Praktikum 3a

Christian Jaeger

24.02.2025

Inhaltsverzeichnis

1 LU-Zerlegung für Tridiagonalmatrizen

1.1 Theorie und Ziele

Für *Tridiagonalmatrizen* lässt sich die *LU-Zerlegung* effizienter durchführen, als für vollbesetzte Matrizen. Solche Matrizen treten bei der numerischen Lösung von eindimensionalen Randwertproblemen (RWP) auf. Wir betrachten exemplarisch

$$-u''(x) = f(x), \quad (0 < x < 1), \qquad u(0) = u_0, u(1) = u_n$$

für vorgegebene Randwerte u_0, u_n. Durch **Diskretisierung** lässt sich daraus ein LGS mit tridiagonaler Koeffizentenmatrix A für die Stützwerte $u_i = u(x_i)$ gewinnen. Wir teilen dazu das Intervall $[0, 1]$ in n Intervalle der Länge $h > 0$ an den Teilungspunkten $x_0 = 0, x_1 = x_0 + h, \ldots, x_n = 1$. Daraus entsteht ein tridiagonales LGS $Au = b$ (s. Unterricht).

Ziel des Praktikums ist es, die Algorithmen für Tridiagonalmatrizen zu implementieren und an einem Beispiel zu testen. Obwohl A für die hier betrachtete Problemklasse immer gleich aussieht (2 auf der Hauptdiagonalen, -1 auf den zwei Nebendiagonalen) soll die LU-Zerlegung allgemein für Tridiagonalmatrizen umgesetzt werden.

Da fast alle Einträge in A verschwinden, wird die Matrix A nicht als Matrix gespeichert, sondern nur die drei Diagonalen als Vektoren. *Eine* Möglichkeit besteht darin, eine $3 \times n$ Matrix zu verwenden (aber es gibt natürlich viele andere Varianten):

$$M = \begin{pmatrix} 0 & a_{21} & \ldots & a_{n-1,n-2} & a_{n,n-1} \\ a_{11} & a_{22} & \ldots & a_{n-1,n-1} & a_{nn} \\ a_{12} & a_{23} & \ldots & a_{n-1,n} & 0 \end{pmatrix}$$

in diesem Fall gilt

$$a_{j,j-1} = m_{1,j}, (2 \le j \le n) \qquad a_{j,j} = m_{2,j}, (1 \le j \le n) \qquad a_{j,j+1} = m_{3,j}, (1 \le j \le n-1)$$

in Python ist zusätzlich zu berücksichtigen, dass die Indices bei 0 starten. Das Resultat der Zerlegung ist ebenfalls tridiagonal und kann auf dieselbe Art gespeichert werden.

1.2 Aufgaben

Bemerkungen zu den Aufgaben:

- Eventuell fällt Ihnen die Aufgabe 1 einfacher, wenn Sie den Algorithmus 2.6 zuerst für vollbesetzte Matrizen implementieren und danach für kompakt gespeicherte tridiagonal Matrizen optimieren.
- Ebenso in der Aufgabe 2. Optimieren Sie erst danach die Summe, in dem Sie die Null Einträge bei tridiagonal Matrizen eliminieren.

Aufgabe 1

Implementieren Sie die LU-Zerlegung effizient für Tridiagonalmatrizen (Algorithmus 2.6). Sie dürfen die Schnittstelle anpassen, wenn Sie das sinnvoll finden.

[ ]:
```
import numpy as np
"""
LU decomposition for tridiagonal matrix
in: a  =  [[0,      a_{21}, ..., a_{n-1,n-2}, a_{n,n-1}],
           [a_{11}, a_{22}, ..., a_{n-1,n-1}, a_{nn}],
           [a_{12}, a_{23}, ..., a_{n-1,n},   0]]

out: LU = [[0,      l_{21}, ..., l_{n-1,n-2}, l_{n,n-1}],
           [r_{11}, r_{22}, ..., r_{n-1,n-1}, r_{nn}],
           [r_{12}, r_{23}, ..., r_{n-1,n},   0]]
"""
def LUT(m):
    # code
    return LU
```

Testen Sie Ihre Umsetzung. Der folgende Testcode funktioniert, falls die Tridiagonalmatrix wie in der Einleitung beschrieben gespeichert wurde.

[ ]:
```
import numpy as np
n=5 # Grösse der Matrizen
# test LUT
for k in range(1000):
    m = np.random.rand(3,n)      # Zufällige Matrix M erzeugen
    m[0][0], m[-1][-1] = 0, 0    # nicht verwendete Einträge löschen
    A = np.diag(m[0][1:], k=-1) + np.diag(m[1], k=0) + np.diag(m[2][:-1], k=1) # volle␣
↪Matrix A erzeugen (nur für Test)

    LU = LUT(m)

    L,U = np.diag(LU[0][1:] , k=-1)+ np.identity(n), np.diag(LU[1], k=0) + np.
↪diag(LU[2][:-1], k=1) # L, U Matrizen
    assert(np.linalg.norm(L@U-A) < 1e-10)
```

Aufgabe 2

Implementieren Sie die Vorwärts- und Rücksubstitution effizient für Tridiagonalmatrizen

[ ]:
```
"""
in: LU (output from LUT), vector b
out: vector x s.t. L@U@x == b
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
"""
def fbSubsT(LU, b):
    # code
    return x
```

Testen Sie Ihre Umsetzung. Der folgende Testcode ist wiederum auf die oben beschriebene Speicherung der Matrizen ausgelegt.

[ ]:
```
# test fbSubsT
for k in range(1000):
    m = np.random.rand(3,n)
    m[0][0], m[-1][-1] = 0, 0
    A = np.diag(m[0][1:], k=-1) + np.diag(m[1], k=0) + np.diag(m[2][:-1], k=1)

    x1 = np.random.rand(n)   # Lösungsvektor
    b = A@x1                   # rechte Seite des LGS

    LU = LUT(m)
    x2 = fbSubsT(LU, b)

    assert(np.linalg.norm(x1-x2) < 1e-10)
```

Aufgabe 3

Wenden Sie die oben implementierten Algorithmen auf das in der Einleitung genannte RWP an, plotten Sie die numerische Lösung zusammen mit der exakten Lösung. Die tridiagonale Matrix des LGS ist nun gegeben durch die finite Differenzen Diskretisierung (Beispiel 2.6 im Skript).

[ ]:
```
n = 100
x = np.linspace(0,1,n+1)
```

[ ]:
```
# Dirichlet Randwerte
u0 = 0
un = 0
```

[ ]:
```
# System Matrix
A = np.zeros((3,n-1))
A[0,1:] = <<snipp>>
A[1,:] = <<snipp>>
A[2,:-1] = <<snipp>>
```

Im Beispiel benutzen wir $f(x) = 1$.

[ ]:
```
# Rechte Seite
h = 1./n
b = <<snipp>>;
b[0] = <<snipp>>;
b[-1] = <<snipp>>;
```

Lösung berechnen und visualisieren:

[ ]:
```python
import matplotlib.pyplot as plt

LU = LUT(A)
u = np.zeros((n+1))
u[0] = u0;   # Randwert links
u[-1] = un;  # Randwert rechts
u[1:-1] = fbSubsT(LU, b)

ue = -0.5*x*(x-1); # Loesung von u''(x) = 1, u(0) = u(1) = 0

plt.plot(x, u)
plt.plot(x, ue,'--')
plt.grid()
plt.show()
```

Abgabe

- Aufgabe 1: Funktion zur Berechnung der LU-Zerlegung für tridiagonal Matrizen
- Aufgabe 2: Funktion zum Lösen des Gleichungssystem mittels Vorwärts- / Rückwärtseinsetzen
- Aufgabe 3: Anwendung auf ein eindimensionales Randwertproblem

Kurzer Bericht mit den Ergebnissen und python Code.

Downloads:

- PDF-Dokumentation:
  - `Anleitung Praktikum 3a`
- Jupyter-Notebooks:
  - `Jupyter-Notebook`