Numerično odvajanje

Datum: 10/11/2024

Avtor: Aleksander Grm

V zapiskih so uporabljeni primeri iz OnLine knjige Numerične metode v ekosistemu Pythona, Janko Slavič

Najprej naložimo celoten potreben Python ekosistem

In […
```python
import numpy as np              # orodja za numeriko
import matplotlib.pyplot as mpl # izdelava grafov

from IPython.display import YouTubeVideo
```

Uvod

Vsako elementarno funkcijo lahko analitično odvajamo. Definicija odvoda je:

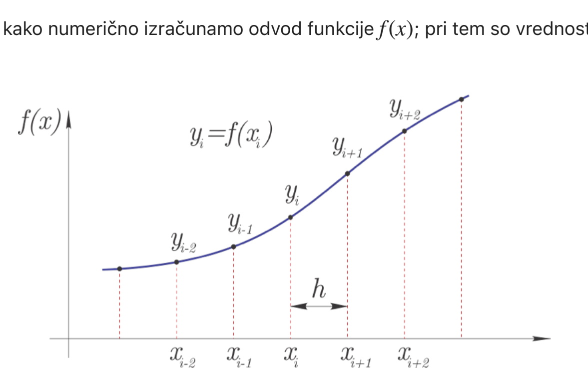
$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}.$$

Neposredna uporaba zgornje enačbe vodi v odštevanje zelo podobnih funkcijskih vrednostih ($f(x+\Delta x), f(x)$), obremenjenih z zaokrožitveno napako, ki jih delimo z majhno vrednostjo Δx; posledično ima odvod bistveno manj signifikantnih števk kakor pa funkcijske vrednosti. Numeričnemu odvajanju se izognemo, če imamo to možnost; je pa v nekaterih primerih (npr. reševanje diferencialnih enačb) nepogrešljivo orodje!

Pri numeričnem odvajanju imamo dva, v principu različna, pristopa:

1. najprej izvedemo **interpolacijo/aproksimacijo**, nato pa na podlagi znanih interpolacijskih/aproksimacijskih funkcij izračunamo odvod (o tej temi smo že govorili pri interpolaciji oz. aproksimaciji) in
2. računanje odvoda **neposredno iz vrednosti iz tabele**.

V okviru tega poglavja se bomo seznanili s tem, kako numerično izračunamo odvod funkcije $f(x)$; pri tem so vrednosti funkcije $f(x)$ podane tabelarično (pari x_i, y_i), kakor je prikazano na sliki



Najprej se bomo osredotočili na ekvidistantno, s korakom h, razporejene vrednosti x_i; vrednosti funkcije pa bodo $y_i = f(x_i)$.

Glede na zgornjo definicijo odvoda, bi prvi odvod (za mesto i) lahko zapisali:

$$y_i' = \frac{y_{i+1} - y_i}{h},$$

kjer je $h = x_{i+1} - x_i$. S preoblikovanjem enačbe:

$$y_i' = -\frac{y_i}{h} + \frac{y_{i+1}}{h},$$

lahko tudi rečemo, da za prvi odvod funkcije na mestu i, **utežimo** funkcijsko vrednost pri i z $-1/h$ in funkcijsko vrednost pri $i+1$ z $+1/h$.

Metoda končnih diferenc

Kot uvod v aproksimacijo odvoda s pomočjo končnih diferenc si oglejte spodnji video!

In […
```python
from IPython.display import YouTubeVideo
YouTubeVideo('YYuGL-VP2BE', width=800, height=300)
```

Odvod $f'(x)$ lahko aproksimiramo na podlagi razvoja Taylorjeve vrste. To metodo imenujemo **metoda končnih razlik** ali tudi **diferenčna metoda**.

Razvijmo **Taylorjevo vrsto naprej** (naprej, zaradi člena $+h$):

$$f(x+h) = \sum_{n=0}^{\infty} \frac{h^n}{n!} \frac{d^n}{dx^n} f(x) = f(x) + h f'(x) + \underbrace{\frac{h^2}{2} f''(x) + \cdots}_{\mathcal{O}(h^2)}$$

Člen $\mathcal{O}\left(h^2\right)$ označuje napako drugega reda. Če iz enačbe izrazimo prvi odvod:

$$f'(x) = \frac{1}{h}(f(x+h) - f(x)) - \underbrace{\frac{h}{2} f''(x) + \cdots}_{\mathcal{O}(h^1)}$$

Ugotovimo, da lahko ocenimo prvi odvod v točki x_i (to je: $f_o'(x_i)$) na podlagi dveh zaporednih funkcijskih vrednosti:

$$f_o'(x_i) = \frac{1}{h}(y_{i+1} - y_i)$$

in pri tem naredimo **napako metode**, ki je prvega reda $\mathcal{O}\left(h^1\right)$.

Uporabili smo $y_i = f(x_i)$ (glejte sliko zgoraj).

Napaka je:

$$e = -\frac{h}{2} f''(\xi),$$

kjer je ξ neznana vrednost na intervalu $[x_i, x_{i+1}]$ in smo zanemarili višje člene.

Velja torej izraz:

$$f'(x_i) = f_o'(x_i) + e.$$

Uporaba `SymPy` paketa za simbolično odvajanje

Paket `sympy` nam omogoča uporabo algebrajičnega ali simboličnega sistema za izpeljevanje komliciranih matematičnih izrazov. Kaj več o samem paketu SymPy si lahko pogledate v dokumentaciji SymPy dokumentacija

In […
```python
import sympy as sym # naložimo sympy paket
sym.init_printing() # postavimo izpis rezultata v pretty način
```

In […
```python
# Za algebrajične manipulacije je najprej potrebno definirati finkcije in simbole, ki služijo v izpeljavi

f = sym.Function('f')
x, h = sym.symbols('x, h')
```

In […
```python
display(f)
```

In […
```python
# Nato nadaljujemo z razvojem **Taylorjeve vrste naprej** (angl. *forward Taylor series*)

f(x+h).series(h, n=2)
```

Člen $\mathcal{O}\left(h^2\right)$ vsebuje člene drugega in višjega reda. V zgornji enačbi je uporabljena začasna spremenljivko za odvajanje ξ_1; izvedmo odvajanje in vstavimo $\xi_1 = x$:

In […
```python
f(x+h).series(h, n=3).doit()
```

In […
```python
# Zapišemo enačbo, ki jo bomo manipulirali, ki vsebuje 1. odvod

eqn_01 = sym.Eq(f(x+h), f(x+h).series(h, n=2).doit())
eqn_01
```

In […
```python
# Izrazimo 1. odvod kot spremenljivko
diff_f = f(x).diff(x)
display(diff_f)

# Sedaj poiščimo kako se izraža 1. odvod, s pomočjo rešitve enačbe
f1_fwd_exact = sym.solve(eqn_01, diff_f)[0]
display(f1_fwd_exact)
```

In […
```python
f1_fwd_exact.expand()
```

In […
```python
# V kolikor drugega in višjih odvodov ne upoštevamo, storimo napako:

f1_fwd_O = f1_fwd_exact.expand().getO()
f1_fwd_O
```

Napaka $\mathcal{O}(h) = \mathcal{O}\left(h^1\right)$ je torej prvega reda in če ta člen zanemarimo, naredimo *napako metode* in dobimo oceno odvoda:

In […
```python
f1_fwd_est = f1_fwd_exact.expand().removeO()
f1_fwd_est
```

Ugotovimo, da gre za isti izraz, kakor smo ga izpeljali zgoraj, torej je:

$$y_i' = \frac{1}{h}(-y_i + y_{i+1}).$$

Uteži torej so:

Odvod↓ \ **Vrednosti** →	y_i	y_{i+1}
$y_i' = \frac{1}{h}\cdot$	-1	1

Centralna diferenčna shema

1. odvod

Najprej si poglejmo razvoj **Taylorjeve vrste nazaj** (angl. *backward Taylor series*):

In […
```python
sym.Eq(f(x-h), f(x-h).series(h, n=3).doit())
```

razvoj **Taylorjeve vrste naprej** (angl. forwward Taylor series):

In […
```python
sym.Eq(f(x+h), f(x+h).series(h, n=3).doit())
```

Ugotovimo, da se pri razliki vrste naprej in nazaj odštevajo členi sodega reda; definirajmo:

In […
```python
def difference(n=3):
    return f(x+h).series(h, n=n).doit()-f(x-h).series(h, n=n).doit()

difference(n=3)
```

Izvedemo sledeče korake:

1. Taylorjevo vrsto nazaj odštejemo od vrste naprej, sodi odvodi se odštejejo,
2. rešimo enačbo za prvi odvod,
3. določimo napako metode,
4. določimo oceno odvoda.

Izvedimo zgornje korake:

In […
```python
f1_cent_exact = sym.solve(
          sym.Eq(f(x+h) - f(x-h), difference(n=3)), # 1 korak
          f(x).diff(x))[0]                          # 2.korak
f1_cent_O = f1_cent_exact.expand().getO()           # 3.korak
f1_cent_est = f1_cent_exact.expand().removeO()      # 4.korak
```

In […
```python
# 1. odvod za centralno diferenčno shemo

f1_cent_est
```

Ali zapisano drugače

$$y_i' = \frac{1}{2h}\left(-y_{i-1} + y_{i+1}\right)$$

Uteži torej so:

Odvod↓ \ **Vrednosti** →	y_{i-1}	y_i	y_{i+1}
$y_i' = \frac{1}{2h}\cdot$	-1	0	1

In […
```python
# Napaka metode je enaka drugemu redu, akr je bistveno bolje od prej

f1_cent_O
```

2. odvod

Če Taylorjevo vrsto naprej in nazaj seštejemo, se odštejejo lihi odvodi:

In […
```python
def sum_prts(n=3):
    return f(x+h).series(h, n=n).doit() + f(x-h).series(h, n=n).doit()

sum_prts(n=4)
```

In […
```python
# Določimo 2. odvod

f2_cent_exact = sym.solve(
          sym.Eq(f(x+h) + f(x-h), sum_prts(n=4)),   # 1 korak
          f(x).diff(x,2))[0]                       # 2.korak
f2_cent_O = f2_cent_exact.expand().getO()          # 3.korak
f2_cent_est = f2_cent_exact.expand().removeO()     # 4.korak
```

In dobimo za oceno 2. odvoda

In […
```python
f2_cent_est
```

Ali zapisano drugače

$$y_i'' = \frac{1}{h^2}\left(y_{i-1} - 2\,y_i + y_{i+1}\right)$$

Uteži torej so:

Odvod↓ \ **Vrednosti** →	y_{i-1}	y_i	y_{i+1}
$y_i'' = \frac{1}{h^2}\cdot$	1	-2	1

Napaka metode je v tem primeru enaka

In […
```python
f2_cent_O
```

Necentralne diferenčne sheme

Centralna diferenčna shema, ki smo jo spoznali zgoraj, je zelo uporabna in relativno natančna. Ker pa je ne moremo vedno uporabiti (recimo na začetku ali koncu tabele), si moramo pomagati z **necentralnimi diferenčnimi shemami** za računanje odvodov.

Poznamo:

- **diferenčno shemo naprej**, ki odvod točke aproksimira z vrednostmi funkcije v naslednjih točkah in
- **diferenčno shemo nazaj**, ki odvod točke aproksimira z vrednostmi v predhodnih točkah.

Izpeljave so podobne, kakor smo prikazali za centralno diferenčno shemo, zato jih tukaj ne bomo obravnavali in bomo prikazali samo končni rezultat.

Diferenčna shema - Naprej (Forward)

Diferenčna shema naprej z redom napake $\mathcal{O}(h^1)$:

Odvod↓ \ **Vrednosti** →	y_i	y_{i+1}	y_{i+2}	y_{i+3}	y_{i+4}
$y_i' = \frac{1}{h}\cdot$	-1	1	0	0	0
$y_i'' = \frac{1}{h^2}\cdot$	1	-2	1	0	0
$y_i''' = \frac{1}{h^3}\cdot$	-1	3	-3	1	0
$y_i^{(4)} = \frac{1}{h^4}\cdot$	1	-4	6	-4	1

Diferenčna shema naprej z redom napake $\mathcal{O}(h^2)$:

Odvod↓ \ **Vrednosti** →	y_i	y_{i+1}	y_{i+2}	y_{i+3}	y_{i+4}	y_{i+5}
$y_i' = \frac{1}{2h}\cdot$	-3	4	-1	0	0	0
$y_i'' = \frac{1}{h^2}\cdot$	2	-5	4	-1	0	0
$y_i''' = \frac{1}{2h^3}\cdot$	-5	18	-24	14	-3	0
$y_i^{(4)} = \frac{1}{h^4}\cdot$	3	-14	26	-24	11	-2

Diferenčna shema - Nazaj (Backward)

Diferenčna shema nazaj z redom napake $\mathcal{O}(h^1)$:

Odvod↓ \ **Vrednosti** →	y_{i-4}	y_{i-3}	y_{i-2}	y_{i-1}	y_i
$y_i' = \frac{1}{h}\cdot$	0	0	0	-1	1
$y_i'' = \frac{1}{h^2}\cdot$	0	0	1	-2	1
$y_i''' = \frac{1}{h^3}\cdot$	0	-1	3	-3	1
$y_i^{(4)} = \frac{1}{h^4}\cdot$	1	-4	6	-4	1

Diferenčna shema nazaj z redom napake $\mathcal{O}(h^2)$:

Odvod↓ \ **Vrednosti** →	y_{i-5}	y_{i-4}	y_{i-3}	y_{i-2}	y_{i-1}	y_i
$y_i' = \frac{1}{2h}\cdot$	0	0	0	1	-4	3
$y_i'' = \frac{1}{h^2}\cdot$	0	0	-1	4	-5	2
$y_i''' = \frac{1}{2h^3}\cdot$	0	3	-14	24	-18	5
$y_i^{(4)} = \frac{1}{h^4}\cdot$	-2	11	-24	26	-14	3

Uporaba `numpy` knjižnice za odvajanje - `numpy.gradient`

Za izračun numeričnih odvodov (centralna diferenčna shema 2. reda) lahko uporabimo tudi `numpy.gradient()` (dokumentacija):

`gradient(f, *varargs, **kwargs)`
kjer `f` predstavlja tabelo vrednosti (v obliki numeričnega polja) funkcije, katere odvod iščemo. `f` je lahko ene ali več dimenzij. Pozicijski parametri `varargs` definirajo razdaljo med vrednostmi argumenta funkcije `f`; privzeta vrednost je 1. Ta vrednost je lahko skalar, lahko pa tudi seznam vrednosti neodvisne spremenljivke (ali tudi kombinacija obojega). Gradientna metoda na robovih uporabi shemo naprej oziroma nazaj; parameter `edge_order` definira red sheme, ki se uporabi na robovih (izbiramo lahko med 1 ali 2, privzeta vrednost je 1).

Rezultat funkcije `gradient` je numerični seznam (ali seznam numeričnih seznamov) z izračunanimi odvodi.

Za podrobnosti glejte dokumentacijo.

Zgled

Pogledali si bomo zgled, kako uporabimo **uteži**, funkcijo gradient in posebnosti na robovih. Najprej pripravimo tabelo podatkov:

In […
```python
x, h = np.linspace(0, 1, 20, retstep=True)
y = np.sin(2*np.pi*x)
```

Uteži difernečnih shem:

In […
```python
centralna = np.array([-0.5, 0, 0.5]) # bi lahko tudi pridobili prek central_diff_weights(3,1)
naprej = np.array([-3/2, 2, -1/2])
nazaj = np.array([1/2, -2, 3/2])
```

Sedaj izvedemo odvod notranjih točk (prvi način je z izpeljevanjem seznamov, drugi je vektoriziran):

In […
```python
odvod_notranje = np.array([y[i-1:i+2] @ centralna/h for i in range(1, len(x)-1)]) # izpeljevanje seznamov
odvod_notranje = np.convolve(y, centralna[::-1], mode='valid') / h # vektoriziran
```

Na robovih uporabimo diferenčno shemo naprej oziroma nazaj:

In […
```python
odvod_prva = y[:len(naprej)] @ naprej / h  # naprej
odvod_zadnja = y[-len(nazaj):] @ nazaj / h # nazaj
```

Sestavimo rezultat:

In […
```python
odvod_cel = np.hstack([odvod_prva, odvod_notranje, odvod_zadnja])
```

Prikažemo rezultat skupaj z rezultatom funkcije `np.gradient`:

In […
```python
mpl.plot(x, odvod_cel, 'ko', lw=3, label='lastna implementacija')
mpl.plot(x, np.gradient(y, h), 'g.', label='np.gradient, edge_order=1')
mpl.plot(x, np.gradient(y, h, edge_order=2), 'r.', label='np.gradient, edge_order=2')
mpl.legend();
```