Numerične metode v tehniki

Datum: 07/10/2024

Avtor: Aleksander Grm

Uvod v numerične metode

Kadar želimo simulirati izbrani fizikalni proces, ponavadi postopamo takole:

1. postavimo matematični model*,
2. izberemo numerično metodo in njene parametre,
3. pripravimo program (pomagamo si z vgrajenimi funkcijami),
4. izvedemo izračun, rezultate analiziramo in vrednotimo.

* Če lahko matematični model rešimo analitično, numerično reševanje ni potrebno.

Matematični model poskušamo rešiti analitično, saj taka rešitev ni obremenjena z napakami. Iz tega razloga se v okviru matematike učimo reševanja sistema enačb, integriranja, odvajanja in podobno. Bistvo **numeričnih metod** je, da matematične modele rešujemo **numerično**, torej na podlagi **diskretnih vrednosti**. Kakor bomo spoznali pozneje, nam numerični pristop v primerjavi z analitičnim omogoča reševanje bistveno obsežnejših in kompleksnejših problemov.

Napake pri numeričnem računanju

Konvertiranje binarnega zapisa v desetiški in obratno

Binarni zapis vsebuje vrednosti $[0,1]$, ki se nahajata na določenem bitnem položaju. Primer 8 bitnega zapisa $[10001001]$. Branje binarnega zapisa se prične na **desni strani**!

$$\mathrm{dec} = 2^{x_1} + 2^{x_2} + 2^{x_3} + 2^{x_4} + 2^{x_5} + 2^{x_6} + 2^{x_7} + 2^{x_8},$$

kjer je binarni vektor zapisan kot $[x_8\, x_7\, x_6\, x_5\, x_4\, x_3\, x_2\, x_1]$

Z uporabo Python okolja si lahko delo konverzije med formati števill zelo olajšamo. Poglejmo si primer:

In [ ]:
```
num = 12345
binary = format(num, 'b')

print('desetiško: {:d}'.format(num))
print('  binarno: {:}'.format(binary))
print()

# binary = '111'
binary = '1111111111' # 10 bit
num = int(binary, 2)
print('  binarno: {:}'.format(binary))
print('desetiško: {:d}'.format(num))
```

Zaokrožitvena napaka

V nadaljevanju si bomo pogledali nekatere omejitve in izzive numeričnega pristopa. Prva omejitev je, da so v računalniku realne vrednosti vedno zapisane s končno natančnostjo. V Pythonu se številla pogosto zapišejo v dvojni natančnosti s približno 15 signifikantnimi številkami.

Število z dvojno natančnostjo se v Pythonu imenuje **float64** (imenujemo ga tudi **double**) in je zapisano v spomin v binarni obliki v 64 bitih (11 bitov eksponent in 53 bitov mantisa (1 bit za predznak)). Ker je mantisa definirana na podlagi 52 binarnih številk, se lahko pojavi pri njegovem zapisu *relativna napaka* največ $\epsilon \approx 2.2 \cdot 10^{-16}$. Ta napaka se imenuje **osnovna zaokrožitvena napaka** in se lahko pojavi pri vsakem vmesnem izračunu!

Če je korakov veliko, lahko napaka zelo naraste in zato je pomembno, da je njen vpliv na rezultat čim manjši!

Spodaj je primer podrobnejših informacij za tip podatkov z dvojno natančnostjo (`float`); pri tem si pomagamo z vgrajenim modulom `sys` za klic parametrov in funkcij python sistema (dokumentacija):

In [ ]:
```
import sys
sys.float_info.epsilon
1e16+2
#sys.float_info #preverite tudi širši izpis!
```

Poleg števila z dvojno natančnostjo se uporabljajo drugi tipi podatkov; dober pregled različnih tipov je prikazan v okviru numpy in python dokumentacije.

Tukaj si poglejmo primer tipa `int8`, kar pomeni celo število zapisano z 8 biti (8 bit = 1 byte). Z njim lahko v dvojiškem sistemu zapišemo cela števila od -128 do +127:

In [ ]:
```
import numpy as np
stevilo = np.int8(1) # poskušite še števila: -128 in nato 127, 128, 129. Kaj se dogaja?
print('Število {:} tipa {:} zapisano v binari obliki: [{:08b}]'.format(stevilo,type(stevilo),stevilo))
```

Napaka metode

Poleg zaokrožitvene napake pa se pogosto srečamo tudi z **napako metode** ali **metodično napako**, ki jo naredimo takrat, ko natančen analitični postopek reševanja matematičnega modela zamenjamo s približnim numeričnim.

Pomembna lastnost numeričnih algoritmov je **stabilnost**. To pomeni, da majhna sprememba vhodnih podatkov povzroči majhno spremembo rezultatov. Če se ob majhni spremembi na vhodu rezultati zelo spremenijo, pravimo, da je **algoritem nestabilen**. V praksi torej uporabljamo stabilne algoritme; bomo pa pozneje spoznali, da je stabilnost lahko pogojena tudi z vhodnimi podatki!

Poznamo pa tudi nestabilnost matematičnega modela/naloge/enačbe; v tem primeru govorimo o **slabi pogojenosti**.

Med izvajanjem numeričnega izračuna se napake lahko širijo. Posledično je rezultat operacije manj natančen (ima manj zanesljivih števk), kakor pa je zanesljivost podatkov izračuna.

Poglejmo si sedaj splošen pristop k oceni napake. Točno vrednost označimo z r, približek z a_1; velja $r = a_1 + e_1$, kjer je e_1 napaka. Če z numeričnim algoritmom izračunamo bistveno boljši približek a_2, velja $r = a_2 + e_2$.

Ker velja $a_1 + e_1 = a_2 + e_2$, lahko ob predpostavki $|e_1| >> |e_2|$ in $|e_2| \approx 0$ izpeljemo $a_2 - a_1 = e_1 - e_2 \approx e_1$.

$|a_1 - a_2|$ je torej zesimistična ocena absolutne napake,

$\left|\frac{a_1 - a_2}{a_2}\right|$ pa ocena relativne napake.

Primer

Obravnavaj približen način izračuna za naslednjo algebraično enačbo

$$x^2 + \varepsilon x - 1 = 0,$$

kjer igra parameter ε vlogo igre drame v postopku reševanja. Njeno točno rešitev poznamo in je

$$x_{1/2} = -\frac{1}{2}\varepsilon \pm \sqrt{1 + \frac{1}{4}\varepsilon^2}$$

Tako lahko vsak približen način izračuna primerjamo z znano rešitvijo in ocenimo napako izračuna.

1. Reševanje z razvojem v vrsto

Za majhne vrednosti parametra $\varepsilon \ll 1$, lahko koren razvijemo v vrsto in dobimo razultat, ki je rešljiv z navadnim kalkulatorjem

$$\sqrt{1 + \frac{1}{4}\varepsilon^2} \approx 1 + \frac{1}{8}\varepsilon^2 - \frac{1}{128}\varepsilon^4 + \mathcal{O}(\varepsilon^5)$$

tako je potem rešitev poenostavljena

$$x_{1/2} \approx \begin{cases} 1 - \frac{1}{2}\varepsilon + \frac{1}{4}\varepsilon^2 - \frac{1}{128}\varepsilon^4 + \mathcal{O}(\varepsilon^5) \\ -1 - \frac{1}{2}\varepsilon - \frac{1}{8}\varepsilon^2 + \frac{1}{128}\varepsilon^4 + \mathcal{O}(\varepsilon^5) \end{cases}$$

In [ ]:
```
import math as mat
import numpy as np
import matplotlib.pyplot as mpl

# MatPlotLib set fonts
mpl.rcParams['font.family'] = 'serif'
mpl.rcParams['font.serif'] = ['DejaVu Serif']

# MatPlotLib set LaTeX use
mpl.rcParams['text.usetex'] = True
mpl.rcParams['text.latex.preamble'] = r'\usepackage{siunitx}'
```

In [ ]:
```
def func_series(e):
    x1 = 1 - e/2 + e**2/8 - e**4/128
    x2 = -1 - e/2 - e**2/8 + e**4/128

    return [x1,x2]
```

In [ ]:
```
eps = np.linspace(0.1, 2, 30)

result = []
for ei in eps:
    sq = mat.sqrt(1 + ei**2/4)
    xe1 = -0.5*ei + sq
    xe2 = -0.5*ei - sq
    [x1, x2] = func_series(ei)
    result.append([[x1,xe1],[x2,xe2]])

result = np.array(result)
```

In [ ]:
```
x1 = result[:,0] # prvi del rešitve
x2 = result[:,1] # drugi del rešitve
```

In [ ]:
```
fig, ax = mpl.subplots(3)
fig.suptitle(r'Napaka pri izračunu - razvoj v vrsto') # Figure title

ax[0].plot(eps,x1[:,0],'g', label='točna')
ax[0].plot(eps,x1[:,1],'r', label='vrsta')
ax[1].plot(eps,np.abs(x1[:,0] - x1[:,1]))
ax[2].plot(eps,np.abs((x1[:,0] - x1[:,1])/x1[:,0]))

ax[0].set_xticklabels([])
ax[0].set_ylabel(r'rešitev')
ax[0].legend()
ax[0].grid()

ax[1].set_xticklabels([])
ax[1].set_ylabel(r'abs. napaka')
ax[1].grid()

ax[2].set_xlabel(r'$\varepsilon$')
ax[2].set_ylabel(r'rel. napaka')
ax[2].grid()

fig.tight_layout()
fig.savefig('sol_x1.pdf')
```

In [ ]:
```
fig, ax = mpl.subplots(3)
fig.suptitle(r'Napaka pri izračunu - razvoj v vrsto') # Figure title

ax[0].plot(eps,x2[:,0],'g', label='točna')
ax[0].plot(eps,x2[:,1],'r', label='vrsta')
ax[1].plot(eps,np.abs(x2[:,0] - x2[:,1]))
ax[2].plot(eps,np.abs((x2[:,0] - x2[:,1])/x2[:,0]))

ax[0].set_xticklabels([])
ax[0].set_ylabel(r'rešitev')
ax[0].legend()
ax[0].grid()

ax[1].set_xticklabels([])
ax[1].set_ylabel(r'abs. napaka')
ax[1].grid()

ax[2].set_xlabel(r'$\varepsilon$')
ax[2].set_ylabel(r'rel. napaka')
ax[2].grid()

fig.tight_layout()
fig.savefig('sol_x2.pdf')
```

2. Računanje z iterativno metodo

Začetno enačbo $x^2 + \varepsilon x - 1 = 0$ nekoliko predrugačimo, da bo uporabna za izračun s pomočjo iterativne metode

$$x = \pm\sqrt{1 - \varepsilon x}.$$

Vsaka rešitev te enačbe, je tako rešitev začetne enačbe in obratno!

Če sedaj uporabimo iterativni postopek, kjer seveda potrebujemo začetni približek rešitve, imamo tako rekurzivno enačbo, ki jo rešujemo

$$x_{n+1} = \pm\sqrt{1 - \varepsilon x_n}.$$

Vzemimo samo pozitivno vejo in enako kakor prej, korenov ne znamo izračunat in si pomagamo z izračunom, ki ni natančen ampak približen. Enako je tudi tukaj potrebno oceniti napako.

Zgornjo enačbo razvijemo v vrsto

$$x_{n+1} = \sqrt{1 - \varepsilon x_n} \approx 1 - \frac{1}{2}\varepsilon x_n.$$

Pri nekem r pričnemo z reševanjem, kjer lahko pričnemo z iterativnim postopkom za $x_0 = 1$ ($\varepsilon = 0$).

In [ ]:
```
x_old = 1
err = 1e-8
e_par = np.linspace(0.1,1.5,100)

data = []
for ei in e_par:
    xe = -0.5*ei + mat.sqrt(1 + ei**2/4)
    sol = []
    idx = 0
    x_old = 1
    x = 1 - 0.5*ei*x_old

    while np.abs(x_old - x) > err:
        x_old = x
        x = 1 - 0.5*ei*x_old
        sol.append([idx, x])
        idx += 1

    data.append([ei, xe, np.array(sol)])
```

In [ ]:
```
fig, ax = mpl.subplots()
fig.suptitle(r'Iterativna metoda ($\varepsilon = {:.2f}$)'.format(ei)) # Figure title

ax.plot(data[0][2][:,0],data[0][2][:,1])

ax.set_xlabel('$n$ - iterativni korak')
ax.set_ylabel(r'$x_{n} - rešitev')
ax.grid()
ax.autoscale_view()

fig.tight_layout()
fig.savefig('sol_iteracija.pdf')
```

In [ ]:
```
error = []
for d in data:
    error.append([d[0], d[2][-1,0], np.abs(d[1] - d[2][-1,1])])
error = np.array(error)
```

In [ ]:
```
fig, ax = mpl.subplots(2)
fig.suptitle(r'Iterativna metoda - napaka'.format(ei)) # Figure title

ax[0].plot(error[:,0],error[:,2])
ax[1].plot(error[:,0],error[:,1])

ax[0].set_xticklabels([])
ax[0].set_ylabel(r'napaka')
ax[0].grid()

ax[1].set_xlabel(r'$\varepsilon$')
ax[1].set_ylabel(r'$n$ korakov')
ax[1].grid()

fig.tight_layout()
fig.savefig('err_iteracija.pdf')
```

In [ ]: