Praktikum 2

Christian Jaeger

24.02.2025

Inhaltsverzeichnis

1 LU-Zerlegung: Vor- / Rückwärtseinsetzen

1.1 Ziel

In dieser Lektion implementieren wir das Lösungsverfahren $Ax = b$ unter der Annahme, dass eine LR-Zerlegung $A = LR$ der Matrix A bereits gegeben ist.

Abgabe: Sie geben als Praktikumsbericht ihren Code Python ab. Für die Bewertung ist ausschlaggebend, dass Sie sich ersichtlich mit der Aufgabe auseinandergesetzt haben, unabhängig davon, ob der Code am Ende läuft oder nicht. Wir werden allerdings im nachfolgenden Praktikum darauf aufbauen.

1.2 Theorie

Lineare Gleichungssysteme $Ax = b$ für $A \in \mathbb{R}^{n\times n}, b \in \mathbb{R}^n$ können durch Vorwärts- bzw. Rückwärtseinsetzen gelöst werden, wenn die Matrix $A = R$ eine obere bzw. $A = L$ eine untere Dreiecksmatrix ist. Sie implementieren und testen in diesem Praktikum die beiden Algorithmen. Das *Vorwärtseinsetzen* ist Algorithmus 1 im Skript. Rückwärtseinsetzen funktioniert analog.

Je nach Kontext kann zudem eine effiziente Implementierung entscheidend sein, sowohl memory-efficient (kein unnötiges Belegen von Speicherplatz) wie auch time-efficient. Im einzelnen bedeutet das

- Die LU-Zerlegung wird in-place ausgeführt, wobei die ursprüngliche Matrix A überschrieben wird. Die resultierende Matrix LR enthält im Dreieck links unten die Matrix L, ohne die Diagonalelemente und im Dreieck oben rechts die Matrix R (inklusive Diagonale). Dies enspricht der Vorgehensweise im Unterricht.
- Es werden keine Zeilenvertauschungen explizit durchgeführt, stattdessen wird mit einem Indexvektor gearbeitet.

1.3 Aufgaben

Aufgabe 1

1. Beschreiben Sie das Vorwärtseinsetzen als Pseudocode, analog zu Algorithmus 1 im Skript.
2. Beschreiben Sie in Pseudocode das Vor- und Rückwärtseinsetzen, wenn die LU-Zerlegung in *einer* Matrix LR gegeben ist. Beachten Sie, dass der Vektor x beim Rückwärtseinsetzen überschrieben werden kann. Drücken Sie die Operationen soweit möglich als Skalarprodukte aus.
3. Beschreiben Sie die in-place LU-Zerlegung ohne Zeilentausch als Pseudocode.

Aufgabe 2

Implementieren Sie eine Funktion **fbSubs** für das Vorwärts- und Rückwärtseinsetzen. Versuchen Sie, so wenig Schleifen wie möglich zu verwenden!

[ ]:
```python
import numpy as np

"""
forwardBackwardSubs: Vorwärts- und Rückwärtseinsetzen
in:
 - Matrix LR, die das Ergebnis einer LU-Zerlegung enthält
 - Vektor b: rechte Seite des LGS
out: Lösung x des LGS
"""
def fbSubs(LR, b):
    # code
```

Aufgabe 3

Testen Sie **fbSubs** zuerst mit dem Zahlenbeispiel aus dem Unterricht (wo Sie alles schrittweise nachvollziehen können) und dann mit dem folgenden Testcode, den Sie auch variieren dürfen, bis Sie sicher sind, dass alles wie gewünscht funktioniert. Sie müssen sich im nachfolgenden Praktikum auf **fbSubs** verlassen können!

[ ]:
```python
#Test LR
n = 7 # Dimension der Koeffizientenmatrix
for k in range(1000):                       # Testläufe
    LR = np.array( np.random.rand(n,n) )    # zufällige Matrix LR
    rhs = np.array(np.random.rand(n))     # zufällige rechte Seite des LGS
    x = fbSubs(LR, rhs)    # Aufruf der eigenen Funktion

    L,R = np.tril(LR,-1)+np.identity(n), np.triu(LR) # L und R extrahieren

    assert( np.linalg.norm(rhs - L@R@x) < 1e-10)     # Test, mit numerischer Toleranz
```

2 Implementierung Gauss-Algorithmus mit Spaltenpivotisierung

2.1 Ziel

In diesem Praktikum implementieren wir das Lösungsverfahren $Ax = b$ inklusive LU-Zerlegung $PA = LR$ der Matrix A mit Spaltenpivotisierung.

Abgabe: Sie geben als Praktikumsbericht ihren Code in Python ab. Für die Bewertung ist ausschlaggebend, dass Sie sich ersichtlich mit der Aufgabe auseinandergesetzt haben, unabhängig davon, ob der Code am Ende läuft oder nicht. Sie können den Code direkt in das notebook schreiben oder in einer Entwicklungsumgebung Ihrer Wahl.

2.2 Theorie

Ein lineares Gleichungssystem $Ax = b$ kann gelöst werden, indem die Matrix A in eine untere bzw. obere Dreiecksmatrix zerlegt wird: $A = LR$, falls keine Zeilenvertauschungen notwendig sind bzw. $PA = LR$ mit der zusätzlichen *Permutationsmatrix* P, falls Zeilen vertauscht werden.

Diesen Prozess nennt man auf Deutsch LR-Zerlegung (englisch: LU-factorization, für *lower, upper*). Der Algorithmus ist eine kleine Erweiterung des Gauss-Algorithmus (Algorithmus 2 im Skript). R ist die gewohnte Zeilenstufenform, in l_{ij} merken wir uns zusätzlich, mit welchem Faktor Zeile i von Zeile j subtrahiert wurde. Die Beschreibung des Vorgehens entnehmen Sie dem Unterricht.

Sind Zeilenvertauschungen notwendig (oder auf Grund einer Pivotstrategie erwünscht), so führt die Permutationsmatrix P Buch über diese Vertauschungen: alle Zeilenvertauschungen in A werden auch in der Matrix P übernommen.

Je nach Kontext kann zudem eine effiziente Implementierung entscheidend sein, sowohl memory-efficient (kein unnötiges Belegen von Speicherplatz) wie auch time-efficient. Im einzelnen bedeutet das

- Die LU-Zerlegung wird *in-place* ausgeführt, wobei die ursprüngliche Matrix A überschrieben wird. Die resultierende Matrix LR enthält im Dreieck links unten die Matrix L, ohne die Diagonalelemente und im Dreieck oben rechts die Matrix R (inklusive Diagonale). Dies enspricht der Vorgehensweise im Unterricht.
- Zeilenvertauschungen können formal durch eine Permutationsmatrix P erfasst werden. Es gilt dann $LR = PA$ und man hat wegen $Ax = b$ <=> $PAx = Pb$ die beiden LGS

  $L \cdot y = P \cdot b$ und $R \cdot x = y$

zu lösen. Um die Matrixmultiplikation mit der Permutationsmatrix zu sparen arbeitet man stattdessen mit einem *Indexvektor* i, der zu Beginn auf $i = [0, 1, ..., n-1]$ initialisiert wird. Werden Zeilen in A getauscht, so tauscht man auch die entsprechenden Einträge in i und löst am Ende die beiden LGS

$L \cdot y = b[i]$ und $R \cdot x = y$

Ziel dieses Praktikums ist eine Funktion **linsolve**, die LGS $A \cdot x = b$ mit Hilfe einer *in-place LU-Zerlegung* mit Spaltenpivotisierung löst. Wir verwenden die Ergebnisse des letzten Prakikums:

2.3 Aufgaben

Aufgabe 1

Implementieren Sie die *LU-Zerlegung*: die Funktion **LU** nimmt als Input eine quadratische Matrix A und gibt Dreiecksmatrizen L, R zurück, so dass $LR = A$ gilt. Im ersten Schritt lassen wir die Pivotisierung weg. Der Indexvektor **idx** bleibt dann unverändert.

[ ]:
```
# LU-Zerlegung der quadratischen Matrix A
# in: quadratische Matrix A
#out:
# - A wird überschrieben, unteres Dreieck = L (ohne Diagonale), oberes Dreieck = R
# - idx: Indexvektor der Zeilenvertauschungen
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
def LU(A):
    m = A.shape[0]
    idx = np.array(range(m))
    # code
    return A, idx
```

Testen Sie **LU** mit zufällig erzeugten Matrizen.

[ ]:
```
n = 7
#test LU
for k in range(1000):
    A = np.array( np.random.rand(n,n) )   # zufällige Matrix A erzeugen
    LR, idx = LU(A.copy())                  # LU-Zerlegung von A
    L,R = np.tril(LR,-1)+np.identity(n), np.triu(LR)  # Matrizen L, R extrahieren
    assert( np.linalg.norm(L@R - A[idx]) < 1e-8)
```

Aufgabe 2

Erstellen Sie eine Funktion **linsolve(A, b)**, die für eine Matrix A und einen Vektor b das LGS $Ax = b$ mit Hilfe der *LR-Zerlegung* und dem Vorwärts- und Rückwärtseinsetzen löst.

[ ]:
```
# lineares Gleichungssystem A*x = b lösen.
def linsolve(A, b):
    # code
```

Testen Sie **linsolve** mit zufällig erzeugten Matrizen

[ ]:
```
#test linsolve
for k in range(1000):
    A = np.random.rand(n,n)
    rhs = np.random.rand(n)
    x = linsolve(A.copy(), rhs)
    assert( lin.norm(rhs - A @ x) < 1e-10)
```

Mit der folgenden Funktion *rndCond(n, cond)* können Sie zufällige $n \times n$ Matrizen mit vorgegebener Konditionszahl *cond* erzeugen.

[ ]:
```
import numpy.linalg as lin
import numpy.random as rnd

# random orthogonal matrix
def rndOrtho(n):
    S = rnd.rand(n,n)
    S = S - S.T
    O = lin.solve(S - np.identity(n), S + np.identity(n))
    return O

# random matrix with specified condition number
def rndCond(n, cond):
    d = np.logspace(-np.log10(cond)/2, np.log10(cond)/2,n);
    A = np.diag(d)
    U,V = rndOrtho(n), rndOrtho(n)
    return U@A@V.T
```

Verwenden Sie diese, um Ihre Implementation von *linsolve* zu testen. Berechnen Sie den relativen Fehler $\frac{|\Delta_x|}{|x|}$ und vergleichen Sie mit dem Ergebnis von *linsolve* aus der Bibliothek *numpy.linalg*.

[ ]:
```
for k in range(N):
    A = rndCond(n, 1e14)
    # code
```

Aufgabe 3

Implementieren Sie die Spaltenpivotisierung und wiederholen Sie die Testläufe aus Aufgabe 2

3 LU-Zerlegung - Step by step

Als Einleitung zum eigentlichen Praktikum setzen Sie hier die LU-Zerlegung Schritt für Schritt an einem Beispiel um. Gegeben ist das LGS $A\vec{x} = \vec{b}$, mit $A, \vec{b}$

3.1 Erste Variante: ohne Pivotisierung

Die Matrix A ist vorgegeben. Wir erstellen gleich eine kleine Funktion, die diese Matrix zurückgibt, um später jederzeit Zugriff auf das Original zu haben.

[1]:
```
import numpy as np

def getA():
    return np.array([[3.,6,3],[1,3,6],[6,3,3]])
```

[2]:
```
A = getA()
print(A)
```

```
[[3. 6. 3.]
 [1. 3. 6.]
 [6. 3. 3.]]
```

erste Zeilenoperation

Subtrahieren Sie geeignete Vielfache der ersten Zeile von der zweiten und dritten. Berechnen Sie jeweils die benötigten Faktoren aus den Matrixeinträgen!

[3]:
```
A = getA()
L = A[1:,0] / A[0,0]          # Faktoren, diese ergeben nachher die Matrix L
for k in range(1,3):
    A[k] -= L[k-1]*A[0]       # Matrix R
print(A, "\n\n", L)
```

```
[[ 3.  6.  3.]
 [ 0.  1.  5.]
 [ 0. -9. -3.]]

 [0.33333333 2.        ]
```

zweite Zeilenoperation

[4]:
```
L = A[2:,1] / A[1,1]
A[2:3] = A[2:3] - L*A[1]
print(A, "\n\n", L)
```

```
[[ 3.  6.  3.]
 [ 0.  1.  5.]
 [ 0.  0. 42.]]

 [-9.]
```

Damit haben wir die Zeilenstufenform von A - also die Matrix R - gefunden. **Jetzt nochmal von vorn**, aber diesmal erstellen Sie die Matrix L (ohne die Diagonale) im linken unteren Dreieck gleich mit:

[5]:
```
A = getA()
print(A)
```

```
[[3. 6. 3.]
 [1. 3. 6.]
 [6. 3. 3.]]
```

Subtrahieren Sie geeignete Vielfache der ersten Zeile von der zweiten und dritten und Speichern Sie die Faktoren am richtigen Ort

[6]:
```
A = getA()

# erste Zeilenoperation
A[1:,0] = A[1:,0] / A[0,0]
for k in range(1,3):
    A[k,1:] -= A[k,0]*A[0,1:]       # Matrix R
print(A, "\n\n")

# zweite Zeilenoperation
A[2:,1] = A[2:,1] / A[1,1]
A[2:,2:] -= A[2:,1]*A[1,2:]

#Ausgabe
print(A)
```

```
[[ 3.          6.          3.        ]
 [ 0.33333333  1.          5.        ]
 [ 2.         -9.         -3.        ]]


[[ 3.          6.          3.        ]
 [ 0.33333333  1.          5.        ]
 [ 2.         -9.         42.        ]]
```

Wir extrahieren die Matrizen L und R aus dem Resultat

[7]:
```
L = np.tril(A, -1) + np.identity(3)
R = np.triu(A)
print("L=\n", L,"\n\nR=\n", R)
```

```
L=
 [[ 1.          0.          0.        ]
 [ 0.33333333  1.          0.        ]
 [ 2.         -9.          1.        ]]

R=
 [[ 3.  6.  3.]
 [ 0.  1.  5.]
 [ 0.  0. 42.]]
```

Jetzt testen wir, ob tatsächlich $LR = A$ gilt:

[8]:
```
print(L@R - getA())  #sollte die Nullmatrix geben, evtl. bis auf Rundungsfehler ~1e-16
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

3.2 Zweite Variante, mit Pivotisierung

Jetzt nochmal von vorn, diesmal mit Pivotisierung

[9]:
```
A = getA()

print(A)
```

```
[[3. 6. 3.]
 [1. 3. 6.]
 [6. 3. 3.]]
```

Initialisieren Sie einen Indexvektor *idx = [0,1,2]*. Dieser Vektor enthält am Ende der LU-Zerlegung die (neue) Reihenfolge der Zeilen.

[10]:
```
idx = np.arange(3)
idxc = idx.copy()
print(idx)
```

```
[0 1 2]
```

Als **Vorübung**: bestimmen Sie die Zeile p mit dem betragsmässig grössten Element der ersten Spalte von A. *Hinweis: **np.argmax**

[11]:
```
p = np.argmax(np.abs(A[:,0]))
print(p)
```

```
2
```

Tauschen Sie in der Matrix A die erste Zeile (Index 0) mit der Zeile p und ebenso den ersten Eintrag von **idx** mit dem Eintrag p

[12]:
```
A = getA()
idx = np.arange(3)
A[[0,p]] = A[[p,0]]
idx[[0,p]]= idx[[p,0]]
print("A=\n", A, "\n\nidx=\n", idx)
```

```
A=
 [[6. 3. 3.]
 [1. 3. 6.]
 [3. 6. 3.]]

idx=
 [2 1 0]
```

Jetzt führen Sie die erste Zeilenoperation mit der aktuellen Matrix A aus und speichern Sie die Faktoren L in der unteren linken Dreiecksmatrix

[13]:
```
# erste Zeilenoperation
A[1:,0] = A[1:,0] / A[0,0]
for k in range(1,3):
    A[k,1:] -= A[k,0]*A[0,1:]       # Matrix R
print(A)
```

```
[[6.         3.         3.        ]
 [0.16666667 2.5        5.5       ]
 [0.5        4.5        1.5       ]]
```

bestimmen Sie die Zeile p mit dem betragsmässig grössten Element der zweiten Spalte von A, von der zweiten Zeile an

[14]:
```
p = 1 + np.argmax(np.abs(A[1:,1]))
print(p)
```

```
2
```

Tauschen Sie die entsprechenden Zeilen von A (das betrifft die Anteile L und R gleichermassen)

[15]:
```
A[[1,p]] = A[[p,1]]
idx[[1,p]]= idx[[p,1]]
print("A=\n", A, "\n\nidx=\n", idx)
```

```
A=
 [[6.         3.         3.        ]
 [0.5        4.5        1.5       ]
 [0.16666667 2.5        5.5       ]]

idx=
 [2 0 1]
```

Jetzt führen Sie die zweite Zeilenoperation mit der aktuellen Matrix A aus und speichern Sie wiederum L

[16]:
```
# zweite Zeilenoperation
A[2:,1] = A[2:,1] / A[1,1]
A[2:,2:] -= A[2:,1]*A[1,2:]
```

[17]:
```
print(A)
```

```
[[6.         3.         3.        ]
 [0.5        4.5        1.5       ]
 [0.16666667 0.55555556 4.66666667]]
```

Jetzt sind wir fertig. Wir extrahieren (testhalber) die Matrizen L, R

[18]:
```
L = np.tril(A, -1) + np.identity(3)
R = np.triu(A)
print("L=\n", L,"\n\nR=\n", R)
```

```
L=
 [[1.         0.         0.        ]
 [0.5        1.         0.        ]
 [0.16666667 0.55555556 1.        ]]

R=
 [[6.         3.         3.        ]
 [0.         4.5        1.5       ]
 [0.         0.         4.66666667]]
```

jetzt sollte $LR = A$ sein:

[19]:
```
print(L@R-getA()[idx])
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

Downloads:

- PDF-Dokumentation:
  - `Anleitung Praktikum 2`
- Jupyter-Notebooks:
  - `Jupyter-Notebook`
  - `Jupyter-Notebook LR-Zerlegung - Step by step`