Sistemi Linearnih Enačb

Datum: 04/11/2024

Avtor: Aleksander Grm

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

Najprej naložimo celoten potreben Python ekosistem

```python
import numpy as np              # orodja za numeriko
import matplotlib.pyplot as mpl # izdelava grafov
import numpy.polynomial as poly # paket za podporo polinomov
import scipy.optimize as opt    # uporaba fsolve() funkcije

from IPython.display import YouTubeVideo
```

Uvod v sisteme linearnih enačb

Pod zgornjim naslovom razumemo sistem m linearnih enačb $(E_i, i=0,1,\dots,m-1)$ z n neznankami $(x_j, j=0,1,\dots,n-1)$:

$$\begin{array}{rlllllll}
E_0: & A_{0,0}\,x_0 & + & A_{0,1}\,x_1 & + \dots + & A_{0,n-1}\,x_{n-1} & = & b_0\\
E_1: & A_{1,0}\,x_0 & + & A_{1,1}\,x_1 & + \dots + & A_{1,n-1}\,x_{n-1} & = & b_1\\
\vdots & & & & \vdots & & & \\
E_{m-1}: & A_{m-1,0}\,x_0 & + & A_{m-1,1}\,x_1 & + \dots + & A_{m-1,n-1}\,x_{n-1} & = & b_{m-1}.
\end{array}$$

Koeficienti $A_{i,j}$ in b_i so znana, ponavadi realna števila. V posebnih primerih so lahko tudi kompleksna števila.

V kolikor je desna stran enaka nič, torej $b_i=0$, imenujemo sistem **homogenem**, sicer je sistem **nehomogen**.

Sistem enačb lahko zapišemo tudi v matrični obliki:

$$\mathbf{A}\,\mathbf{x}=\mathbf{b},$$

kjer sta $\mathbf{A}$ in $\mathbf{b}$ znana matrika in vektor, vektor $\mathbf{x}$ vsebuje neznanke in tako ni znan. Matriko $\mathbf{A}$ imenujemo **matrika koeficientov**, vektor $\mathbf{b}$ **vektor konstant** (tudi: vektor prostih členov ali vektor stolpec desnih strani) in $\mathbf{x}$ **vektor neznank**. Če matriki $\mathbf{A}$ dodamo kot stolpec vektor $\mathbf{b}$, dobimo t. i. **razširjeno matriko** in jo označimo $[\mathbf{A}|\mathbf{b}]$.

Opomba glede zapisa:

- skalarne spremenljivke pišemo poševno, npr.: a, A,
- vektorske spremenljivke pišemo z majhno črko poudarjeno, npr.: $\mathbf{a}$,
- matrične spremenljivke pišemo z veliko črko poudarjeno, npr.: $\mathbf{A}$.

Postopek reševanja sistema linearnih enačb

Spodaj si lahko ogledate kratko video predstavitev

```python
from IPython.display import YouTubeVideo
YouTubeVideo('7YbyijGUbYw', width=800, height=300)
```

Če nad sistemom linearnih enačb izvajamo **elementarne operacije**:

- množenje poljubne enačbe s konstanto (ki je različna od nič),
- spreminjanje vrstnega reda enačb,
- prištevanje ene enačbe (pomnožene s konstanto) drugi enačbi.

Z opisanimi opreacijami **rešitve sistema ne spremenimo** in dobimo ekvivalentni sistem enačb.

S pomočjo elementarnih operacij nad vrsticami matrike $\mathbf{A}$ jo lahko preoblikujemo v t. i. **vrstično kanonično obliko**:

1. če obstajajo ničelne vrstice, so te na dnu matrike,
2. prvi neničelni element se nahaja desno od prvih neničelnih elementov predhodnih vrstic,
3. prvi neničelni element v *vrstici* imenujemo **pivot** in je enak 1,
4. pivot je edini neničelni element v *stolpcu*.

Rang matrike predstavlja število neničelnih vrstic v vrstični kanonični obliki matrike; število neničelnih vrstic predstavlja **število linearno neodvisnih enačb in je enako številu pivotnih elementov**. **Rang matrike je torej enak številu linearno neodvisnih vrstic matrike**. Transponiranje matrike njenega ranga ne spremeni, zato je rang matrike enak tudi številu linearno neodvisnih *stolpcev* matrike.

Primer preoblikovanja matrike $\mathbf{A}$:

```python
A_org = np.arange(9).reshape((3,3))+1
A = A_org
print(A)
```

Element `A[0,0]` je neničelen in ima vrednost 1 tako je **pivotni element**.

Prvo vrstico `A[0,:]` pomnožimo z -4 in produkt prištejemo drugi vrstici `A[1,:]-4A[0,:]`:

```python
A[1,:] -= A[1,0]*A[0,:]
print(A)
```

V enakem stilu naredimo tudi za tretjo vrstico.

```python
A[2,:] -= A[2,0]*A[0,:]
print(A)
```

Drugo vrstico sedaj delimo z `A[1,1]`, da dobimo pivotni element v vrstici 1:

```python
A[1,:] = A[1,:]/A[1,1]
print(A)
```

Odštejemo drugo vrstico od ostalih, da dobimo v drugem stolpcu ničle povsod, razen v drugi vrstici vrednost 1:

```python
A[0,:] -= A[0,1]*A[1,:] # odštevanje od prve vrstice
A[2,:] -= A[2,1]*A[1,:] # odštevanje od zadnje vrstice
print(A)
```

Tako nam po algebrajični manipulaciji ostaneta samo dve neničelni vrstici sledi, da ima matrika `A` dva pivota in predstavlja dve linearno neodvisni enačbi. Rang matrike je 2.

Rang matrike lahko določimo tudi s pomočjo `numpy` funkcije `numpy.linalg.matrix_rank` (dokumentacija):

`matrix_rank(M, tol=None)`
kjer je `M` matrika, katere rang iščemo, `tol` opcijski parameter, ki določa mejo, pod katero se vrednosti v algoritmu smatrajo enake nič.

```python
# Test naše originalne matrike in manipulirane matrike

rk_A_org = np.linalg.matrix_rank(A_org)
rk_A = np.linalg.matrix_rank(A)

print('rk(A_org):', rk_A_org)
print('    rk(A):', rk_A)
```

Če velja $r=\mathbf{rang}(\mathbf{A})=\mathbf{rang}([\mathbf{A}|\mathbf{b}])$, potem rešitev **obstaja** (rečemo tudi, da je sistem **konsistenten**).

Konsistenten sistem ima:

- natanko eno rešitev, ko je število neznank n enako rangu r (rešitev je neodvisna) in
- neskončno mnogo rešitev, ko je rang r manjši od števila neznank n (rešitev je odvisna od $n-r$ parametrov).

Najprej se bomo omejili na sistem $m=n$ linearnih enačb z n neznankami ter velja $n=r$:

$$\mathbf{A}\,\mathbf{x}=\mathbf{b}.$$

Pod zgornjimi pogoji je matrika koeficientov $\mathbf{A}$ nesingularna ($|\mathbf{A}|\neq 0$) in sistem ima rešitev:

$$\mathbf{x}=\mathbf{A}^{-1}\,\mathbf{b}.$$

Poglejmo si primer sistema, ko so **enačbe linearno odvisne** ($r<n$):

```python
A = np.array([[1 , 2],
              [2, 4]])
b = np.array([1, 2])
Ab = np.column_stack((A,b))

print('razširjena matrika:\n\n', Ab)
```

S pomočjo `numpy` knjižnice poglejmo sedaj rang matrike koeficientov in razširjene matrike ter determinanto z uporabo `numpy.linalg.det` (dokumentacija):

`det(A)`
kjer je `A` matrika (ali seznam matrik), katere determinanto iščemo; funkcija `det` vrne determinanto (ali seznam determinant).

```python
f'rang(A)={np.linalg.matrix_rank(A)}, rang(Ab)={np.linalg.matrix_rank(Ab)}, \
število neznank: {len(A[:,0])}, det(A)={np.linalg.det(A)}'
```

Poglejmo še primer, ko **rešitve sploh ni** (nekonsistenten sistem):

```python
A = np.array([[1 , 2],
              [2, 4]])
b = np.array([1, 1])
Ab = np.column_stack((A,b))

print('razširjena matrika:\n\n', Ab)
```

```python
f'rang(A)={np.linalg.matrix_rank(A)}, rang(Ab)={np.linalg.matrix_rank(Ab)}, \
število neznank: {len(A[:,0])}, det(A)={np.linalg.det(A)}'
```

Norma in pogojenost sistema linearnih enačb

Numerična naloga je slabo pogojena, če majhna sprememba podatkov povzroči veliko spremembo rezultata. V primeru *majhne spremembe podatkov*, ki povzročijo *majhno spremembo rezultatov*, pa je naloga **dobro pogojena**.

Sistem enačb je ponavadi dobro pogojen, če so absolutne vrednosti diagonalnih elementov matrike koeficientov velike v primerjavi z absolutnimi vrednostmi izven diagonalnih elementov.

Za sistem linearnih enačb $\mathbf{A}\,\mathbf{x}=\mathbf{b}$ lahko računamo **število pogojenosti** (*angl.* condition number):

$$\mathrm{cond}(\mathbf{A})=||\mathbf{A}||\,||\mathbf{A}^{-1}||.$$

Z $||\mathbf{A}||$ je označena **norma** matrike.

Obstaja več načinov računanja norme; navedimo dve:

- Evklidska norma (tudi Frobeniusova):

$$||\mathbf{A}||_e=\sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n}A_{ij}^2}$$

- Norma vsote vrstic ali tudi neskončna norma:

$$||\mathbf{A}||_\infty=\max_{1\le i\le n}\sum_{j=1}^{n}|A_{ij}|$$

Pogojenost računamo z vgrajeno funkcijo `numpy.linalg.cond` (dokumentacija):

`cond(x, p=None)`
ki sprejme dva parametra: matriko `x` in opcijski tip norme `p` (privzeti tip je `None`; v tem primeru se uporabi Evklidska/Frobeniusova norma).

Če je število pogojenosti majhno, potem je matrika dobro pogojena in obratno - pri slabi pogojenosti se število pogojenosti zelo poveča.

Žal je izračun pogojenosti matrike numerično relativno zahteven.

Primer slabo pogojene matrike

Pogledali si bomo slabo pogojen sistem, kjer bomo z malenkostno spremembo na matriki koeficientov povzročili veliko spremembo rešitve.

Matrika koeficientov:

```python
A = np.array([[1 , 1],
              [1, 1.00001]])
cond_A = np.linalg.cond(A)

print('pogojenost:', cond_A)
```

Dodamo še desno stran sistema

```python
b = np.array([3, -3])
Ab = np.column_stack((A,b))

print(Ab)
```

Preverimo osnovne lastnosti SLE

```python
msg = 'rk(A): {:}, rk(A|b): {:}, size(x): {} (št. neznank), det(A): {:}'.format(np.linalg.matrix_rank(A), np.linalg.matrix_rank(Ab),len
print(msg)
```

Rešimo zgornji SLE, za algebrajično manipulacijo:

```python
# od druge enačbe odštejemo prvo
Ab[1,:] -= Ab[0,:]
Ab
```

```python
# določimo vrednost x_1
x1 = Ab[1,2]/Ab[1,1]
x1
```

```python
# določimo vrednost x_0
x0 = (Ab[0,2] - Ab[0,1]*x1)/Ab[0,0]
x0
```

Sedaj zgornji SLE **rahlo spremenimo**

```python
A = np.array([[1 , 1],
              [1, 1.0001]]) # prejšnje stanje [1, 1.00001]
cond_A = np.linalg.cond(A)

print('pogojenost:', cond_A)
```

```python
b = np.array([3, -3])
Ab = np.column_stack((A,b))

print(Ab)
```

Preverimo osnovne lastnosti SLE

```python
msg = 'rk(A): {:}, rk(A|b): {:}, size(x): {} (št. neznank), det(A): {:}'.format(np.linalg.matrix_rank(A), np.linalg.matrix_rank(Ab),len
print(msg)
```

Rešimo zgornji SLE, za algebrajično manipulacijo:

```python
# od druge enačbe odštejemo prvo
Ab[1,:] -= Ab[0,:]
Ab
```

```python
# določimo vrednost x_1
x1 = Ab[1,2]/Ab[1,1]
x1
```

```python
# določimo vrednost x_0
x0 = (Ab[0,2] - Ab[0,1]*x1)/Ab[0,0]
x0
```

Ugotovimo, da je malenkostna sprememba enega koeficienta v matriki koeficientov povzročila veliko spremembo v rezultatu. Majhni spremembi podatkov se ne moremo izogniti, zaradi zapisa podatkov v računalniku.

Numerično reševanje SLE

Obstajata dva, v principu različna pristopa k reševanju sistemov linearnih enačb:

A) **Direktni pristop**: nad sistemom enačb izvajamo elementarne operacije, s katerimi predelamo sistem enačb v lažje rešljivega,

B) **Iterativni pristop**: izberemo začetni približek, nato pa približek iterativno izboljšujemo.

Mi si bomo pogledali samo sistem **A**, za sistem **B** bo uporabljena interna `Python` metoda.

Gaussova eliminacijska metoda - Direktna metoda

Predpostavimo, da rešujemo sistem n enačb za n neznank, ki ima rang n. Tak sistem je enolično rešljiv.

Gaussova eliminacija spada med direktne metode, saj s pomočjo elementarnih vrstičnih operacij sistem enačb prevedemo v zgornje poravnani trikotni sistem (pod glavno diagonalo v razširjeni matriki so vrednosti nič).

Najprej pripravimo razširjeno matriko koeficientov:

$$[\mathbf{A}|\mathbf{b}]=\left[\begin{array}{cccc|c}
A_{0,0} & A_{0,1} & \cdots & A_{0,n-1} & b_0\\
A_{1,0} & A_{1,1} & \cdots & A_{1,n-1} & b_1\\
\vdots & \vdots & \ddots & \vdots & \vdots\\
A_{n-1,0} & A_{n-1,1} & \cdots & A_{n-1,n-1} & b_{n-1}
\end{array}\right]$$

Poglejmo si postopek Gaussove eliminacije na primeru!

```python
# primer matrike A in vektroja desne strani b

A = np.array([[ 8., -6, 3],
              [-6, 6,-6],
              [ 3, -6, 6]])
b = np.array([-14, 36, 6])
Ab = np.column_stack((A,b))
```

```python
print('A:\n', A)
print('b:\n', b)
print('A|b:\n', Ab)
```

Sedaj je potrebno pridelati obliko matrike, ki je **zgornje trikotna**!

```python
# Eliminacija v prvem stolpcu
# korak 1:
# - prvo vrstico množimo za A[1,0] in delimo z A[0,0]
# - dobljeno vrstico sedaj odštejemo od druge vrstice
# - dobimo ničelni element na prvem mestu v drugi vrstici

k = Ab[1,0]/Ab[0,0]
row_0 = k*Ab[0,:]

print('row_1 start:\n',Ab[1,:])
Ab[1,:] -= row_0
print('row_1 end:\n',Ab[1,:])

print()
print('new AB:\n',Ab)
```

```python
# korak 2:
# - prvo vrstico množimo za A[2,0] in delimo z A[0,0]
# - dobljeno vrstico sedaj odštejemo od tretje vrstice
# - dobimo ničelni element na prvem mestu v tretji vrstici

k = Ab[2,0]/Ab[0,0]
row_0 = k*Ab[0,:]

print('row_2 start:\n',Ab[2,:])
Ab[2,:] -= row_0
print('row_2 end:\n',Ab[2,:])

print()
print('new AB:\n',Ab)
```

```python
# Eliminacija v drugem stolpcu
# korak 1:
# - drugo vrstico množimo za A[2,1] in delimo z A[1,1]
# - dobljeno vrstico sedaj odštejemo od tretje vrstice
# - dobimo ničelni element na drugem mestu v tretji vrstici

k = Ab[2,1]/Ab[1,1]
row_1 = k*Ab[1,:]

print('row_2 start:\n',Ab[2,:])
Ab[2,:] -= row_1
print('row_2 end:\n',Ab[2,:])

print()
print('new AB:\n',Ab)
```

Postopek je končan! Dobili smo zgornje trikotno matriko. Sedaj je potrebno določiti vrednosti neznanega vektorja $\mathbf{x}$.

```python
# rezerviramo prostor za rešitev in jo določimo
x = np.zeros(3) # rezervacija

x[2] = Ab[2,-1]/Ab[2,2] # izračun x_2 je enostaven
x
```

```python
# določimo x_1
x[1] = (Ab[1,-1] - Ab[1,2]*x[2]) / Ab[1,1]
x
```

```python
# določimo x_0
x[0] = (Ab[0,3] - Ab[0,1:3]@x[1:]) / Ab[0,0]
x
```

```python
# preverimo pravilnost rešitve
b_cal = np.matmul(A,x) # lahko uporabite tudi A @ x, ki predstavlja matrično množenje

print('b_cal:\n', b_cal)
print('b - b_cal:\n', b - b_cal)
```

Prikazani algoritem, s katerim smo iz zgornje trikotnega sistema enačb $\mathbf{U}\,\mathbf{x}=\mathbf{b}$ izračunali rešitev, imenujemo **obratno vstavljanje** (angl. *back substitution*); $\mathbf{U}$ je zgornje trikotna matrika.

V kolikor bi reševali sistem $\mathbf{L}\,\mathbf{x}=\mathbf{b}$ in je $\mathbf{L}$ spodnje trikotna matrika, bi to metodo imenovali **direktno vstavljanje** (angl. *forward substitution*).

Reševanje sistema linearnih enačb z `numpy.linalg.solve` (dokumentacija):

`solve(aA, b)`
kjer je `A` matrika koeficientov (ali seznam matrik) in je `b` vektor konstant (ali seznam vektorjev). Funkcija vrne vektor (ali seznam vektorjev) rešitev.

```python
x_np = np.linalg.solve(A, b)

print()
print('Rešitev tako lahko preverimo z našo:')
print('x_np:\n', x_np)
print('x_np - x:\n', x_np-x)
```