješenje
2. Rješenje je izraženo kao sympy jednadžba (objekt tipa `Eq`) i za pristup samoj funkciji možemo koristiti metodu `rhs`
3. Rješenje je općenito tj. uključuje nepoznate konstante (C_1 i C_2) koje treba odrediti iz početnih uvjeta. To je moguće npr. na slijedeći način

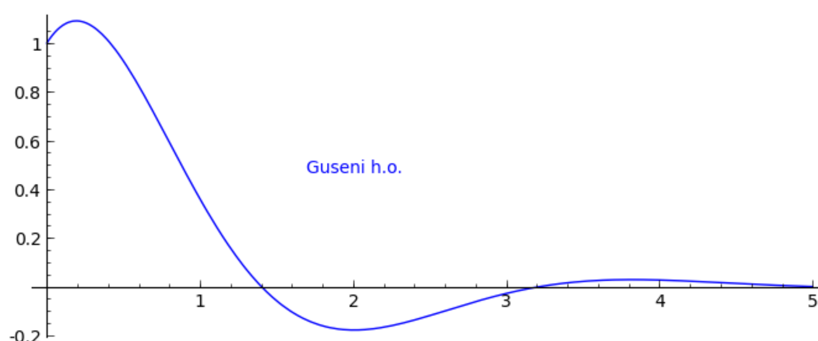
```
>>> C1, C2 = symbols('C1, C2')
>>> consts = linsolve([gsol.rhs.subs(t,0) - 1, gsol.rhs.diff(t).subs(t, 0) -1], C1, C2);
{(2*sqrt(3)/3, 1)}
```

Sad možemo ove konstante uvrstiti u rješenje. Prvo ćemo tupl koji je element jednočlanog skupa rješenja pretvoriti u rječnik:

```
>>> dics = [{C1:c1, C2:c2} for c1, c2 in consts]; dics
[{C2: 1, C1: 2*sqrt(3)/3}]

>>> sol = gsol.rhs.subs(dics[0]); sol
(2*sqrt(3)*sin(sqrt(3)*t)/3 + cos(sqrt(3)*t))*exp(-t)
```

To je sad konkretno rješenje koje možemo nacrtati npr kao

```
>>> plot(sol, (t, 0, 10))
```



Zadatak 1

Riješite simbolički diferencijalnu jednadžbu

$$\frac{dy}{dt} - y^2 - 1 = 0 \, ,$$

uz početni uvjet $y(0) = 1/2$, provjerite rješenje uvrštavanjem, te ga nacrtajte za t u rasponu $0 < t < 1$.

4.5.2 Numeričko rješavanje

Neke jednadžbe se ne mogu riješiti analitički, pa moramo pribjeći numeričkom integriranju. Za to ćemo koristiti funkciju `odeint` iz paketa `scipy.integrate`.

```
>>> from scipy.integrate import odeint
>>> import matplotlib.pyplot as plt
>>> import numpy as np
```

Kao prvi primjer integrirati ćemo jednadžbu iz gornjeg zadatka. Za `odeint` tu jednadžbu treba transformirati u oblik

$$\frac{dy}{dt} = f(y, t)$$

i korisnik treba definirati Python funkciju koja odgovara desnoj strani ove jednadžbe.

U našem slučaju je $f(y, t) = y^2 + 1$ pa stoga imamo

```
>>> def func(y, t):
...     return y**2 + 1
```

Funkciju `odeint` treba pozvati s ovom funkcijom kao prvim argumentom, početnom vrijednošću $y(0)$ kao drugim argumentom i listom vremenskih točaka za koje želimo odrediti položaj sustava kao trećim argumentom:

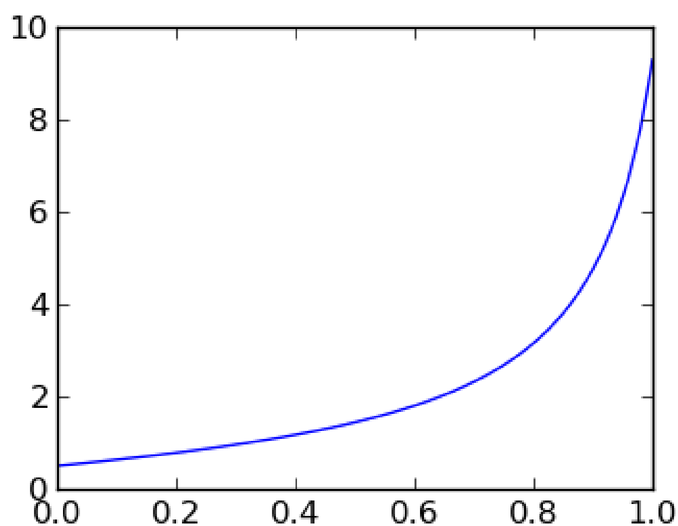
```
>>> y0 = 0.5
>>> ts = np.linspace(0, 1)
>>> ts.shape
(50,)
```

Funkcija `odeint` vraća listu položaja u traženim vremenskim točkama:

```
>>> ys = odeint(func, y0, ts)
>>> ys.shape
(50, 1)
```

```
>>> print("\n{:8s}  {:8s}".format('vrijeme', 'položaj'))
>>> print(18*'-')
>>> for t,y in zip(ts[:5], ys[:5, 0]):
...     print("{:8f}  {:8f}".format(t, y))
>>> print("{:8s}  {:8s}".format('  (...)  ', '  (...)  '))

 vrijeme  položaj
------------------
0.000000  0.500000
0.020408  0.525777
0.040816  0.552113
0.061224  0.579049
0.081633  0.606630
 (...)     (...)
```

```
plt.plot(ts, ys)
```



Kao slijedeći primjer proučiti ćemo tzv. van der Polovu jednadžbu, koja je drugog reda:

$$\frac{d^2x}{dt^2} - (1 - x^2)\frac{dx}{dt} + x = 0\,.$$

Jednadžbe višeg reda se za numeričku analizu treba prirediti tako da se svaka pretvori u sustav od dvije jednadžbe prvog reda, što se izvodi uvođenjem nove funkcije $y(t) = dx/dt$ uz pomoć koje gornju jednadžbu možemo pretvoriti u ekvivalentni sustav

$$\frac{dx}{dt} = y$$

$$\frac{dy}{dt} = (1 - x^2)y - x$$

Taj sustav sad treba zapisati u vektorskom obliku tako da poprimi strukturu

$$\frac{dy_k}{dt} = f_k(\vec{y}, t)\,, \qquad k = 1, 2$$

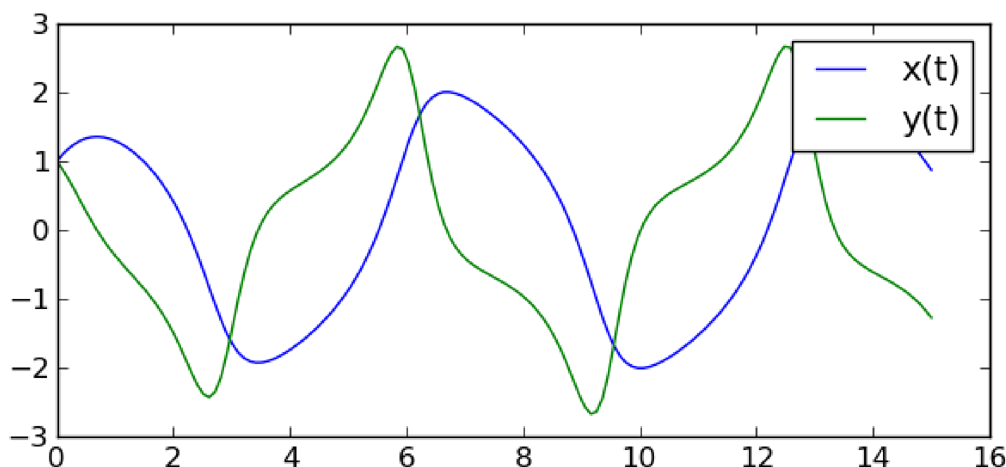
gdje je dvokomponentni vektor $\vec{y} = (x, y) \equiv$ `[y[0], y[1]]`. Sada funkcija koju priređujemo za `odeint` opet odgovara desnoj strani ove jednadžbe i mora kao rezultat vratiti dvokomponentni vektor $\vec{f} = (f_1, f_2)$

```
>>> def func(y, t):
...     return [y[1], (1 - y[0]**2)*y[1] - y[0]]
```

Početni uvjet je isto vektor $(x(0), y(0))$, a i rezultat će biti NumPy polje oblika `(broj vremenskih točaka) x (broj nepoznatih funkcija)`.

```
>>> ts = np.linspace(0,15,150)
>>> y0 = [1., 1.]
>>> ys = odeint(func, y0, ts)
>>> ys.shape
(150, 2)
```

```
>>> fig, ax = plt.subplots(figsize=[7,3])
>>> ax.plot(ts, ys[:,0], label='x(t)')
>>> ax.plot(ts, ys[:,1], label='y(t)')
>>> ax.legend()
```



Zadatak 2

Nacrtajte graf brzine gornjeg rješenja Van der Polove jednadžbe $y(t) = dx(t)/dt$, ali ne u ovisnosti o vremenu t, već o položaju $x(t)$ (tzv. fazni dijagram).

Zadatak 3

Riješite numerički diferencijalnu jednadžbu gušenog harmoničkog oscilatora. Nacrtajte funkcije `y(t)` i `x(t)`, te, na posebnom grafu, fazni dijagram `y(x)`.

4.6 Statistika

Za statističke funkcije koristit ćemo paket `scipy.stats`.

```
>>> from scipy import *
>>> import scipy.stats
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> np.random.seed(42)   # za reproducibilnost
```

Kreirajmo prvo NumPy listu brojeva. NumPy liste imaju metode za izračun osnovnih statističkih veličina poput srednje vrijednosti, varijance, standardne devijacije itd.

```
>>> xs=np.linspace(1,9,21); xs
array([ 1. ,  1.4,  1.8,  2.2,  2.6,  3. ,  3.4,  3.8,  4.2,  4.6,  5. ,
        5.4,  5.8,  6.2,  6.6,  7. ,  7.4,  7.8,  8.2,  8.6,  9. ])
```

```
>>> print("{:.2f} +- {:.2f}".format(xs.mean(), xs.std()))
5.00 +- 2.42
```

Obične liste nemaju gornje statističke metode pa ukoliko nas zanima npr. srednja vrijednost brojeva u običnoj listi treba je prvo konvertirati u numpy listu funkcijom `np.array()` ili možemo primijeniti izravno primijeniti numpy funkciju `mean()` (dakle ne metodu).

```
>>> np.mean([3, 5, 7, 9, 11])
7.0
```

scipy.stats paket ima implementirane brojne statističke raspodjele:

- `norm` - normalna (Gaussova) raspodjela
- `poisson` - Poissonova raspodjela
- `gamma` - gamma raspodjela
- `chi2` - χ^2 raspodjela
- `t` - studentova t raspodjela
- ...

Svaka od tih raspodjela ima svoje parametre (npr. za Gaussovu raspodjelu su to srednja vrijednost i standardna devijacija). Oni se najčešće specificiraju putem opcionalnih argumenata, a točnu sintaksu saznajemo iz standardne dokumentacije.

Najvažnije metode distribucija su:

- `pdf()` - sama distribucija (*probability distribution function*) $f(x)$
- `cdf()` - integral distribucije (*cumulative distribution function*) $\Phi(x) = \int_{-\infty}^{x} f(y)dy$
- `rvs()` - slučajni brojevi (*random variates*) koji slijede distribuciju

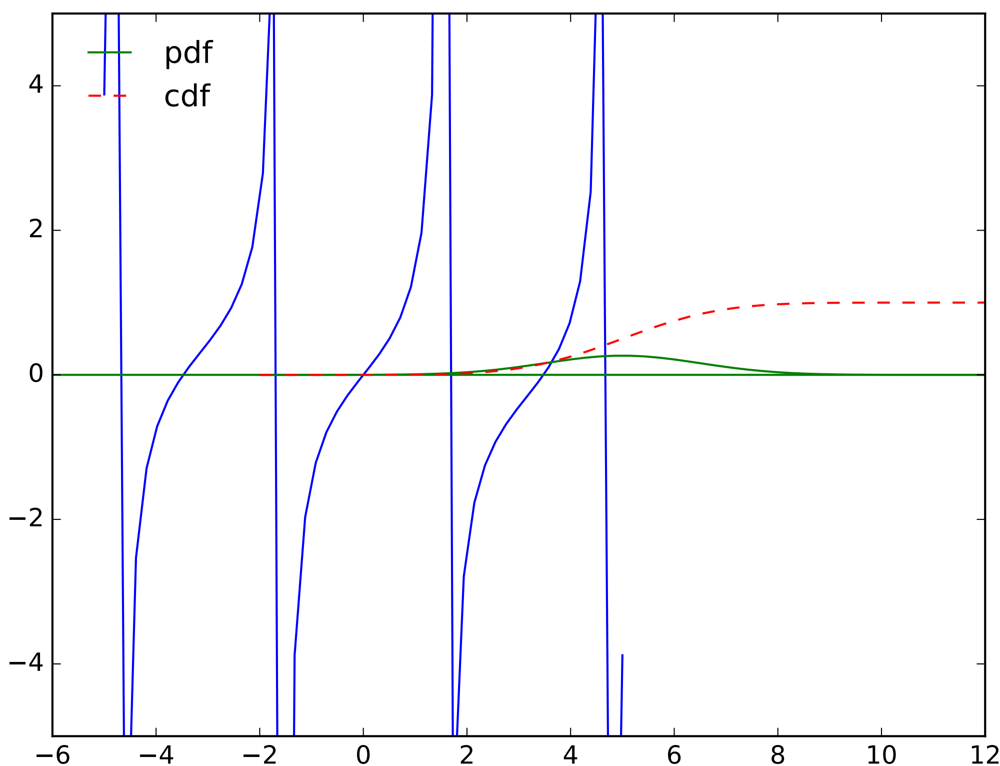
Tako je npr. vrijednost u $x = 0$ normalne distribucije sa srednjom vrijednosti $\mu = 5$ i standardnom devijacijom $\sigma = 1.5$, $f(x = 0, \mu = 5, \sigma = 1.5)$ dan s:

```
>>> scipy.stats.norm.pdf(0, loc=5., scale=1.5)
0.0010281859975274036
```

Skica distribucije i njenog integrala:

```
>>> from scipy import *
>>> import scipy.stats
>>> import numpy as np
>>> import matplotlib.pyplot as plt
```

```
>>> xs = np.linspace(-2,12)
>>> fig = plt.plot(xs, scipy.stats.norm.pdf(xs, loc=5., scale=1.5), label='pdf')
>>> fig = plt.plot(xs, scipy.stats.norm.cdf(xs, loc=5., scale=1.5), 'r--', label='cdf')
>>> fig = plt.legend(loc='upper left').draw_frame(0)
```



Izračunajmo vjerojatnost da se slučajna varijabla nađe unutar jedne standardne devijacije σ od srednje vrijednosti μ. Ona je dana integralom distribucije od $\mu - \sigma$ do $\mu + \sigma$. Kako na raspolaganju imamo funkciju `cdf()` koja daje integral od $-\infty$ do x, treba samo oduzeti dva takva integrala (koristimo i to da su defaultne vrijednosti `loc=0` i `scale=1`):

```
>>> scipy.stats.norm.cdf(1) - scipy.stats.norm.cdf(-1)
0.68268949213708585
```

(To je čuvenih 68% vjerojatnosti.)

Zadatak 1

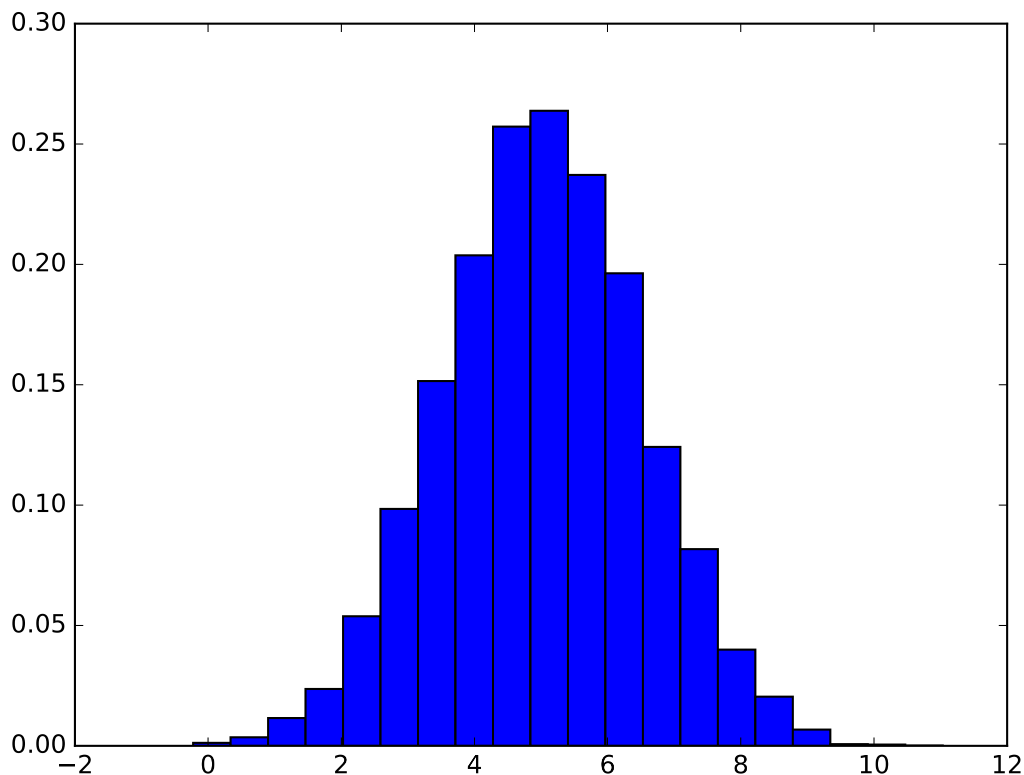
Slučajna varijabla X slijedi normalnu raspodjelu sa srednjom vrijednosti $\mu = 50$ i standardnom devijacijom $\sigma = 2$. Kolika je vjerojatnost da se X nađe između 47 i 54?

Sad ćemo metodom `rvs()` izgenerirati uzorak od 10000 brojeva raspodjeljenih po normalnoj raspodjeli s $\mu = 5$ i $\sigma = 1.5$ i testirati da su srednja vrijednost i standardna devijacija prema očekivanjima:

```
>>> pts=scipy.stats.norm.rvs(loc=5., scale=1.5, size=10**4)
>>> print("broj točaka = {}".format(len(pts)))
broj točaka = 10000
>>> print("srednja vrijednost = {:.3f}".format(pts.mean()))
srednja vrijednost = 4.997
>>> print("standardna devijacija = {:.3f}".format(pts.std()))
standardna devijacija = 1.505
```

Za crtanje histograma ovog uzorka koristimo Matplotlibovu `hist()` funkciju, čiji opcionalni argument `bins` kontrolira broj "binova" dakle broj intervala apscise u kojima se prebrojavaju točke:

```
>>> pts=scipy.stats.norm.rvs(loc=5., scale=1.5, size=10**4)
>>> fig = plt.hist(pts, bins=20, normed=True)
```



Ovo je bilo za jedan uzorak. Listu npr. standardnih devijacija za 5 nezavisnih uzoraka možemo konstruirati ovako:

```
>>> [scipy.stats.norm.rvs(loc=5, scale=1.5, size=10**4).std() for k in
...                                                    range(5)]
[1.4870528846676399, 1.5066552692834156, 1.5005205582344263, 1.5145268557221607, 1.48869
```

Zadatak 2

Kolika je vjerojatnost da mjerenje slučajne varijable koja slijedi normalnu razdiobu sa srednjom vrijednošću μ i standardnom devijacijom σ odstupi od srednje vrijednosti za više od 3σ?

Važna raspodjela je tzv. χ^2-raspodjela

$$P(\chi^2, \nu) = \frac{(\chi^2)^{\nu/2-1} e^{-\chi^2/2}}{2^{\nu/2}\Gamma(\nu/2)}$$

Vidimo da χ^2 raspodjela, pored argumenta x, ima samo jedan parametar: ν = df = broj stupnjeva slobode (*degrees of freedom*) [1].

Srednja vrijednost χ^2 raspodjele jednaka je broju stupnjeva slobode:

```
>>> scipy.stats.chi2.rvs(3, size=10**5).mean()
3.0026555158916577
```

Jedno od svojstava χ^2 raspodjele je da integral repa te raspodjele (s jednim stupnjem) od 1, 4, 9 do ∞ daje vjerojatnost da mjerenje slučajne varijable raspodjeljene po Gaussovoj raspodjeli sa standardnom devijacijom σ odstupi od srednje vrijednosti za $\sigma, 2\sigma, 3\sigma, \ldots$.

```
>>> [1-scipy.stats.chi2.cdf(eps**2,1) for eps in [1,2,3]]
[0.31731050786291404, 0.045500263896358528, 0.0026997960632602069]
```

Zadatak 3

Središnji granični teorem kaže da su srednje vrijednosti dovoljno velikih uzoraka neke raspodjele raspodjeljene prema normalnoj (Gaussovoj) raspodjeli bez obzira na to kakva je originalna raspodjela iz koje te uzorke uzimamo. U to ćemo se uvjeriti na primjeru χ^2 raspodjele.

1. Nacrtajte tu raspodjelu za df=3 stupnja slobode i uvjerite se da je asimetrična oko srednje vrijednosti.
2. Uzmite jedan uzorak od milijun točaka te raspodjele i uvjerite se da je standardna devijacija $\sigma = \sqrt{2 \cdot df}$
3. Uzmite 500 uzoraka od po 400 točaka svaki i napravite listu srednjih vrijednosti tih uzoraka. Uvjerite se da je standardna devijacija vrijednosti u listi $\sigma/\sqrt{400}$, kako traži teorem.
4. Nacrtajte histogram srednjih vrijednosti u listi i uvjerite se (superponiranjem krivulje normalne razdiobe) da su one zaista raspodjeljene prema normalnoj razdiobi srednje vrijednosti $\mu = df = 3$ i standardne devijacije $\sigma/\sqrt{400}$

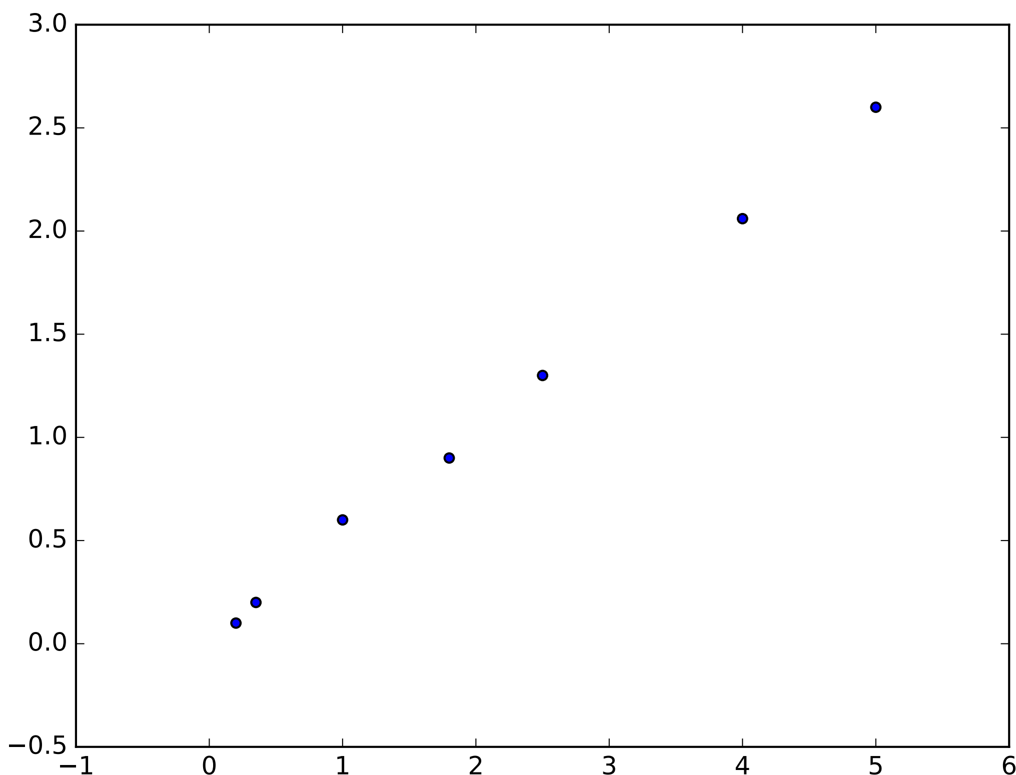
Bilješke

4.7 Prilagodba funkcije podacima

Često je potrebno neke podatke dobivene npr. mjerenjima grafički prikazati i približno opisati nekom funkcijom (tzv. prilagodba ili "fitanje").

[1] Naziv tog parametra postaje jasan u kontekstima određivanja dobrote prilagodbe i statističkog *testiranja hipoteze*.

```
from scipy import *
import numpy as np
import matplotlib.pyplot as plt
data = np.array([[0.2,  0.1], [0.35,  0.2], [1, 0.6], [1.8, 0.9], [2.5, 1.3], [4, 2.06],
x = data[:, 0]
y = data[:, 1]
plt.scatter(x, y)
```



Klasična situacija je potreba za prilagodbom pravca $f(x) = a + bx$ metodom najmanjih kvadrata (linearna regresija). Pogledajmo prvo kako bismo sami implementirali standardne formule za metodu najmanjih kvadrata:

```
>>> from scipy import *
>>> import numpy as np
>>> data = np.array([[0.2,  0.1], [0.35,  0.2], [1, 0.6], [1.8, 0.9], [2.5, 1.3], [4, 2.
>>> x = data[:, 0]
>>> y = data[:, 1]
>>> n = len(data)
>>> n = len(data)
>>> delc = n*(x**2).sum() - x.sum()**2
>>> b = (n*(x*y).sum()-x.sum()*y.sum())/delc
>>> a = ((x**2).sum()*y.sum()-x.sum()*(x*y).sum())/delc
>>> sigsq = ((y-b*x-a)**2).sum()/(n-2)
>>> erra = sqrt((x**2).sum()*sigsq/delc)
>>> errb = sqrt(n*sigsq/delc)

>>> print("a = {:.3f} +/- {:.3f}".format(a, erra))
a = 0.020 +/- 0.023
```

```
>>> print("b = {:.3f} +/- {:.3f}".format(b, errb))
b = 0.513 +/- 0.008
```

Paket `scipy.optimize` sadrži funkciju `curve_fit` koja radi otprilike to isto, ali je općenitija u smislu da krivulja koju prilagođavamo može biti zadana bilo kojim matematičkim izrazom. Pogledajmo prvo prilagodbu pravca. Prvo je potrebno definirati odgovarajuću Python funkciju:

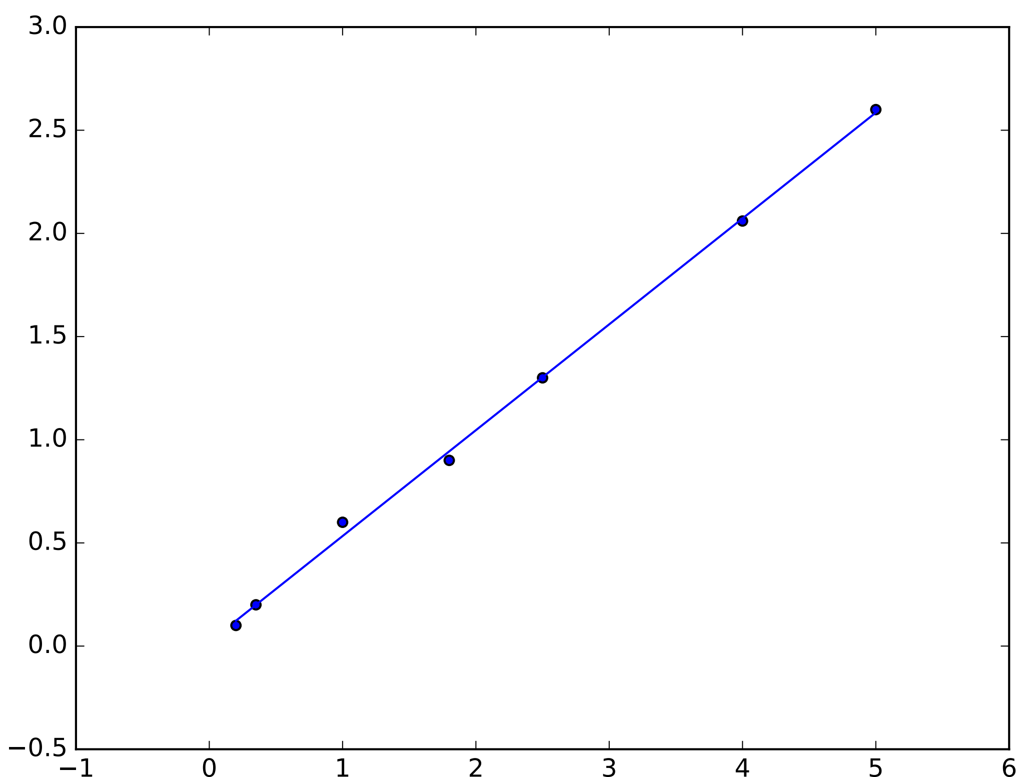
```
>>> def model(x, a, b):
...     return a + b*x

>>> from scipy.optimize import curve_fit
>>> pars, cov = curve_fit(model, x, y)
>>> pars
array([ 0.02015952,  0.51305612])
```

Funkcija `curve_fit` vraća dva objekta: vektor s numeričkim vrijednostima parametara dobivenih prilagodbom i matricu kovarijanci. Za nas je dovoljno znati da dijagonala te matrice sadrži kvadrat neodređenosti (varijancu) parametara. Tako vrijednosti i neodređenosti parametara možemo ispisati npr. ovako:

```
>>> for k, par in enumerate(['a', 'b']):
...    print('{} = {:.3f}  +/-  {:.3f}'.format(par, pars[k], sqrt(cov[k,k])))
a = 0.020  +/-  0.023
b = 0.513  +/-  0.008
```

Vidimo da se rezultat slaže s gornjim. Nacrtajmo prilagođeni pravac zajedno s podacima:

```
from scipy.optimize import curve_fit
def model(x, a, b):
    return a + b*x
pars, cov = curve_fit(model, x, y)
plt.plot(x, model(x, *pars))
```



Možemo se sad zapitati da li je linearni model tj. pravac dovoljan ili ima smisla dodati još i kvadratni član. U tom slučaju prilagodba izgleda ovako:

```
>>> def model(x, a, b, c):
...     return a + b*x + c*x**2

>>> pars, cov = curve_fit(model, x, y)
>>> for k, par in enumerate(['a', 'b', 'c']):
...    print('{} = {:.3f}  +/-  {:.3f}'.format(par, pars[k], sqrt(cov[k,k])))
a = 0.027  +/-  0.035
b = 0.502  +/-  0.037
c = 0.002  +/-  0.007
```

Činjenica da je neodređenost paramera `c` znatno veća od vrijednosti samog parametra, sugerira da je kvadratni član suvišan. (Štoviše, vidimo da je i slobodni član `a` od malog značaja.)

Zadatak 1

Donjim podacima iz liste `data2` prilagodite polinom trećeg reda, te funkciju $f(x) = a\sin(x)$ te nacrtajte sve na jednom grafu. Radi lakšeg snalaženja neka krivulje budu različitih boja. Izračunajte zbroj kvadrata odstupanja $\sum_i (y_i - f(x_i))^2$ za oba slučaja. Razmislite koji model je bolja "teorija" koja objašnjava dane eksperimentalne podatke.

```
>>> data2 = np.array([[0, 0.1], [0.3, 0.4], [1, 0.7], [1.8, 1], [2.5, 0.5], [4, -0.6],
```

Fizika

5.1 Mehanika

Zadatak M-1

Neka na česticu u ravnini djeluje potencijal $U(x, y) = -4x^2y^3$. Nacrtajte ekvipotencijalne konture ovog potencijala. Nadalje, rješavajući numerički Newtonovu diferencijalnu jednadžbu gibanja, pronađite putanju čestice, uz početne uvjete $x(0) = y(0) = -1, v_x(0) = 2, v_y(0) = -3$, od trenutka $t = 0$ do trenutka $t = 1.43$. Nacrtajte tu putanju, a onda je pokušajte superponirati na gore dobivene konture potencijala. Igrajte se malo s početnim uvjetima i uvjerite se da je sve OK. Grafički usporedite kinetičku, potencijalnu i ukupnu energiju u ovisnosti o vremenu.

Zadatak M-2

Formule za nerelativističku i relativističku kinetičku energiju tijela mase `m` koje se giba brzinom `v` su

$$K_{\mathrm{nr}} = \frac{1}{2}mv^2\,, \qquad K_{\mathrm{rel}} = mc^2\left(\frac{1}{\sqrt{1 - v^2/c^2}} - 1\right)\,.$$

1. Definirajte odgovarajuće funkcije `knr(v)` i `krel(v)` te usporedite na istom dijagramu ponašanje tih funkcija za brzine od 0 do $3 \cdot 10^8$ m/s za jediničnu vrijednost mase. Označite veličine i njihove jedinice na koordinatnim osima!
2. Odredite brzinu pri kojoj je greška nerelativističke formule tisućinku promila.
3. Da li se `knr(v)` i `krel(v)` slažu za male vrijednosti brzine? Ako ne, korigirajte `krel()` da postignete slaganje.

5.1.1 Problem tri tijela

Numeričkim rješavanjem Newtonovih jednadžbi simuliramo gibanje tri tijela koja gravitiraju, a čije je gibanje ograničeno na x-y ravninu. Radimo s jediničnim masama i G=1 vrijednošću Newtonove gravitacijske konstante. Za tri tijela u dvodimenzionalnoj ravnini imamo 6 diferencijalnih jednadžbi drugog reda koje pretvaramo u 12 diferencijalnih jednadžbi prvog reda. Donji kod je napravljen za Sage sustav. Možete ga izvršiti ako instalirate Sage ili online na sagemathcloud-u. Dobra je vježba konvertirati to u Jupyter/Python.

```
sage: def system(t, y):
....:     # y_i = (x1, y1, x2, y2, x3, y3
....:     #  i=     0   1   2   3   4   5
....:     #        vx1, vy1, vx2, vy2, vx3, vy3)
....:     #         6    7    8    9   10   11
....:     return [y[6], y[7], y[8], y[9], y[10], y[11],
....:     -((y[0] - y[2])/((y[0] - y[2])**2 + (y[1] - y[3])**2)**1.5) -
....:     (y[0] - y[4])/((y[0] - y[4])**2 + (y[1] - y[5])**2)**1.5,
....:     -((y[1] - y[3])/((y[0] - y[2])**2 + (y[1] - y[3])**2)**1.5) -
....:     (y[1] - y[5])/((y[0] - y[4])**2 + (y[1] - y[5])**2)**1.5,
....:     (y[0] - y[2])/((y[0] - y[2])**2 + (y[1] - y[3])**2)**1.5 -
....:     (y[2] - y[4])/((y[2] - y[4])**2 + (y[3] - y[5])**2)**1.5,
....:     (y[1] - y[3])/((y[0] - y[2])**2 + (y[1] - y[3])**2)**1.5 -
....:     (y[3] - y[5])/((y[2] - y[4])**2 + (y[3] - y[5])**2)**1.5,
....:     (y[0] - y[4])/((y[0] - y[4])**2 + (y[1] - y[5])**2)**1.5 +
....:     (y[2] - y[4])/((y[2] - y[4])**2 + (y[3] - y[5])**2)**1.5,
....:     (y[1] - y[5])/((y[0] - y[4])**2 + (y[1] - y[5])**2)**1.5 +
....:     (y[3] - y[5])/((y[2] - y[4])**2 + (y[3] - y[5])**2)**1.5,
....:     ]
```

Za početne uvjete uzimamo neke od 15 periodičkih orbita navedenih u Tablici 1 članka Šuvakov i Dmitrašinović, Phys. Rev. Lett. 110, 114301 (2013), dostupno online: arXiv:1303.0181.

```
sage: figure8=[[-1,0,1,0,0,0,0.347111,0.532728,0.347111,0.532728,-0.694222,
....:            -1.06546],[-1.05783,1.05784,-0.522919,0.522917],6.32445]
sage: goggles=[[-1,0,1,0,0,0,0.0833,0.127889,0.0833,0.127889,-0.1666,
....:            -0.255778], [-1.04973,1.04972,-0.738248,0.738249],10.4668]
sage: butterfly2=[[-1,0,1,0,0,0,0.392955,0.097579,0.392955,0.097579,-0.78591,
....:               -0.195158],[-1.07444,1.07449,-0.174197,0.17414],7.00391]
sage: yinyang2a=[[-1,0,1,0,0,0,0.416822,0.330333,0.416822,0.330333,
....:      -0.833644,-0.660666],[-1.0874,1.08734,-0.88595,0.885999],55.7898]
```

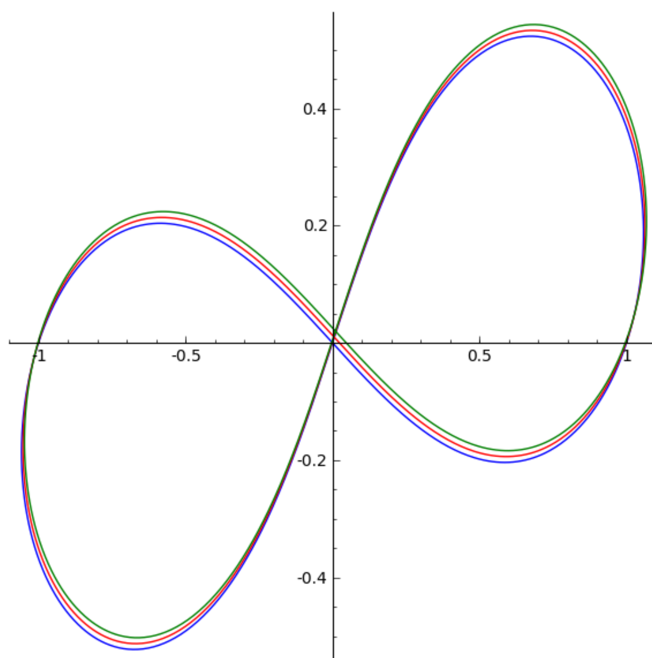
Umjesto `odeint`, koristit čemo `ode_solver`.

```
sage: S = ode_solver()
sage: S.function = system
sage: S.ode_solve(y_0=figure8[0], t_span=[0, figure8[2]], num_points=1000)
sage: sol_figure8 = S.solution
sage: S.ode_solve(y_0=goggles[0], t_span=[0, goggles[2]], num_points=2000)
sage: sol_goggles = S.solution
sage: S.ode_solve(y_0=butterfly2[0], t_span=[0, butterfly2[2]],
....:                                                     num_points=2000)
sage: sol_butterfly2 = S.solution
sage: S.ode_solve(y_0=yinyang2a[0], t_span=[0, yinyang2a[2]],
....:                                                     num_points=6000)
sage: sol_yinyang2a = S.solution
```

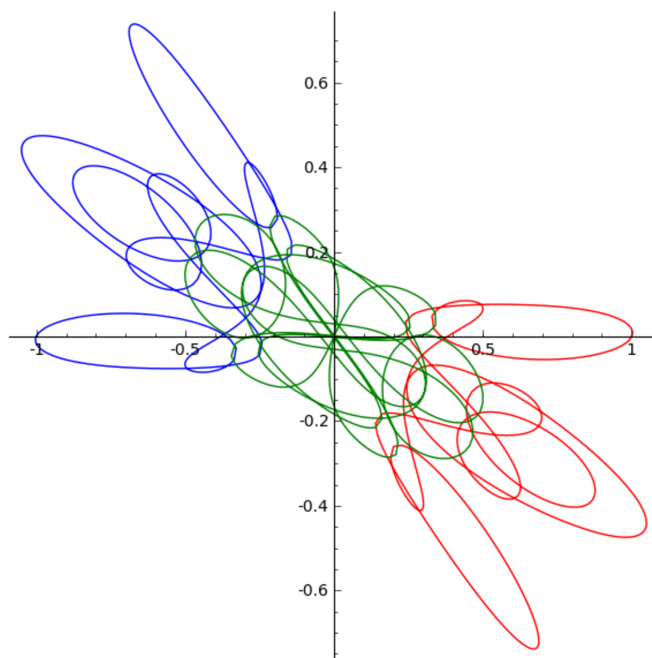
Također, umjesto crtanja Matplotlibom koristit ćemo Sageove grafičke elemente, konkretno primitivni grafički objekt `line`.

```
sage: G = Graphics()
sage: xshift = 0.005 # for visibility
sage: yshift = 0.01 # for visibility
sage: G += line([(x[0],x[1]) for t,x in sol_figure8])
sage: G += line([(x[2]+xshift,x[3]+yshift) for t,x in sol_figure8],
....:                                                     color='red')
sage: G += line([(x[4]+2*xshift,x[5]+2*yshift) for t,x in sol_figure8],
....:                                                   color='green')
```

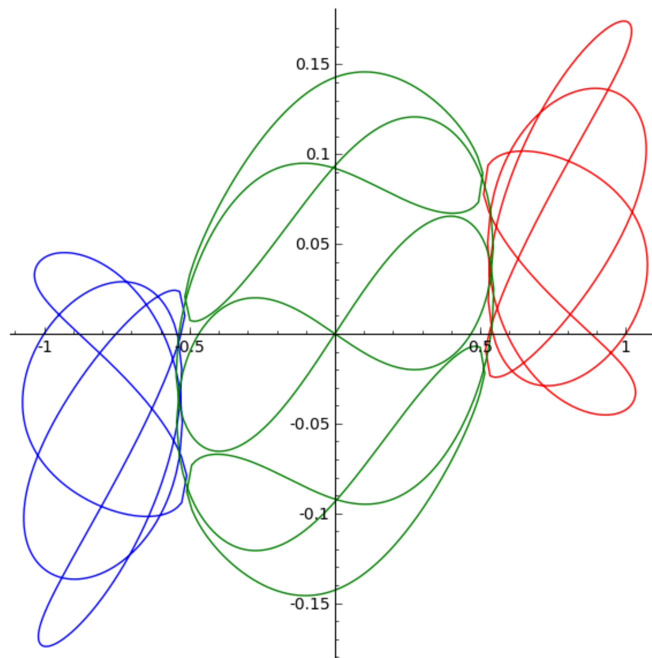
```
sage: show(G, figsize = [6,6])
```



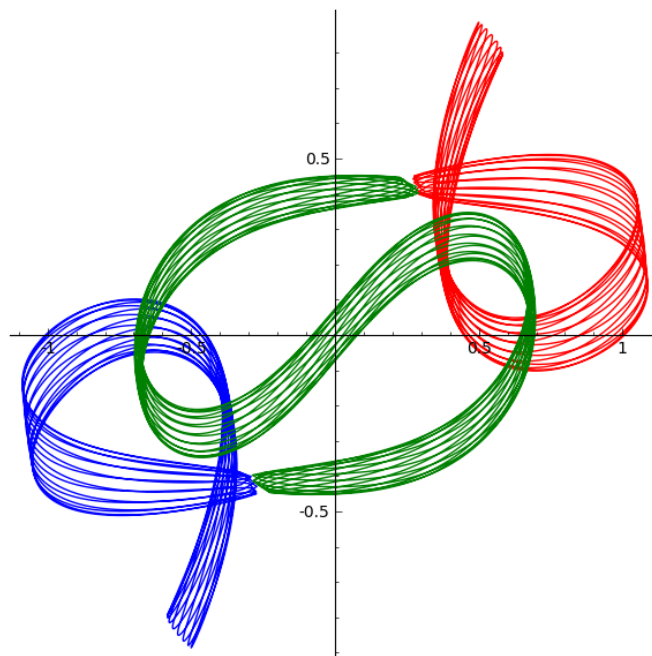

```
sage: G = Graphics()
sage: G += line([(x[0],x[1]) for t,x in sol_goggles])
sage: G += line([(x[2],x[3]) for t,x in sol_goggles], color='red')
sage: G += line([(x[4],x[5]) for t,x in sol_goggles], color='green')
sage: show(G, figsize = [6,6])
```




```
sage: G = Graphics()
sage: G += line([(x[0],x[1]) for t,x in sol_butterfly2])
sage: G += line([(x[2],x[3]) for t,x in sol_butterfly2], color='red')
sage: G += line([(x[4],x[5]) for t,x in sol_butterfly2], color='green')
sage: show(G, figsize = [6,6])
```




```
sage: G = Graphics()
sage: G += line([(x[0],x[1]) for t,x in sol_yinyang2a])
sage: G += line([(x[2],x[3]) for t,x in sol_yinyang2a], color='red')
sage: G += line([(x[4],x[5]) for t,x in sol_yinyang2a], color='green')
sage: show(G, figsize = [6,6])
```



5.2 Elektrodinamika

Zadatak E-1

Potencijal $\phi(\vec{r})$, u točki $\vec{r} = (x, y, z)$ koji je posljedica statičke raspodjele N nabijenih čestica s nabojima $q_1, q_2, \ldots, q_N$ i položajima $\vec{r}_1 = (x_1, y_1, z_1)$, $\vec{r}_2 = (x_2, y_2, z_2)$,..., $\vec{r}_N = (x_N, y_N, z_N)$ je dan formulom:

$$\phi(x, y, z) = \sum_{k=1}^{N} \frac{q_k}{|\vec{r} - \vec{r}_k|} .$$

Definirajte funkciju `plotpot(naboji, xgranice, ygranice)` koja crta ekvipotencijalne konture u ravnini $z = 0$, gdje su naboji i njihovi položaji zadani kao lista oblika $[(q_1, x_1, y_1, z_1), (q_2, x_2, y_2, z_2), \ldots]$. Upotrijebite tu funkciju da nacrtate ekvipotencijalne konture za naboje `[(-1, 1, 1, 1), (+1, -1, -1, 0.3), (+1, -1, 1, 0.5)]`.
Naputak: Koristite funkciju `contour_plot()` da nacrtate $\phi(x, y, 0)$

5.2.1 Zračenje ubrzavanog naboja

(Donji kod je napravljen za Sage sustav. Možete ga izvršiti ako instalirate Sage ili online na sagemathcloud-u. Dobra je vježba konvertirati to u Jupyter/Python.)

Snaga zračenja po prostornom kutu za točkasti naboj q brzine $\vec{\beta}$ (u jedinicama brzine svjetlosti c) i ubrzanja $\vec{a}$ dana je Poyntingovim vektorom:

$$\frac{dP}{d\Omega} = \frac{q^2 a^2}{16\pi^2 \epsilon_0 c^3} \frac{|\hat{n} \times ((\hat{n} - \vec{\beta}) \times \hat{a})|^2}{(1 - \hat{n} \cdot \vec{\beta})^5} .$$

gdje je $\hat{n}$ jedinični vektor u smjeru promatrača, a $\hat{a}$ jedinični bezdimenzionalni vektor ubrzanja. Cf. npr. Griffits, Eq. (11.72)

```
sage: var('theta phi')
(theta, phi)
```

```
sage: def unitn(theta, phi):
....:     """Unit vector \hat{n} in Cartesian coordinate system."""
....:
....:     return vector((sin(theta)*cos(phi), sin(theta)*sin(phi),
....:                                                 cos(theta)))
```

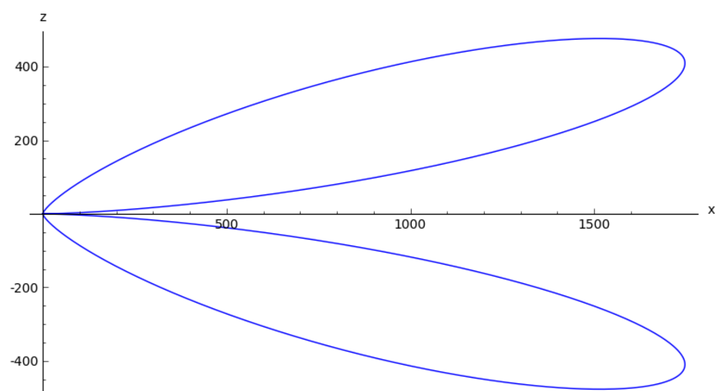
Funkcija poynting() definira samo drugi bezdimenzionalni faktor u gornjem izrazu koji opisuje kutnu ovisnost

```
sage: def poynting(theta, phi, beta, ahat):
....:     """Angular part of Poynting vector."""
....:
....:     nk = unitn(theta, phi)
....:     num = (nk.cross_product((nk-beta).cross_product(ahat))).norm()^2
....:     denom = (1-nk.dot_product(beta))^5
....:     return (num/denom) * nk
```

```
sage: def poynting2d(theta, v, a):
....:     """Projection of Poynting vector on phi=0, i.e x-z plane."""
....:
....:     p = poynting(theta, 0, v, a)
....:     return (p[0], p[2])
```

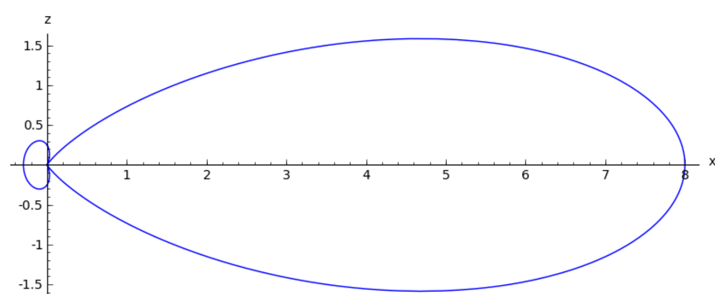
Zračenje kad je akceleracija u smjeru brzine. Cf. Griffits, slika 11.14.

```
sage: parametric_plot(poynting2d(theta, vector((0.9, 0, 0)),
....:     vector((1, 0, 0))), (theta, 0, 2*pi), axes_labels=['x', 'z'])
```



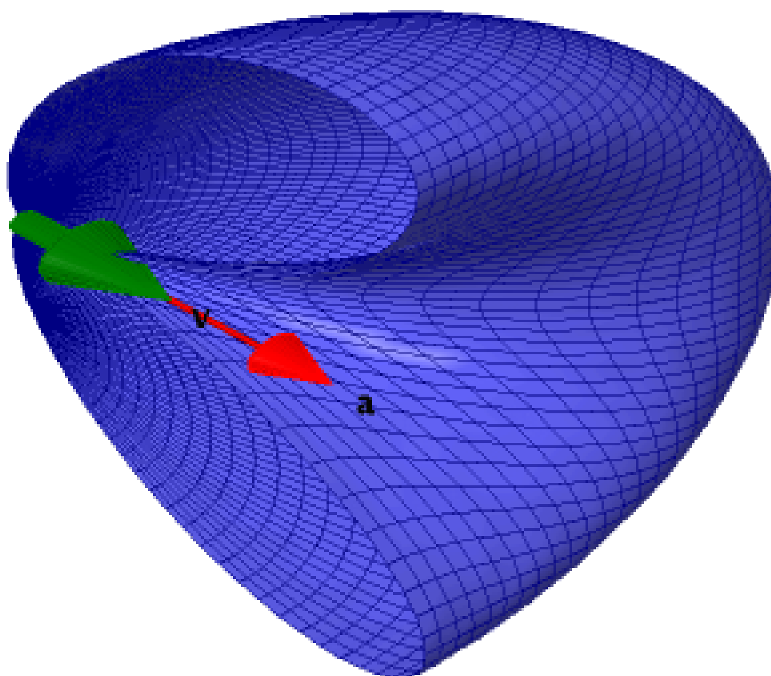
Sinhrotronsko zračenje, kad je akceleracija okomita na brzinu. Cf. Griffits, slika 11.16.

```
sage: parametric_plot(poynting2d(theta, vector((0.5, 0, 0)),
....:    vector((0, 0, 1))), (theta, 0, 2*pi), axes_labels=['x', 'z'])
```




```
sage: def plotField(v=vector((0.5, 0, 0)), a=vector((1, 0, 0))):
....:     """Plot the 3D shape of radiation field (cut along x-z plane)."""
....:
....:     R = parametric_plot3d(poynting(theta, phi, v, a), (theta, 0, pi),
....:          (phi, 0, pi), plot_points=[100,100], frame=False, opacity=0.7)
....:     Av = arrow3d((0,0,0), 3*v, color='green', width=5)
....:     Aa = arrow3d((0,0,0), 3*a, color='red', width=3)
....:     Tv = text3d("v", 3.6*v, size=15)
....:     Ta = text3d("a", 3.3*a, size=50)
....:     return R + Av + Aa +Tv + Ta
```

```
sage: plotField().show(mesh=True)
```



5.2.2 Ukupna snaga zračenja za sinhrotronsko zračenje:

Za elektron u magnetskom polju B, okomitom na njegovu trenutnu brzinu, ubrzanje usljed Lorentzove sile je

$$a = \frac{qc\beta B}{m_e} \; .$$

Nakon uvrštavanja, rezultat je zgodno izraziti preko Thomsonovog udarnog presjeka

$$\sigma_T = \frac{2\mu_0 e^4}{12\pi m_e^2 \epsilon_0 c^2} = 6.652\,\mathrm{m}^2 \; ,$$

i gustoće energije magnetskog polja

$$U_B = \frac{B^2}{2\mu_0} \; ,$$

pa se konačno dobije za ukupnu snagu zračenja:

$$P = \frac{3}{4\pi}\sigma_T c\beta^2 U_B \int d\Omega \frac{|\hat{n} \times ((\vec{n} - \vec{\beta}) \times \hat{a}|^2}{(1 - \hat{n} \cdot \vec{\beta})^5} \; .$$

Iskoristiti ćemo ovaj primjer za demonstraciju računanja s mjernim jedinicama upotrebom SciPy modula constants i Pythonovog modula units.

```
sage: from scipy import constants
sage: c = constants.c * (units.length.meter / units.time.second)
sage: q = units.charge.elementary_charge
sage: sigmaT = constants.physical_constants['Thomson cross section'
....:                                       ][0] * units.length.meter^2
sage: mu0 = constants.mu_0
sage: print "Thomsonov udarni presjek = " + str(sigmaT)
Thomsonov udarni presjek = (6.652458734e-29)*meter^2
```

```
sage: UB(B) = B^2/2/constants.mu_0 * (units.energy.joule/units.length.meter^3)
```

Relevantni integral po prostornom kutu:

```
sage: def Kint(v=vector((0.5, 0, 0)), a=vector((0, 0, 1))):
....:     return integral_numerical(lambda theta: sin(theta) *
....:       integral_numerical(lambda phi: poynting(theta, phi, v,
....:          a).dot_product(unitn(theta, phi)), 0, 2*pi)[0], 0, pi)[0]
```

```
sage: def pwr(beta, B):
....:     return (3/4/pi.n()) * sigmaT * c * beta^2 * UB(B) * Kint(
....:                                                  v=vector((beta,0,0)))
```

```
sage: print "Ukupna snaga zracenja = " + str(pwr(0.5, 1))
Ukupna snaga zracenja = (7.05359483944320e-15)*joule/second
```

Za kontrolu uspoređujemo s formulom

$$P = 2\sigma_T \beta^2 \gamma^2 c U_B$$

```
sage: gamma(beta) = 1/(1-beta^2)
sage: pwr2(beta, B) = 2 * sigmaT * beta^2 * gamma(beta)^2 * c * UB(B)
```

```
sage: pwr2(0.5, 1.)
(7.05359483944320e-15)*joule/second
```

5.3 Termodinamika i statistička fizika

5.3.1 Brownovo gibanje

Zadatak T-1

Konstruirajte funkciju `brown(n)` koja daje listu $[(x_0, y_0), (x_1, y_1), \ldots]$ pozicija čestice koja se giba u ravnini slučajnim gibanjem koje je definirano rekurzijama $x_i = x_{i-1} + r$ i $y_i = y_{i-1} + s$ gdje su r i s slučajni brojevi između -1 i 1. Nacrtajte odgovarajuću putanju čestice.
Naputak:

- Inicijalizirajte listu pozicija tako da sadrži `(0,0)` kao prvi vektor pozicije i putem petlje dodajte u svakom koraku toj listi novi slučajni vektor.

Zadatak T-2

Konstruirajte funkciju `walk1D()` tako da simulira tzv. jednodimenzionalnog nasumičnog šetača. Svaki šetačev korak treba biti iste, jedinične duljine, ulijevo ili udesno. Dakle, umjesto vektora `(r, s)` iz gornjeg Brownovog gibanja trebamo slučajan odabir koraka +1 ili -1. Uvjerite se u ispravnost tvrdnje da je prosječna kvadratna udaljenost nasumičnog jednodimenzionalnog šetača od ishodišta nakon N koraka N, sa standardnom devijacijom $\sqrt{2}N$, dok je prosječna apsolutna udaljenost $\sqrt{2N/\pi}$.

5.4 Kvantna fizika

5.4.1 Pravokutna potencijalna jama

Donji kod je napravljen za Sage sustav. Možete ga izvršiti ako instalirate Sage ili online na sagemathcloud-u. Dobra je vježba konvertirati to u Jupyter/Python.

Isprogramirat ćemo funkciju koja računa kvantnomehaničke valne funkcije i pripadajuće energije vezanih stanja čestice mase m u pravokutnoj potencijalnoj jami dubine V i širine L. Koristimo oznake s Wikipedijine stranice.

```
sage: var('a k m L V E A B G H')
(a, k, m, L, V, E, A, B, G, H)
```

```
sage: import scipy.linalg as la
sage: import numpy as np
sage: import functools
```

Definiramo valnu funkciju za potencijalnu jamu širine L.

```
sage: # components of piecewise wave function
sage: psi1(G, a, x) = G*exp(a*x)
sage: psi2(A, B, k, x) = A*sin(k*x) + B*cos(k*x)
sage: psi3(H, a, x) = H*exp(-a*x)
sage: # and a complete wave function
sage: def psi(x, coefs=(0,0,0,0), wns=(0, 0), L=1, norm=1, shift=0):
....:     """ coefs = (A, B, G, H)
....:         wns = (k, a)
....:     """
....:     if x < -L/2:
....:         psi = psi1(coefs[2], wns[1], x)
....:     elif x > L/2:
....:         psi = psi3(coefs[3], wns[1], x)
....:     else:
....:         psi = psi2(coefs[0], coefs[1], wns[0], x)
....:     return norm*psi + shift
```

Valni brojevi u unutrašnjosti k i izvan jame a.

```
sage: # hbar=1
sage: wns = {k: sqrt(2*m*E), a: sqrt(2*m*(V-E))}
```

Rubni uvjeti na lijevom i desnom rubu jame su da tamo valna funkcija i njena prva derivacija budu neprekidne.

```
sage: boundary_conditions = [
....:     psi1(G, a, -L/2) == psi2(A, B, k, -L/2),
....:     psi3(H, a,  L/2) == psi2(A, B, k,  L/2),
....:     diff(psi1(G, a, x), x).subs(x=-L/2) == diff(psi2(A, B, k, x),
....:                                                 x).subs(x=-L/2),
....:     diff(psi3(H, a, x), x).subs(x= L/2) == diff(psi2(A, B, k, x),
....:                                                 x).subs(x= L/2)
....:                          ]
```

Ovi rubni uvjeti predstavljaju homogeni linearni sustav jednadžbi s nepoznanicama A, B, G i H. Energije vezanih stanja su određene zahtjevom da taj sustav ima rješenje. Konkretno, pretvorit ćemo rubne uvjete

u matričnu jednadžbu oblika

$$M \begin{pmatrix} A \\ B \\ G \\ H \end{pmatrix} = 0$$

a energije su onda dane kao rješenja obične jednadžbe $\det M = 0$. Zbog trenutnog buga u Sageovoj rutini za determinantu matrica 4x4 i većih (trebao bi biti ispravljen u nadolazećoj verziji Sagea) definiramo svoju rutinu za izračun determinante:

```
sage: def mydet(m):
....:     "Evaluates determinant of 4x4 matrix m. (Sage's det() has a bug.)"
....:     d = 0
....:     for c in range(4):
....:         ci = range(4)
....:         ci.pop(c)
....:         sub = m.matrix_from_rows_and_columns([1,2,3], ci)
....:         d += (-1)^(c) * m[0,c] * det(sub)
....:     return d
```

Elemente matrice M dobivamo metodom simboličkog izraza `coef()`, a jednadžbu $\det M = 0$ rješavamo algoritmom u kojem krečemo od liste intervala `[(0,V)]`, koja sadrži u početku samo jedan interval `(0, V)` i postupamo ovako:

1. Maknemo prvi interval `(a, b)` iz liste (metoda `pop()`) i tražimo u njemu nultočku. To može imati dva ishoda

1. Našli smo nultočku (taj ishod je zagarantiran u prvom koraku jer jedno vezano stanje uvijek postoji). Zapišemo nađenu energiju, a listu intervala nadopunimo s dva nova intervala: `(a, E)` i `(E, b)`.

2. Nismo našli nultočku. Ne radimo ništa (interval je već maknut s popisa).

2. Ponavljamo postupak iz točke 1. dok ne iscrpimo sve intervale. (Radi jednoznačnosti, svi intervali gore su u samom kodu zapravo za malu vrijednost `eps` udaljeni od navedenih rubova.)

```
sage: def energies(system = {m: 1/2, L: 1, V: 25}, eps=1e-3):
....:     bcs = [c.subs(wns).subs(system) for c in boundary_conditions]
....:     # Preparing linear system  M*coefs = 0
....:     M = matrix([[(eq.lhs()-eq.rhs()).coefficient(v) for v in (A,B,G,H)]
....:                 for eq in bcs])
....:     dt = mydet(M)
....:     res = []
....:     # Energies of bound states are all solutions of det(M) = 0
....:     ints = [(eps, system[V]-eps)]
....:     while ints:
....:         int = ints.pop(0)
....:         try:
....:             e = find_root(dt, int[0], int[1])
....:             res.append(e)
....:             ints.extend([(int[0], e-eps), (e+eps, int[1])])
....:         except:
....:             pass
....:     res.sort()
....:     return res
```

Za provjeru odredit ćemo energije i odgovarajuće vrijednosti bezdimenzionalne varijable $v = kL/2$ za slučaj sa spomenute stranice na Wikipediji:

```
sage: ens = energies(system = {m: 1/2, L: 1, V: 80}); ens
[6.557920590133058, 25.767519828716722, 55.56613192980969]
```

```
sage: # compare with wikipedia values = 1.28, 2.54, 3.73
sage: [(k*L/2).subs(wns).subs({m: 1/2, L: 1, V: 80}).subs(E=en).n()
....:      for en in ens]
[1.28042186311124, 2.53808588451596, 3.72713468799458]
```

Sada definiramo funkciju koja za datu energiju određuje koeficijente valnih funkcija A, B, G, H. Sustav jednadžbi rubnih uvjeta nije dovoljno određen ($\det M = 0$!). Dodatni uvjet koji bi potpuno odredio sustav je uvjet normalizacije valnih funkcija (integral gustoće vjerojatnosti $|\psi(x)|^2$ po cijelom prostoru mora biti jedan), ali dodavanje tog uvjeta bi učinilo jednadžbe nelinearnima pa ćemo raditi na način da privremeno fiksiramo normalizaciju valne funkcije uvjetom $\psi(L//2) = 1$, što nam fiksira vrijednost koeficijenta H i onda za preostale koeficijente rješavamo nehomogenu linearnu matričnu jednadžbu

$$M_{\rm red} \begin{pmatrix} A \\ B \\ G \end{pmatrix} = F_{\rm red}$$

```
sage: def coefs(en, system = {m: 1/2, L: 1, V: 25}):
....:     """Returns (A, B, G, H) for given energy."""
....:     # Fixing H by normalization psi(L/2)=1
....:     hval = numerical_approx(exp(a*L/2).subs(wns).subs(system).subs(
....:                 {E: en}))
....:     bcred = [c.subs(H=hval).subs(wns).subs(system).subs({E: en})
....:                                  for c in boundary_conditions]
....:     # Preparing linear system  Mred * coefs[:-1] - Fred = 0
....:     Mred = matrix([[(eq.lhs()-eq.rhs()).coeff(v) for v in (A,B,G)]
....:                                  for eq in bcred])
....:     Fred = vector([(eq.lhs()-eq.rhs()).subs({A:0, B:0, G:0})
....:                                  for eq in bcred])
....:     cfs = la.lstsq(Mred, -Fred)[0].tolist()
....:     cfs.append(hval)
....:     return tuple(cfs)
```

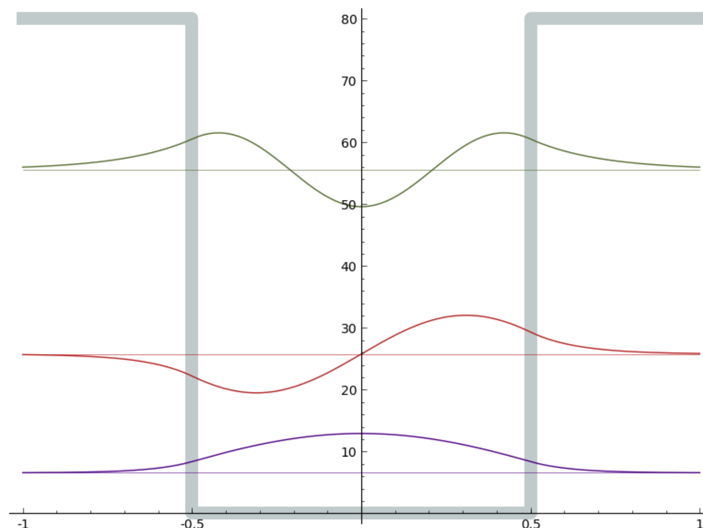
Sada definiramo funkciju koja kombinira gornje funkcije, normalizira dobivene valne funkcije i vraća listu [(E_1, ψ_1), (E_2, ψ_2), ...] gdje su $\psi_i(x)$ funkcije jednog argumenta (x) dobivene parcijalnim izvrijednjavanjem (cf. *currying*) na početku definirane funkcije `psi` za različite vrijednosti njenih opcionalnih argumenata, pomoću modula `functools`. (Normalizirane funkcije još množimo factorom `scale` radi čitlijvijeg crtanja.)

```
sage: def sqwell(system = {m: 1/2, L: 1, V: 25}, scale=5.):
....:     ens = energies(system)
....:     print "Energies = %s" % str(ens)
....:     res = []
....:     for en in ens:
....:         cfs = coefs(en, system)
....:         #print cfs
....:         norm = 1/sqrt(numerical_integral(lambda x: psi(x, cfs,
....:             (k.subs(wns).subs(system).subs(E=en), a.subs(wns).subs(
....:             system).subs(E=en)), L=1)**2, (-oo, oo))[0])
....:         #print en,
....:         fun = functools.partial(psi, coefs=cfs,
....:               wns=(k.subs(wns).subs({m: 1/2, L: 1, V: 25}).subs(E=en),
....:               a.subs(wns).subs(system).subs(E=en)),
```

```
....:                        L=system[L], norm=norm*scale)
....:         #print fun(0)
....:         res.append((en, fun))
....:     return res
```

```
sage: def square_well(system = {m: 1/2, L: 1, V: 25}, span=2, scale=5.):
....:     """span = horizontal span of figure in widhts of well
....:        scale = scaling factor for wavefunctions for visibility
....:     """
....:     colors = ['indigo', 'firebrick', 'darkolivegreen', 'plum']
....:     sol = sqwell(system)
....:     P = line([[-span*system[L]/2, system[V]], [-system[L]/2,
....:           system[V]], [-system[L]/2, 0],
....:             [system[L]/2, 0], [system[L]/2, system[V]],
....:                  [span*system[L]/2, system[V]]],
....:               color='darkslategray', thickness=10, alpha=0.3)
....:     for (en, fun), k in zip(sol,range(len(sol))):
....:         P += line([[-span*system[L]/2, en], [span*system[L]/2, en]],
....:                             color=colors[k], thickness=0.4)
....:         P += plot(lambda x: fun(x, shift=en), (-span*system[L]/2,
....:                        +span*system[L]/2), color=colors[k])
....:     P.show()
```

```
sage: square_well(system = {m: 1/2, L: 1, V: 80})
Energies = [6.557920590133058, 25.767519828716722, 55.56613192980968]
```



5.5 Kozmologija

Donji kod je napravljen za Sage sustav. Možete ga izvršiti ako instalirate Sage ili online na sagemathcloud-u. Dobra je vježba konvertirati to u Jupyter/Python.

Nobelova nagrada za fiziku 2011. godine dodijeljena je Perlmutteru, Schmidtu i Riessu *"za otkriće ubrzanog širenja svemira putem opažanja dalekih supernova"*. U ovom odjeljku ćemo prikazati taj rezultat.

Udaljenost (luminosity distance) D_L (u megaparsecima Mpc) do supernove s prividnim sjajem m (mjeri se izravno) i apsolutnim sjajem M (poznat je za supernove tipa Ia) je dana formulom $\mu = m - M =$

$5 \log_{10} D_L + 25$. Na internet adresi http://supernova.lbl.gov/Union/ nalaze se podaci o nekoliko stotina supernovih organiziranih u tablici sa stupcima:

1. ime supernove
2. z - crveni pomak
3. μ - atenuacija sjaja supernove
4. $\Delta\mu$ - eksperimentalna neodređenost (greška) od μ

Učitavamo ih koristeći NumPy funkciju `loadtxt()` koja numeričke podatke organizirane u stupce obične datoteke pretvara u NumPy listu.

```
sage: import matplotlib.pyplot as plt
sage: import numpy as np
sage: import scipy
sage: import scipy.stats
sage: import urllib
```

```
sage: sns = urllib.urlopen(
....:  'http://supernova.lbl.gov/Union/figures/SCPUnion2_mu_vs_z.txt')
sage: dat = np.loadtxt(sns, usecols=(1,2,3)); dat.shape
(557, 3)
```

```
sage: dat[:2,:]
array([[  2.84880000e-02,   3.53355511e+01,   2.26143926e-01],
       [  5.00430000e-02,   3.66754416e+01,   1.67114252e-01]])
```

Pretvaramo sada mjerene podatke iz μ i $\Delta\mu$ u podatke za D_L i ΔD_L pazeći na ispravnu propagaciju greške.

```
sage: var('mu DL')
(mu, DL)
sage: DL(mu) = DL.subs(solve(mu == 5*log(DL, base=10) + 25, DL)[0]); DL
mu |--> 1/100000*e^(1/5*mu*log(10))
```

```
sage: DLdat = np.array([(z, DL(valmu).n(), diff(DL)(valmu).n()*errmu)
....:                            for z,valmu,errmu in dat]); DLdat.shape
(557, 3)
```

Definiramo funkcije za prilagodbu, poznate iz odjeljka o *prilagodbi funkcija*, a dodajemo i dio za ocjenu dobrote prilagodba, vidi odjeljak o *testiranju hipoteze*.

```
sage: def dist(p, fun, data):
....:     return np.array([(y - fun(p, x))/err for (x,y,err) in data])
....:
sage: def chisq(p, fun, data):
....:     return sum( dist(p, fun, data)^2 )
....:
sage: def leastsqfit(fcn, init, data):
....:     """Fit function fcn, starting from init initial values to data."""
....:     p_final, cov_x, infodict, msg, ier = scipy.optimize.minpack.leastsq(
....:                           dist, init,  (fcn, data), full_output=True)
....:     parerrs = sqrt(scipy.diagonal(cov_x))
....:     for k in range(len(init)):
....:         print "p[%d] = %.3f +- %.3f" % (k, p_final[k], parerrs[k])
....:     chi2 = chisq(p_final, fcn, data)
....:     df = len(data)-len(init)
```

```
....:     print "chi^2/d.o.f = %.1f/%i  (p-value = %.4f)" % (chi2, df,
....:       scipy.stats.chisqprob(chi2, df))
....:     return list(p_final)
```

5.5.1 Određivanje H_0 iz podataka o bliskim supernovama

Za male crvene pomake, ovisnost udaljenosti i crvenog pomaka dana je Hubbleovim zakonom

$$H_0 D_L = cz$$

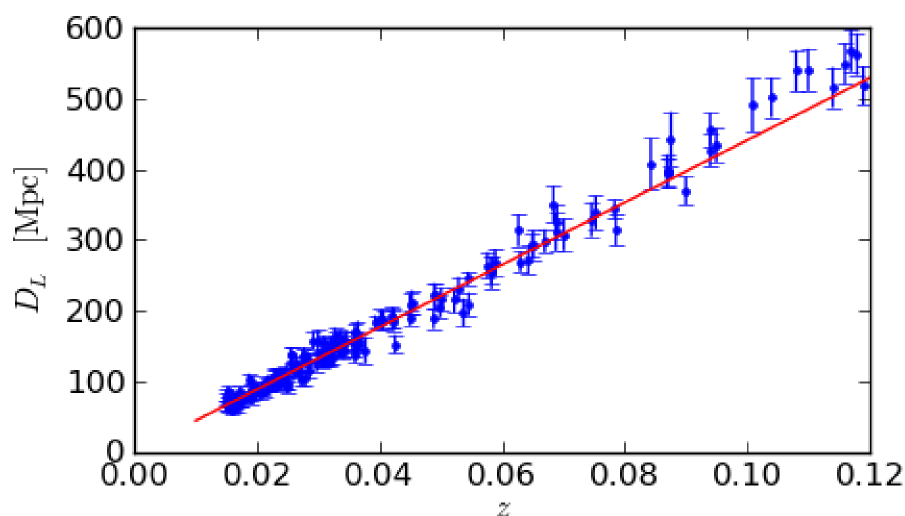
gdje je c brzina svjetlosti, a H_0 Hubbleova konstanta. Određujemo Hubbleovu konstantu (u jedinicama km/s/Mpc) prilagodbom Hubbleovog zakona na podskup mjerenja za koja je $z < 0.12$.

```
sage: lowz = np.array([pt for pt in DLdat if pt[0]<0.12]); lowz.shape
(174, 3)
```

```
sage: c=3.e5
sage: def linfun(p, z):
....:     return c*z/p[0]
```

```
sage: H0, = leastsqfit(linfun, (60,), lowz)
p[0] = 68.012 +- 0.385
chi^2/d.o.f = 179.1/173  (p-value = 0.3585)
sage: print "\nHubbleova constanta H0 = %.1f  km/s/Mpc" % H0
<BLANKLINE>
Hubbleova constanta H0 = 68.0  km/s/Mpc
```

```
sage: fig, ax = plt.subplots(figsize=[5,3])
sage: ax.errorbar(lowz[:,0], lowz[:,1], yerr=lowz[:,2], marker='.',
....:                                                   linestyle='None')
sage: zvals = np.linspace(0.01, 0.12)
sage: ax.plot(zvals, [linfun([H0], z) for z in zvals],'r-')
sage: ax.set_xlabel('$z$')
sage: ax.set_ylabel('$D_L \; {\\rm [Mpc]}$')
sage: fig.tight_layout()
sage: fig.savefig('fig')
```



5.5.2 Određivanje ubrzanja ekspanzije svemira

Za veće crvene pomake, Hubbleov zakon se modificira i u prvoj aproksimaciji se može pisati kao

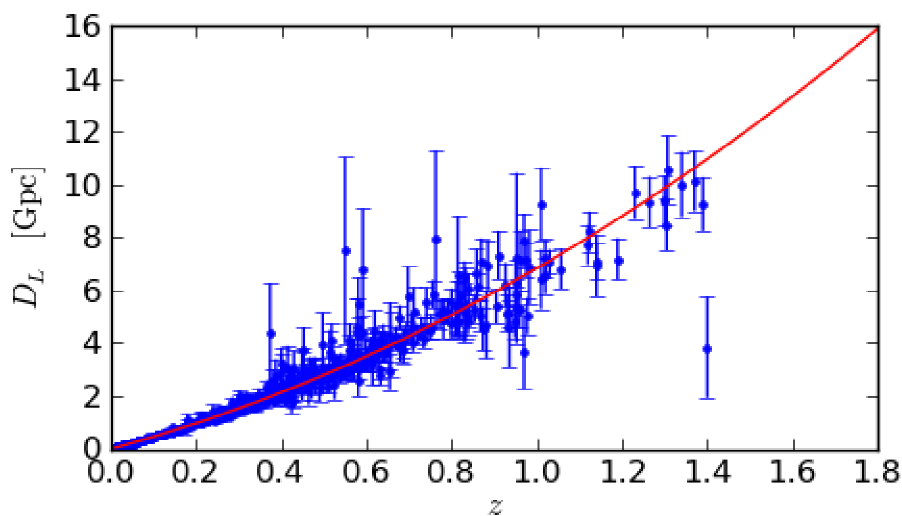
$$H_0 D_L = cz\big(1 + (1 - q_0)\frac{z}{2}\big)$$

gdje je q_0 parametar deceleracije. Definiran je tako da pozitivni q_0 odgovara svemiru čija se brzina širenja usporava (kako se očekivalo prije ovih mjerenja). Određujemo taj parametar prilagodbom ove formule na cijeli skup mjerenja.

```
sage: def qfun(p, z):
....:     return c*z/p[0] * (1  +  0.5*(1 - p[1])*z)
```

```
sage: qpars = leastsqfit(qfun, (50, 0.1), DLdat)
p[0] = 69.184 +- 0.368
p[1] = -0.153 +- 0.039
chi^2/d.o.f = 553.7/555  (p-value = 0.5081)
```

```
sage: fig, ax = plt.subplots(figsize=[5,3])
sage: ax.errorbar(DLdat[:,0], DLdat[:,1]/1e3, yerr=DLdat[:,2]/1e3,
....:                                     marker='.', linestyle='None')
sage: zvals = np.linspace(0.01, 1.8)
sage: ax.plot(zvals, qfun(qpars, zvals)/1e3,'r')
sage: ax.set_xlabel('$z$')
sage: ax.set_ylabel('$D_L \; {\\rm [Gpc]}$')
sage: fig.tight_layout()
sage: fig.savefig('fig')
```



Iako iz mjernih podataka nije očito da postoji odstupanje od linearnog Hubbleovog zakona, podataka ima dovoljno da statistička analiza nedvosmisleno pokazuje da je parametar deceleracije negativan i različit od nule:

$$q_0 = -0.15 \pm 0.04$$

5.5.3 Određivanje gustoće materije i tamne energije

Gornja formula ne vrijedi za $z \approx 1$, dakle ovakav račun nije sasvim korektan i može se rabiti samo u pedagoške svrhe. Ispravan pristup zahtijeva integraciju propagacije zrake svjetlosti od supernove do

promatrača kroz svemir u širenju. Ta propagacija je određena sastavom svemira (obična i tamna materija različito djeluju od tzv. tamne energije koja uzrokuje ubrzanje širenja svemira). Hubbleov zakon postaje

$$H_0 D_L = c(1+z) Z(z, \Omega_M, \Omega_\Lambda)$$

$$Z(z, \Omega_M, \Omega_\Lambda) = \int_{1/(1+z)}^{1} \frac{da}{a\sqrt{X(a, \Omega_M, \Omega_\Lambda)}}$$

$$X(a, \Omega_M, \Omega_\Lambda) = \frac{\Omega_M}{a} + \Omega_\Lambda a^2$$

$$\Omega_M + \Omega_\Lambda = 1$$

gdje je Ω_M udio materije (obične i tamne), a Ω_Λ udio tamne energije u ukupnoj energiji svemira.Još općenitije formule, koje ne rade pretpostavku $\Omega_M + \Omega_\Lambda = 1$, mogu se naći na stranicama Neda Wrighta.

Provjeravamo egzaktne formule razvojem u red po z:

```
sage: var('z a Om Ov Z X q0'); assume(z>0, a>0, Om>0, Ov>0);
(z, a, Om, Ov, Z, X, q0)
sage: X(a, Om, Ov) = Om/a + Ov*a**2 + (1-Om-Ov)
sage: DLH0 = taylor(((1+z)^2 * (Z/(1+z)) * (1 +(1-Om-Ov)*Z^2/6)).subs(
....:           Z==taylor(integral(1/(a*sqrt(X(a, Om, Ov))),
....:                            a, 1/(1+z), 1), z, 0, 2)), z, 0, 2)
sage: solve(DLH0 == z*(1  +  (1 - q0)*z/2), q0)[0]
q0 == 1/2*Om - Ov
```

Ova korespondencija s parametrom deceleracije se slaže s (Peacock 3.34).

```
sage: def J(x):
....:     if x<0:
....:         return sin(sqrt(-x))/sqrt(-x)
....:     elif x==0:
....:         return 1
....:     else:
....:         return sinh(sqrt(x))/sqrt(x)
```

```
sage: def Xfun(a, Om, Ov):
....:     return Om/a + Ov*a**2 + (1-Om-Ov)
```

```
sage: def Zfun(z, Om, Ov):
....:     return numerical_integral(lambda a: 1/(a*sqrt(X(a, Om, Ov))),
....:                                                     1/(1+z), 1)[0]
```

```
sage: def DLfun(p, z):
....:     """DL(z) with pars p=[H0, Omega_m, Omega_Lambda]."""
....:     H0 = p[0]
....:     Om = p[1]
....:     Ov = p[2]
....:     Otot = Om + Ov
....:     Zval = Zfun(z, Om, Ov)
....:     return c/H0 * (1+z) * Zval * J((1-Otot)*Zval**2)
```

```
sage: def DLfunflat(p, z):
....:     """DL(z) with pars p=[H0, Omega_Lambda]. Flat universe."""
....:     H0 = p[0]
....:     Ov = p[1]
....:     Om = 1-Ov
....:     Zval = Zfun(z, Om, Ov)
....:     return c/H0 * (1+z) * Zval
```

Prilagodbom parametara H_0, Ω_M i Ω_Λ određujemo udjele pojedinih komponenti svemira

```
sage: fullpars = leastsqfit(DLfun, (70, 0.3, 0.), DLdat) # long time
p[0] = 70.717 +- 0.466
p[1] = 0.368 +- 0.075
p[2] = 0.834 +- 0.122
chi^2/d.o.f = 527.7/554  (p-value = 0.7833)
```

Ukoliko pak fiksiramo ravnu geometriju ($\Omega_M + \Omega_\Lambda = 1$) dobivamo uobičajenu jednu trećinu materije i dvije trečine tamne energije:

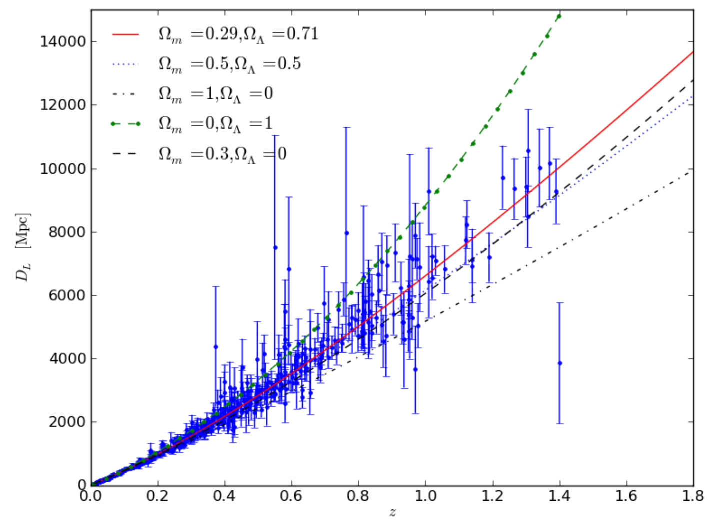
```
sage: flatpars = leastsqfit(DLfunflat, (70, 0.3), DLdat) # long time
sage: print "Om = 1-Ov = %8.3f" % (1-flatpars[1],)
p[0] = 70.412 +- 0.363
p[1] = 0.708 +- 0.021
chi^2/d.o.f = 528.7/555  (p-value = 0.7827)
Om = 1-Ov =    0.292
```

Dakle,

$$\Omega_M = 0.29$$
$$\Omega_\Lambda = 0.71$$

Crtamo ovaj rezultat (crvena linija) zajedno s nekim drugim karakterističnim scenarijima.

```
sage: fig, ax = plt.subplots(figsize=[8,6])
sage: ax.errorbar(DLdat[:,0], DLdat[:,1], yerr=DLdat[:,2], marker='.',
....:             linestyle='None')
sage: zvals = np.linspace(0.01, 1.8)
sage: ax.plot(zvals, [DLfunflat(flatpars, z) for z in zvals],'r-',
....:                  label='$\\Omega_m=0.29, \\Omega_\\Lambda=0.71$')
sage: ax.plot(zvals, [DLfun([H0, 0.5, 0.5], z) for z in zvals],'b:',
....:                     label='$\\Omega_m=0.5, \\Omega_\\Lambda=0.5$')
sage: ax.plot(zvals, [DLfun([H0, 1, 0], z) for z in zvals],'k-.',
....:                         label='$\\Omega_m=1, \\Omega_\\Lambda=0$')
sage: ax.plot(zvals, [DLfun([H0, 0, 1], z) for z in zvals],'g--.',
....:                         label='$\\Omega_m=0, \\Omega_\\Lambda=1$')
sage: ax.plot(zvals, [DLfun([H0, 0.3, 0.], z) for z in zvals],'k--',
....:                       label='$\\Omega_m=0.3, \\Omega_\\Lambda=0$')
sage: ax.set_ylim(0, 1.5e4)
sage: ax.set_xlabel('$z$')
sage: ax.set_ylabel('$D_L \; {\\rm [Mpc]}$')
sage: ax.legend(loc='upper left').draw_frame(0)
sage: fig.tight_layout()
sage: fig.savefig('fig')
```


$\Omega_m = 0.29, \Omega_\Lambda = 0.71$
$\Omega_m = 0.5, \Omega_\Lambda = 0.5$
$\Omega_m = 1, \Omega_\Lambda = 0$
$\Omega_m = 0, \Omega_\Lambda = 1$
$\Omega_m = 0.3, \Omega_\Lambda = 0$
D_L [Mpc]
z
0
2000
4000
6000
8000
10000
12000
14000
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
1.8

Dodaci

Donji kod je napravljen za Sage sustav. Možete ga izvršiti ako instalirate Sage ili online na sagemathcloud-u. Dobra je vježba konvertirati to u Jupyter/Python.

6.1 Testiranje hipoteze

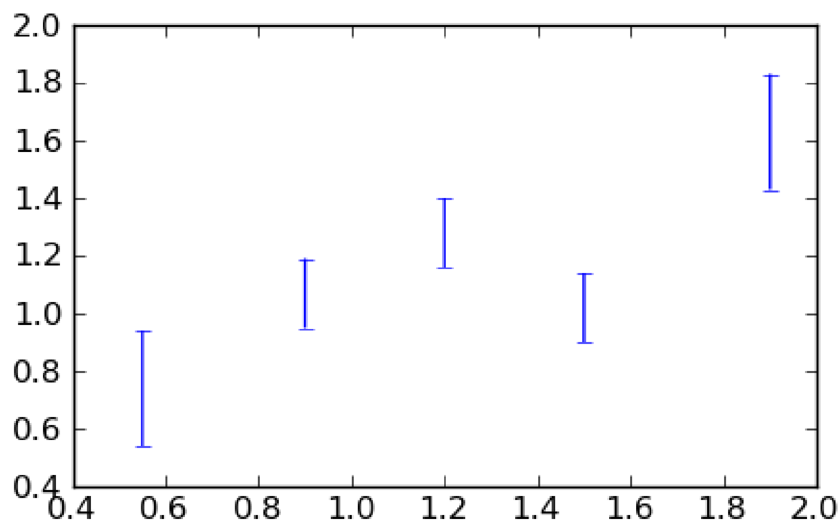
Često je potrebno kvantificirati dobrotu neke prilagodbe te odrediti da li predloženi teorijski model dobro opisuje mjerenja. Uzmimo slijedeći niz mjerenja s greškama $(x_i, y_i, \sigma_i \equiv \Delta y_i)$:

```
sage: errdata = [[0.55, 0.74, 0.20], [0.9, 1.07, 0.12],
....:       [1.2, 1.28, 0.12], [1.5, 1.02, 0.12], [1.9, 1.63, 0.20]]
sage: xs = [x for x,y,err in errdata]
sage: ys = [y for x,y,err in errdata]
sage: errs = [err for x,y,err in errdata]
```

Za crtanje točaka s greškama (*errorbars*), koristimo Matplotlib funkciju `errorbar()` gdje greške idu u opcionalni argument `yerr`:

```
sage: import matplotlib.pyplot as plt
sage: import numpy as np
sage: fig, ax = plt.subplots(figsize=[5,3])
```

```
sage: ax.errorbar(xs, ys, yerr=errs, linestyle='None')
sage: fig.savefig('fig')
```

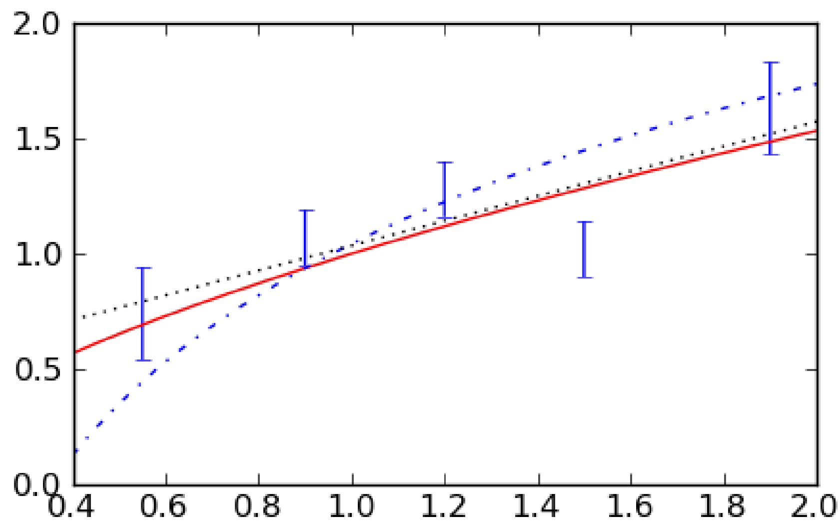


Promotrimo tri moguća teorijska opisa ovih mjerenja: potenciju x^a, logaritam $\log(ax)$ i obični pravac $a + bx$ i odredimo parametre prilagodbom pomoću funkcije *find_fit()*. (Zasad zanemarimo različite greške pojedinih mjerenja.)

```
sage: var('a, b')
(a, b)
sage: powmodel(x) = x^a
sage: logmodel(x) = log(a*x)
sage: linmodel(x) = a*x+b
```

```
sage: powfit=find_fit([[x, y] for x,y,err in errdata], powmodel,
....:                             solution_dict=True); print powfit
{a: 0.6166290268412898}
sage: logfit=find_fit([[x, y] for x,y,err in errdata], logmodel,
....:                             solution_dict=True); print logfit
{a: 2.836898540272243}
sage: linfit=find_fit([[x, y] for x,y,err in errdata], linmodel,
....:                                   solution_dict=True); linfit
{b: 0.49690475972372317, a: 0.5380952396683008}
```

```
sage: xvals = np.linspace(0.4, 2)
sage: ax.plot(xvals, [powmodel(x).subs(powfit).n() for x in xvals],'r')
sage: ax.plot(xvals, [logmodel(x).subs(logfit).n() for x in xvals],'b-.')
sage: ax.plot(xvals, [linmodel(x).subs(linfit).n() for x in xvals],
....:                                   color='black', linestyle=':')
sage: fig.savefig('fig')
```



Teško je "od oka" procijeniti koji je opis najbolji, a još teže koji opisi su prihvatljivi a koji nisu. Standardna mjera dobrote fita je

$$\chi^2 = \sum_i \frac{(y_i - f(x_i))^2}{\sigma_i^2}$$

gdje su σ_i greške mjerenja. (Usput, kad je riječ o mjerenju frekvencija onda je greška dana kao $\sigma_i = \sqrt{y_i}$.). Dobre su prilagodbe s $\chi^2 \approx df$, gdje je df broj stupnjeva slobode (broj mjerenja minus broj slobodnih parametara funkcije koju prilagođavamo). Za točnije kvantificiranje dobrote prilagodbe gleda se integral statističke χ^2 raspodjele s df stupnjeva slobode od izračunatog χ^2 do beskonačnosti koji se često zove p-vrijednost i koji ne bi trebao biti manji od 0.1 ili barem 0.01 ili hipotezu treba odbaciti. Taj integral je implementiran kao funkcija `scipy.stats.chisqprob(chisq, df)`. (Inače, za razliku od ovog primjera gdje tražimo statističku podršku hipotezi da funkcije opisuju mjerenja, često smo u obrnutoj situaciji gdje testiramo tzv. nul-hipotezu. Tu se npr. pitamo koja je vjerojatnost nekih mjerenja ukoliko neka čestica ne postoji ili ukoliko neki neki lijek ne djeljuje. Tada je mala p-vrijednost "poželjna" jer predstavlja statističku podršku postojanju te čestice ili dobrom djelovanju lijeka.)

Definirajmo tri funkcije koje prilagođavamo ...

```
sage: def powfun(p, x):
....:     return x^p[0]
....:
sage: def logfun(p, x):
....:     return log(p[0]*x)
....:
sage: def linfun(p, x):
....:     return p[1]*x+p[0]
```

... i odgovarajuću funkciju udaljenosti tj. odstupanja (čije kvadrate će minimizirati `leastsq()`). Tu trebamo staviti i težinski faktor $1/\sigma_i$ za svaku točku kako bi točke s manjom greškom više utjecale na prilagodbu. Ovdje ćemo dodati još jedan argument putem kojeg ćemo prosljeđivati jednu od gornje tri funkcije:

```
sage: def dist(p, fun, data):
....:     return np.array([(y - fun(p, x))/err for (x,y,err) in data])
```

Definirajmo i odgovarajući χ^2:

```
sage: def chisq(p, fun, data):
....:     return sum( dist(p, fun, data)^2 )
```

Nakon minimizacije ćemo testirati i zastavicu `ier` tako da ukoliko `leastsq()` nije uspješna dobijemo odgovarajuću poruku o grešci:

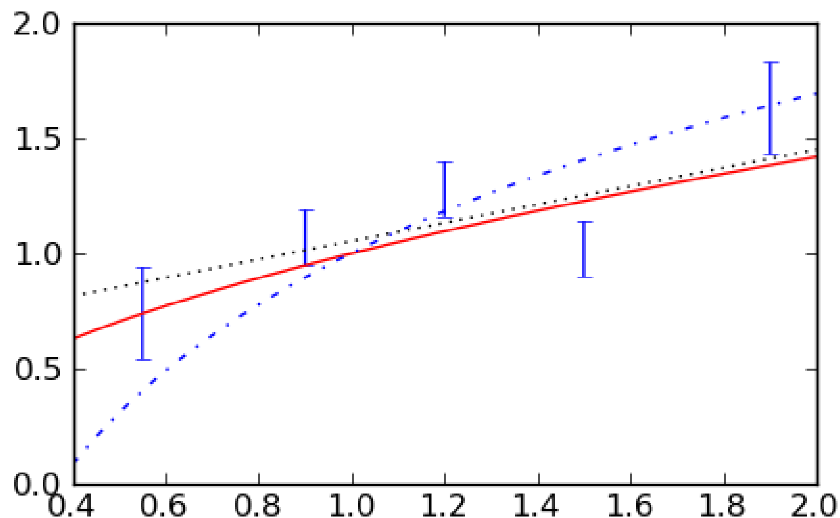
```
sage: import scipy
sage: import scipy.stats
sage: fitfun = powfun
sage: ppow_final, cov_x, infodict, msg, ier = scipy.optimize.minpack.leastsq(
....:                  dist, [1.],  (fitfun, errdata), full_output=True)
sage: if ier not in (1,2,3,4):
....:     print " -----> No fit! <--------"
....:     print msg
....: else:
....:     print "a =%8.4f +- %.4f" % (ppow_final, sqrt(cov_x[0,0]))
....:     chi2 = chisq([ppow_final], fitfun, errdata)
....:     df = len(errdata)-1
....:     print "chi^2 = %.3f" % chi2
....:     print "df = %i" % df
....:     print "p-vrijednost =  %.4f" % scipy.stats.chisqprob(chi2, df)
a =  0.5055 +- 0.1486
chi^2 = 7.880
df = 4
p-vrijednost =  0.0961
```

```
sage: fitfun = logfun
sage: plog_final, cov_x, infodict, msg, ier = scipy.optimize.minpack.leastsq(
....:                 dist, [1.],  (fitfun, errdata), full_output=True)
sage: if ier not in (1,2,3,4):
....:     print " -----> No fit! <--------"
....:     print msg
....: else:
....:     print "a =%8.4f +- %.4f" % (plog_final, sqrt(cov_x[0,0]))
....:     chi2 = chisq([plog_final], fitfun, errdata)
....:     df = len(errdata)-1
....:     print "chi^2 = %.3f" % chi2
....:     print "df = %i" % df
....:     print "p-vrijednost =  %.4f" % scipy.stats.chisqprob(chi2, df)
a =  2.7219 +- 0.1693
chi^2 = 15.973
df = 4
p-vrijednost =  0.0031
```

```
sage: fitfun = linfun
sage: plin_final, cov_x, infodict, msg, ier = scipy.optimize.minpack.leastsq(
....:           dist, [1., 1.],  (fitfun, errdata), full_output=True)
sage: if ier not in (1,2,3,4):
....:     print " -----> No fit! <--------"
....:     print msg
....: else:
....:     parerrs = sqrt(scipy.diagonal(cov_x))
....:     print "p[0] =%8.3f +- %.4f" % (plin_final[0], parerrs[0])
....:     print "p[1] =%8.3f +- %.4f" % (plin_final[1], parerrs[1])
....:     chi2 = chisq(plin_final, fitfun, errdata)
....:     df = len(errdata)-2    # dva parametra
....:     print "chi^2 = %.3f" % chi2
```

```
....:     print "df = %i" % df
....:     print "p-vrijednost =  %.4f" % scipy.stats.chisqprob(chi2, df)
p[0] =   0.656 +- 0.2121
p[1] =   0.398 +- 0.1683
chi^2 = 7.116
df = 3
p-vrijednost =  0.0683
```

```
sage: fig, ax = plt.subplots(figsize=[5,3])
sage: ax.errorbar(xs, ys, yerr=errs, linestyle='None')
sage: xvals = np.linspace(0.4, 2)
sage: ax.plot(xvals, [powfun([ppow_final], x) for x in xvals],'r')
sage: ax.plot(xvals, [logfun([plog_final], x) for x in xvals],'b-.')
sage: ax.plot(xvals, [linfun(plin_final, x) for x in xvals], color='black',
....:             linestyle=':')
sage: fig.savefig('fig')
```



Ova analiza sugerira da možemo odbaciti logaritamski model, dok su druga dva modela, u nedostaku boljih, prihvatljiva.

Zadatak 4

Prilagodite funkciju $f(x) = ax$ donjim podacima. Odredite parametar a i njegovu grešku te ocijenite dobrotu fita.

```
sage: xs2, ys2, errs2 = [range(2,25,2), [5.3,14.4,20.7,30.1,35.0,41.3,52.7,
....:                   55.7,63.0,72.1,80.5,87.9], [1.5 for k in range(2,25,2)]]
sage: errdata2 = zip(xs2, ys2, errs2)
```

6.2 Funkcionalno programiranje

Isti matematički problem je često moguće riješiti na različite načine, putem algoritama iza kojih stoje sasvim različiti načini razmišljanja. Razložimo to na primjeru funkcije faktorijel.

```
sage: factorial(7)
5040
```

Klasični način programiranja, upotrebljavan od dana prvih računala, je tzv. proceduralno (ili imperativno) programiranje kod kojeg na program gledamo kao na niz naredbi koje obično postupno mijenjaju vrijednosti nekih varijabli pohranjenih u memoriji računala:

```
sage: def factorialProc(n):
....:     fac = 1
....:     i = 1
....:     while i<n:
....:         i = i + 1
....:         fac = fac * i
....:     return fac
sage: factorialProc(7)
5040
```

Ovdje je očito riječ o standardnom algoritmu kakvog bismo isprogramirali u bilo kojem proceduralnom jeziku poput Fortrana ili C-a: Imamo petlju koja se prolazi n puta i svaki puta množimo rezultat prošlog prolaza sa za 1 većim brojem.

Međutim, postoje i drugačiji pristupi. Tako je kod funkcionalnog programiranja naglasak na izvrijednjavanju funkcija tj. primjeni funkcija na izraze. Faktorijel prirodno zamišljamo kao funkciju koja daje "umnožak svih brojeva od 1 do zadanog broja".

Kao prvo, treba nam modul `operator` koji implementira funkcije koje odgovaraju standardnim matematičkim operacijama *, +, -,...

```
sage: import operator
sage: (operator.mul(2,3), operator.add(2,3))
(6, 5)
```

Zatim koristimo funkciju `reduce()`, koja kumulativno primjenjuje prvi argument (koji mora biti funkcija) na elemente drugog argumenta:

```
sage: f = function('f')
sage: reduce(f, range(1,5))
f(f(f(1, 2), 3), 4)
```

```
sage: reduce(operator.mul, range(1,8))
5040
```

Primijetite da nam ovdje nisu trebale pomoćne varijable (poput `i` i `fac` iz `factorialProc`), ali nam je trebalo više memorije jer smo trebali pohraniti čitavu listu `[1, 2, ...]` koju daje `range()`.

Funkcionalno programiranje je stil programiranja koji stavlja naglasak na izvrijednjavanje izraza, a ne na sukcesivno izvršavanje komandi. Kod čistih funkcionalnih programa obično nema pomoćnih varijabli i nepotrebnih popratnih efekata izvrijednjavanja funkcija. U tom smislu funkcionalno programiranje je pomalo slično radu s tabličnim kalkulatorom (npr. MS Excel): redosljed izračunavanja ćelija je nebitan tj. očekujemo automatsku konzistenciju. Isto, vrijednosti ćelija su dane izrazima, a ne slijedovima komandi. (Misli se na Excel, a ne Sage ćelije.)

(Više informacija o funkcionalnom programiranju nudi Functional Programming FAQ ili WWW stranice Haskell funkcionalnog jezika.)

Takav stil programiranja često rabi nekoliko specijalnih funkcija (mogli bismo ih zvati "meta-funkcije") čija je uloga upravljanje primjenivanjem drugih funkcija koje dolaze kao argumenti ovih meta-funkcija. Možda najvažnija takva funkcija je `map()` koja distribuira (*mapira*) funkciju po elementima neke liste:

```
sage: f = function('f')
sage: map(f, range(4))
[f(0), f(1), f(2), f(3)]
```

Ukoliko funkcija prima više argumenata, može se mapirati na više listi:

```
sage: map(f, range(4), range(3,7))
[f(0, 3), f(1, 4), f(2, 5), f(3, 6)]
```

Recimo da sad želimo ispisati tablicu s nizom prirodnih brojeva i njihovih faktorijela. Možemo prvo postupiti tako da prvo definiramo pomoćnu funkciju koja generira jedan red tablice i onda je pomoću `map()` primijenimo na niz brojeva:

```
sage: def auxfun(k):
....:     return (k, factorial(k))
```

```
sage: map(auxfun, range(7))
[(0, 1), (1, 1), (2, 2), (3, 6), (4, 24), (5, 120), (6, 720)]
```

Međutim, baš kao i pomoćne varijable, tako ni pomoćne funkcije nisu u duhu funkcionalnog programiranja. Zato postoje tzv. *lambda-funkcije* koje su bezimene i definiraju se i upotrebljavaju na slijedeći način:

```
sage: map(lambda k: (k, factorial(k)), range(7))
[(0, 1), (1, 1), (2, 2), (3, 6), (4, 24), (5, 120), (6, 720)]
```

```
sage: matrix(_)  # čitljiviji ispis bez uptrebe formatirajućih stringova
[  0   1]
[  1   1]
[  2   2]
[  3   6]
[  4  24]
[  5 120]
[  6 720]
```

Treba uočiti kako je često upotrebu funkcije `map()` i lambda-funkcija moguće izbjeći korištenjem *obuhvaćanja liste*. Npr, gornji primjer se elegantnije realizira ovako:

```
sage: [(k, factorial(k)) for k in range(7)]
[(0, 1), (1, 1), (2, 2), (3, 6), (4, 24), (5, 120), (6, 720)]
```

Kako su Sage/Python funkcije punopravni objekti, možemo i sami definirati funkcije koje primaju druge funkcije kao argumente. Evo funkcije koja implementira kompoziciju funkcija:

```
sage: def compose(f, n, x):
....:     """Uzastopno komponiranje funkcije sa samom sobom n puta."""
....:     if n <= 0:
....:         return x
....:     x = f(x)
....:     for i in range(n-1):
....:         x = f(x)
....:     return x
```

Npr. tzv. zlatni omjer $(\sqrt{5}+1)/2$ se može izraziti kao beskonačni razlomak

$$\frac{\sqrt{5}+1}{2} = 1 + \frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{\cdots}}}} = 1.618034\ldots$$

Takav razlomak možemo izvrijedniti na slijedeći način:

```
sage: gr = compose(lambda x: 1+1/x, 5, x); gr
1/(1/(1/(1/(1/x + 1) + 1) + 1) + 1) + 1
```

```
sage: gr.subs(x=1).n()
1.62500000000000
```

Ili, uz povećanje točnosti:

```
sage: numerical_approx((1+sqrt(5))/2)
1.61803398874989
sage: compose(lambda x: 1+1/x, 32, x).subs(x=1).n()
1.61803398874986
```

6.2.1 Primjer razvoja funkcionalnog programa

Promotrimo razvoj programa koji ispisuje tablicu frekvencija pojavljivanja elemenata u listi. Načinimo prvo jednostavnu testnu listu

```
sage: lst = ['a', 'c','b', 'a', 'c', 'a', 'd', 'b', 'c','d', 'c']
```

Metoda liste koja daje broj pojavljivanja nekog elementa je `count()`

```
sage: lst.count('c')
4
```

Da bismo ispisivali tablicu trebamo listu oblika `[(element1, frekvencija element1), (element2, frekvencija elementa2),...]`. Element te liste (red tablice) se može dobiti ovako:

```
sage: def el(x):
....:     return (x, lst.count(x))
sage: el('c')
('c', 4)
```

Tablicu frekvencija za različite elemente dobijemo korištenjem lambda-funkcije analogne `el()` i njenom primjenom pomoću funkcije `map()` na popis elemenata:

```
sage: map(el, ['a', 'b', 'c', 'd'])
[('a', 3), ('b', 2), ('c', 4), ('d', 2)]
```

```
sage: map(lambda x: (x, lst.count(x)), ['a', 'b', 'c', 'd'])
[('a', 3), ('b', 2), ('c', 4), ('d', 2)]
```

To je sad to, jedino još želimo izbjeći da sami moramo ručno identificirati koji se sve elementi pojavljuju u listi. To nam može raditi funkcija `uniq()` koja izbacuje duplikate iz liste:

```
sage: res = map(lambda x: (x, lst.count(x)), uniq(lst)); res
[('a', 3), ('b', 2), ('c', 4), ('d', 2)]
```

Kao zadnju stvar, poželjno je listu sortirati po frekvencijama. Funkcija `sorted()` je već definirana tako da zna sortirati liste različitih objekata, no ovdje bi po defaultu sortirala parove po prvom elementu i to po abecednom redu, ...

```
sage: sorted(res)
[('a', 3), ('b', 2), ('c', 4), ('d', 2)]
```

... dok nama treba sortiranje po drugom elementu i to po veličini. Stoga moramo putem opcionalnog argumenta `key` funkciji `sorted()` proslijediti funkciju koja će vraćati onaj dio elementa liste po kojem želimo sortiranje:

```
sage: def keyfun(item):
....:     """Return last element of item."""
....:     return item[-1]
```

```
sage: sorted(res, key=keyfun, reverse=True)
[('c', 4), ('a', 3), ('b', 2), ('d', 2)]
```

(Opcionalni argument reverse daje silazno sortiranje umjesto defaultnog uzlaznog.)

Naravno, i ovu pomoćnu funkciju `keyfun()` možemo zamijeniti lambda funkcijom:

```
sage: sorted(res, key=lambda it: it[-1], reverse=True)
[('c', 4), ('a', 3), ('b', 2), ('d', 2)]
```

I to je to. Sad definiramo kompletnu funkciju:

```
sage: def frequencies(lst):
....:     """Elements of list lst with their frequencies."""
....:     return sorted(map(lambda x: (x, lst.count(x)), set(lst)),
....:            key=lambda it: it[-1], reverse=True)
```

```
sage: for it in frequencies(lst):
....:     print "%s: %d" % it
c: 4
a: 3
b: 2
d: 2
```

Primijenimo to na frekvenciju pojavljivanja slova u Shakespeareovom Mletačkom trgovcu (zapravo koristimo engleski original The Merchant of Venice, sa WWW stranica projekta Gutenberg)

```
sage: import urllib
sage: venice = urllib.urlopen(
....:   'http://www.gutenberg.org/ebooks/2243.txt.utf-8').read()[16400:]
```

```
sage: len(venice)
120784
```

```
sage: print venice[:370]
The Merchant of Venice

Actus primus.

Enter Anthonio, Salarino, and Salanio.

  Anthonio. In sooth I know not why I am so sad,
It wearies me: you say it wearies you;
But how I caught it, found it, or came by it,
What stuffe 'tis made of, whereof it is borne,
I am to learne: and such a Want-wit sadnesse makes of
mee,
That I haue much ado to know my selfe
```

```
sage: for it in frequencies(venice)[:10]:
....:     print "%s:  %d" % it
 :  20927
e:  12250
o:  7260
t:  7227
a:  6233
h:  5560
n:  5518
s:  5317
r:  5232
i:  5206
```

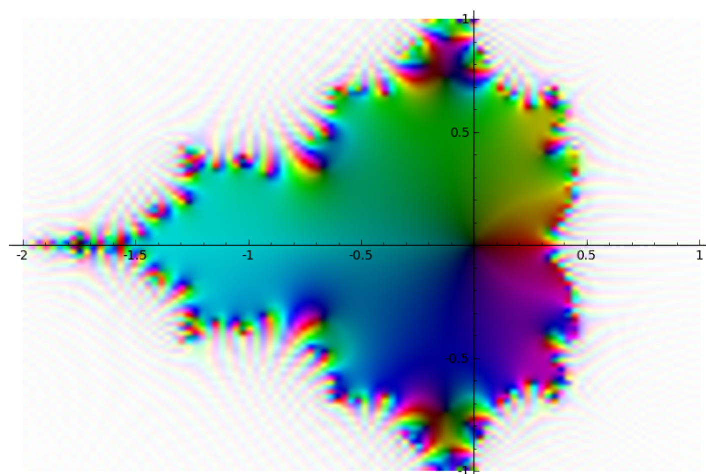
Vidimo da je "e" daleko najčešće slovo (poslije razmaka), činjenica vrlo važna pri dešifriranju engleskih tekstova.

Kao netrivijalniji primjer upotrebe gore definirane funkcije `compose()` možemo nacrtati tzv. Mandelbrotov skup, definiran kao skup svih točaka c kompleksne ravnine za koje iteracija

$$z_0 = 0 \; ; \qquad z_{n+1} = z_n^2 + c$$

ne divergira.

```
sage: complex_plot(compose(lambda z: z^2+x, 8, x), (-2, 1), (-1, 1))
```



(Analizirajte sami kako radi ovaj "program".)

Zadatak 1

Definirajte funkciju `allequal()` koja testira da li su svi elementi neke liste jednaki i iskoristite je da pronađete neprekidni niz od 6 istih znamenki u prvih 1000 decimala broja π. (Za pretvaranje broja u listu znamenaka možete iskoristiti funkciju `str()`.) Koristite ili obuhvaćanje liste ili `map()` tj. nije dopušteno korištenje petlji osim eventualno unutar funkcije `allequal()`, gdje to isto nije nužno (*Naputak*: `all()`).

Zadatak 2

Logističko preslikavanje je dano iteracijskom formulom $x_{n+1} = \lambda x_n(1 - x_n)$. Za male vrijednosti parametra λ, to preslikavanje za veliki n konvergira k jednoj vrijednosti. Kad λ raste, negdje blizu $\lambda = 3$, dolazi do bifurkacije i iteracije više ne konvergiraju već skaču između dvije vrijednosti. S daljnjim rastom λ dolazi do nove bifurkacije i sustav se za veliki n ponavlja s periodom četiri, itd. Rezultat se može prikazati kao tzv. Feigenbaumov bifurkacijski dijagram (vidi sliku). Nacrtajte ga tako da prvo definirate funkciju `composelist()` koja je analogna gornjoj funkciji `compose()`, ali ne vraća samo kranji rezultat kompozicije funkcija već i listu sa svim međukoracima. Nakon toga iskoristite tu funkciju za iteriranje logističkog preslikavanja. Eliminirajte prvi dio liste (dok se iteracije ne stabiliziraju) i nacrtajte drugi dio pomoću `list_plot(...., pointsize=1)`. Uočite da svaka vrijednost apscise `lambda` traži posebno iteriranje.

Zadatak 3

Odredite najmanji prirodni broj koji se *ne može* dobiti iz brojeva 2, 3, 4 i 5 korištenjem operacija zbrajanja, oduzimanja i množenja, a gdje se svaki broj smije koristiti najviše jednom. (Npr. 1=3-2, 2=3-5+4, ..., 6=2*3, ..., 14=4*5-3*2, ..., 30=(2+4)*5, ... *Naputak*: Od koristi se možda može pokazati već ranije korišteni modul `operator`, te funkcije `permutations()` i `combinations`.

Zadatak 4

Prim-brojevi blizanci su parovi prim-brojeva koji se razlikuju za dva, poput (3,5) ili (17,19). V. Brun je 1919. dokazao da suma recipročnih vrijednosti prim-brojeva blizanaca konvergira

$$B = \left(\frac{1}{3} + \frac{1}{5}\right) + \left(\frac{1}{5} + \frac{1}{7}\right) + \left(\frac{1}{11} + \frac{1}{13}\right) + \left(\frac{1}{17} + \frac{1}{19}\right) + \cdots = 1.902160583104$$

Ovako preciznu vrijednost je vrlo teško dobiti, no napišite funkciju `brun(n)` koja izračunava B koristeći listu od prvih n prim-brojeva. Pokušajte je isprogramirati i unutar paradigme funkcionalnog i unutar proceduralnog programiranja. Inače, zanimljivo je da je upravo računalno određivanje ove konstante ukazalo na bug u prvoj generaciji Pentium procesora.

Literatura

- Functional programming for Mathematicians
- Opsežniji tekst koji i kritizira upotrebu funkcionalnog programiranja u pythonu.
- Eseji Paula Grahama (oni koji se tiču programiranja)