Übungsserie 10

Abgabe: gemäss Angaben Dozent

Scannen Sie ihre manuellen Lösungen für die Aufgaben 1 und 2 in die Dateien *Name_S10_Aufg1.pdf* bzw. *Name_S10_Aufg2.pdf* und fassen Sie diese mit Ihrer Python-Funktion *Name_S10_Aufg3a.py* und dem Skript *Name_S10_Aufg3b.py* in einer ZIP-Datei *Name_S10.zip* zusammen. Laden Sie dieses File vor der Übungsstunde nächste Woche auf Moodle hoch.

Aufgabe 1 (ca. 45 Minuten):

Gegeben ist das lineare Gleichungssystem

$$Ax = b \text{ mit } A = \begin{pmatrix} 8 & 5 & 2 \\ 5 & 9 & 1 \\ 4 & 2 & 7 \end{pmatrix} \text{ und } b = \begin{pmatrix} 19 \\ 5 \\ 34 \end{pmatrix}.$$

a) Überprüfen Sie, ob das obige System bzgl. dem Jacobi-Verfahren konvergiert.

b) Berechnen Sie auf vier Stellen nach dem Komma die Näherung $x^{(3)}$ mit dem Jacobi-Verfahren ausgehend vom Startvektor $x^{(0)} = \begin{pmatrix} 1 \\ -1 \\ 3 \end{pmatrix}$. Schreiben Sie alle benötigten Matrizen sowie die verwendete Iterationsgleichung explizit auf. Die Iterationen selber führen Sie aber natürlich mit Python durch.

c) Wie gross ist gemäss der a-posteriori Abschätzung der absolute Fehler von $x^{(3)}$?

d) Schätzen Sie a-priori die Anzahl Iterationsschritte ab, damit der berechnete Näherungsvektor in jeder Kompnente maximal um 10^{-4} von der exakten Lösung $x = (2, -1, 4)^T$ abweicht.

e) Wiviele Iterationsschritte würden Sie a-priori benötigen, wenn Sie als Startvektor nicht $x^{(0)}$ sondern $x^{(2)}$ aus b) verwenden würden?

Aufgabe 2 (ca. 30 Minuten):

Wiederholen Sie die obige Aufgabe, diesmal für das Gauss-Seidel Verfahren. Sie dürfen (ausnahmsweise) die Inverse von $D + L$ benutzen (müssen aber nicht, wenn Sie nicht wollen).

Aufgabe 3 (ca. 75 Minuten):

a) Implementieren Sie das Jacobi- und Gauss-Seidel-Verfahren zusammen in einer Funktion als `[xn, n, n2] = Name_S10_Aufg3a(A,b,x0,tol,opt)`. Sie können dabei die Matrix-Funktionen von `numpy` und `numpy.linalg` in Python benutzen, (z.B. `triu(A), diag(diag(A)), tril(A), inv(D+L)`), ohne aber `inv(A)` zu berechnen. Dabei soll `xn` der Iterationsvektor nach `n` Iterationen sein, zusätzlich soll `n2` die Anzahl benötigter Schritte gemäss der a-priori Abschätzung angeben. Über den Parameter `opt` soll gesteuert werden, ob das Jacobi- oder das Gauss-Seidel Verfahren zur Anwendung kommt. Überlegen Sie sich, wie die Abbruchbedingung für Ihre while-Schleife lauten muss, um die Iteration bei Erreichen einer vorgegebenen Fehlertoleranz `tol` abzubrechen. Sie werden dafür

Aufgabe 1 (ca. 45 Minuten):

Gegeben ist das lineare Gleichungssystem

$$Ax = b \text{ mit } A = \begin{pmatrix} 8 & 5 & 2 \\ 5 & 9 & 1 \\ 4 & 2 & 7 \end{pmatrix} \text{ und } b = \begin{pmatrix} 19 \\ 5 \\ 34 \end{pmatrix}.$$

a) Überprüfen Sie, ob das obige System bzgl. dem Jacobi-Verfahren konvergiert.

b) Berechnen Sie auf vier Stellen nach dem Komma die Näherung $x^{(3)}$ mit dem Jacobi-Verfahren ausgehend vom Startvektor $x^{(0)} = \begin{pmatrix} 1 \\ -1 \\ 3 \end{pmatrix}$. Schreiben Sie alle benötigten Matrizen sowie die verwendete Iterationsgleichung explizit auf. Die Iterationen selber führen Sie aber natürlich mit Python durch.

c) Wie gross ist gemäss der a-posteriori Abschätzung der absolute Fehler von $x^{(3)}$?

d) Schätzen Sie a-priori die Anzahl Iterationsschritte ab, damit der berechnete Näherungsvektor in jeder Kompnente maximal um 10^{-4} von der exakten Lösung $x = (2, -1, 4)^T$ abweicht.

e) Wiviele Iterationsschritte würden Sie a-priori benötigen, wenn Sie als Startvektor nicht $x^{(0)}$ sondern $x^{(2)}$ aus b) verwenden würden?

a.) $A = \begin{pmatrix} 8 & 5 & 2 \\ 5 & 9 & 1 \\ 4 & 2 & 7 \end{pmatrix}$ A ist Diagonaldominant, also konvergiert es.

$D = \begin{pmatrix} 8 & 0 & 0 \\ 0 & 9 & 0 \\ 0 & 0 & 7 \end{pmatrix}$ $D^{-1} = \begin{pmatrix} \frac{1}{8} & 0 & 0 \\ 0 & \frac{1}{9} & 0 \\ 0 & 0 & \frac{1}{7} \end{pmatrix}$

$L = \begin{pmatrix} 0 & 0 & 0 \\ 5 & 0 & 0 \\ 4 & 2 & 0 \end{pmatrix}$ $R = \begin{pmatrix} 0 & 5 & 2 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$

$$B = -D^{-1}(L+R) = -\begin{pmatrix} \frac{1}{8} & 0 & 0 \\ 0 & \frac{1}{9} & 0 \\ 0 & 0 & \frac{1}{7} \end{pmatrix} \cdot \begin{pmatrix} 0 & 5 & 2 \\ 5 & 0 & 1 \\ 4 & 2 & 0 \end{pmatrix} = \begin{pmatrix} 0 & -\frac{5}{8} & -\frac{1}{4} \\ -\frac{5}{9} & 0 & -\frac{1}{9} \\ -\frac{4}{7} & -\frac{2}{7} & 0 \end{pmatrix}$$

$\|B\|_\infty = \max\left\{\frac{7}{8}, \frac{6}{9}, \frac{6}{7}\right\} = \frac{7}{8} < 1$ ✓ konvergiert

b) Berechnen Sie auf vier Stellen nach dem Komma die Näherung $x^{(3)}$ mit dem Jacobi-Verfahren ausgehend vom Startvektor $x^{(0)} = \begin{pmatrix} 1 \\ -1 \\ 3 \end{pmatrix}$. Schreiben Sie alle benötigten Matrizen sowie die verwendete Iterationsgleichung

$$x_1^{(k+1)} = a_{11} \cdot x_1^{(k)} + a_{12} \cdot x_2^{(k)} + a_{13} \cdot x_3^{(k)}$$
$$x_2^{(k+1)} = a_{21} \cdot x_1^{(k)} + a_{22} \cdot x_2^{(k)} + a_{23} \cdot x_3^{(k)}$$
$$x_3^{(k+1)} = a_{31} \cdot x_1^{(k)} + a_{32} \cdot x_2^{(k)} + a_{33} \cdot x_3^{(k)}$$

$$\boldsymbol{x}^{(n+1)} = -\boldsymbol{D}^{-1}((\boldsymbol{L}+\boldsymbol{R})\,\boldsymbol{x}^{(n)} - \boldsymbol{b})$$

$$\vec{x}^{(k+1)} = \underbrace{-D^{-1}(L+R)\vec{x}^{(k)}}_{B\vec{x}^{(k)}} + D^{-1}\cdot\vec{b} \qquad + \vec{c}$$

$$\vec{x}^{(1)} = \begin{pmatrix} 0 & -\frac{5}{8} & -\frac{1}{4} \\ -\frac{5}{9} & 0 & -\frac{1}{9} \\ -\frac{4}{7} & -\frac{2}{7} & 0 \end{pmatrix} \cdot \underbrace{\begin{pmatrix} 1 \\ -1 \\ 3 \end{pmatrix}}_{x^{(0)}} + \begin{pmatrix} \frac{1}{8} & 0 & 0 \\ 0 & \frac{1}{9} & 0 \\ 0 & 0 & \frac{1}{7} \end{pmatrix} \cdot \begin{pmatrix} 19 \\ 5 \\ 34 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{5}{8} - \frac{3}{4} \\ -\frac{5}{9} - \frac{3}{9} \\ -\frac{4}{7} + \frac{2}{7} \end{pmatrix} + \begin{pmatrix} \frac{19}{8} \\ \frac{5}{9} \\ \frac{34}{7} \end{pmatrix} = \begin{pmatrix} -\frac{1}{8} + \frac{19}{8} \\ -\frac{8}{9} + \frac{5}{9} \\ -\frac{2}{7} + \frac{34}{7} \end{pmatrix} = \underbrace{\begin{pmatrix} \frac{18}{8} \\ -\frac{1}{3} \\ \frac{32}{7} \end{pmatrix}}_{\vec{x}^{(1)}} = \begin{pmatrix} 2.5 \\ -0.\overline{3} \\ 4.571 \end{pmatrix}$$

$\vec{x}^{(2)} =$ // Python

$\vec{x}^{(3)} =$ // $\begin{pmatrix} 2.209 \\ -0.65 \\ 4.377 \end{pmatrix}$ $\lim_{k\to\infty} \vec{x}^{(k)} = -\begin{pmatrix} 2 \\ -1 \\ 4 \end{pmatrix}$ exakte Lösung

$$\| \boldsymbol{x}^{(n)} - \bar{\boldsymbol{x}} \| \leq \frac{\| \boldsymbol{B} \|}{1 - \| \boldsymbol{B} \|} \| \boldsymbol{x}^{(n)} - \boldsymbol{x}^{(n-1)} \| \quad \text{a-posteriori Abschätzung}$$

Python

c.) $\|B\|_\infty = \frac{7}{8}$ $\qquad \|x^{(3)} - x^{(2)}\| = \begin{pmatrix} 2.209 - 1.44 \\ -0.65 + 1.20 \\ 4.37 - 3.66 \end{pmatrix} = \left\| \begin{pmatrix} 0.769 \\ 0.55 \\ 0.71 \end{pmatrix} \right\|_\infty = 0.769$

$$\frac{\frac{7}{8}}{1 - \frac{7}{8}} \cdot 0.769 = \frac{\frac{7}{8}}{\frac{1}{8}} \cdot 0.769 = \underline{\underline{5.383}}$$

d) Schätzen Sie a-priori die Anzahl Iterationsschritte ab, damit der berechnete Näherungsvektor in jeder Kompnente maximal um 10^{-4} von der exakten Lösung $x = (2, -1, 4)^T$ abweicht.

$\| x^{(n)} - \bar{x} \| \leq \frac{\| B \|^n}{1 - \| B \|} \| x^{(1)} - x^{(0)} \|$ a-priori Abschätzung

$\| x^{(1)} - x^{(0)} \|_\infty = \begin{pmatrix} 2.25 & -1 \\ -\frac{1}{3} & +1 \\ 4.571 & -3 \end{pmatrix} = \begin{pmatrix} 1.25 \\ \frac{2}{3} \\ 1.571 \end{pmatrix}$

$= 1.571$

$10^{-4} \leq \frac{\left(\frac{7}{8}\right)^n}{1 - \frac{7}{8}} \cdot 1.571 \quad | \cdot 7$

$\frac{10^{-4}}{1.571} \leq \frac{\left(\frac{7}{8}\right)^n}{1 - \frac{7}{8}}$

$\leq \left(\frac{7}{8}\right)^n \quad | \cdot \ln$

$\frac{10^{-4} \cdot \left(1 - \frac{7}{8}\right)}{1.571} \leq \left(\frac{7}{8}\right)^n$

$\ln\left(\frac{1}{125680}\right) \leq n \cdot \ln\left(\frac{7}{8}\right)$

$87.93 \leq n$

es benötigt ca. 88 Iterationen

e) Wiviele Iterationsschritte würden Sie a-priori benötigen, wenn Sie als Startvektor nicht $x^{(0)}$ sondern $x^{(2)}$ aus b) verwenden würden?

a-priori: $\| x^{(n)} - \bar{x} \| \leq \frac{\| B \|^n}{1 - \| B \|} \| x^{(1)} - x^{(0)} \|$

$x^{(2)} = \begin{pmatrix} 1.44 \\ -1.202 \\ 3.67 \end{pmatrix}$

$x^{(1)} - x^{(0)} = \begin{pmatrix} 2.21 \\ -0.65 \\ 4.38 \end{pmatrix} - \begin{pmatrix} 1.44 \\ -1.202 \\ 3.67 \end{pmatrix} = \begin{pmatrix} 0.77 \\ 0.552 \\ 0.71 \end{pmatrix}$

$10^{-4} \leq \frac{\left(\frac{7}{8}\right)^n}{\frac{1}{8}} \cdot 0.77$

$\frac{10^{-4}}{0.77} \leq \frac{\left(\frac{7}{8}\right)^n}{\frac{1}{8}} \Rightarrow \frac{10^{-4} \cdot \frac{1}{8}}{0.77} \leq \left(\frac{7}{8}\right)^n \Rightarrow \frac{1}{61'600} \leq \left(\frac{7}{8}\right)^n \quad | \cdot \ln()$

$\ln\left(\frac{1}{61'600}\right) \leq n \cdot \ln\left(\frac{7}{8}\right) \Rightarrow \frac{\ln\left(\frac{1}{61'600}\right)}{\ln\left(\frac{7}{8}\right)} \leq n$

$82.59 \leq n$ es benötigt 83 Iterationen

Aufgabe 2 (ca. 30 Minuten):

Wiederholen Sie die obige Aufgabe, diesmal für das Gauss-Seidel Verfahren. Sie dürfen (ausnahmsweise) die Inverse von $D + L$ benutzen (müssen aber nicht, wenn Sie nicht wollen).

a) $B = -(D+L)^{-1} R \Rightarrow \text{Python} = \left\| \begin{pmatrix} 0 & -0.625 & -0.25 \\ 0 & 0.34722 & 0.02778 \\ 0 & 0.25794 & 0.13492 \end{pmatrix} \right\|_\infty = \frac{7}{8} < 1$ also konvergiert es

b) $x^{(1)} = \begin{pmatrix} 2.01471 \\ -1.00535 \\ 3.99312 \end{pmatrix}$

c) $\frac{\frac{7}{8}}{1-\frac{7}{8}} \cdot \left\| \begin{pmatrix} 2.01471 \\ -1.00535 \\ 3.99312 \end{pmatrix} - \begin{pmatrix} 2.0510 \\ -1.0173 \\ 3.9746 \end{pmatrix} \right\|_\infty = 7 \cdot \left\| \begin{pmatrix} -0.036 \\ -0.00795 \\ 0.01852 \end{pmatrix} \right\|_\infty = 7 \cdot 0.036 = \underline{0.252}$

d.) gemäss Python 87 Iterationen

e.) gemäss Python 60 Iterationen

Aufgabe 3 (ca. 75 Minuten):

a) Implementieren Sie das Jacobi- und Gauss-Seidel-Verfahren zusammen in einer Funktion als `[xn, n, n2] = Name_S10_Aufg3a(A,b,x0,tol,opt)`. Sie können dabei die Matrix-Funktionen von `numpy` und `numpy.linalg` in Python benutzen, (z.B. `triu(A)`, `diag(diag(A))`, `tril(A)`, `inv(D+L)`), ohne aber `inv(A)` zu berechnen. Dabei soll `xn` der Iterationsvektor nach `n` Iterationen sein, zusätzlich soll `n2` die Anzahl benötigter Schritte gemäss der a-priori Abschätzung angeben. Über den Parameter `opt` soll gesteuert werden, ob das Jacobi- oder das Gauss-Seidel Verfahren zur Anwendung kommt. Überlegen Sie sich, wie die Abbruchbedingung für Ihre while-Schleife lauten muss, um die Iteration bei Erreichen einer vorgegebenen Fehlertoleranz `tol` abzubrechen. Sie werden dafür

$n2 = \| x^{(n)} - \bar{x} \| \leq \frac{\| B \|^n}{1 - \| B \|} \| x^{(1)} - x^{(0)} \|$

$tol \leq \frac{B^n}{1-B} \cdot (x^1 - x^0)$

$\frac{tol(1-B)}{(x^1-x^0)} \leq B^n \quad | \cdot \ln()$

$\ln\left(\frac{tol(1-B)}{(x^1-x^0)}\right) \leq n \cdot \ln(B) \Rightarrow \frac{\ln\left(\frac{tol(1-B)}{x^1-x^0}\right)}{\ln(B)} \leq n$

die Norm brauchen: `norm(...,np.inf)`. Achten sie darauf, dass Sie Matrizen, die Sie in ihrer Funktion nicht mehr brauchen, gleich wieder löschen, um Speicher freizugeben[1].

b) Schreiben Sie ein kurzes Skript `Name_S10_Aufg3b.m`. Testen Sie damit die Laufzeit Ihres Programmes für ihre Implementation des Jacobi- und Gauss-Seidel im Vergleich zum Gauss-Verfahren, welches Sie in Serie 6 implementiert hatten (siehe *Name_S6_Aufg2.m*) und im Vergleich zur Python-Funktion np.linalg.solve(). Verwenden Sie dafür die folgenden Werte für A, b, x_0 und tol:

```
>> dim = 3000
>> A = np.diag(np.diag(np.ones((dim,dim))*4000))+np.ones((dim,dim))
>> dum1 = np.arange(1,np.int(dim/2+1),dtype=np.float64).reshape((np.int(dim/2),1))
>> dum2 = np.arange(np.int(dim/2),0,-1,dtype=np.float64).reshape((np.int(dim/2),1))
>> x = np.append(dum1,dum2,axis=0)
>> b = A@x
>> x0 = np.zeros((dim,1))
>> tol = 1e-4
```

Den Zeitvergleich können Sie dabei analog wieder mit `timeit` messen (verzichten Sie auf 'repeat', da die Gauss-Zerlegung einige Zeit braucht ... lassen Sie die Gauss-Zerlegung deshalb nur laufen, wenn Sie Ihren Computer für einige Minuten nicht für anderes brauchen.).

Wieviel länger braucht Ihre eigene Gauss-Zerlegung als z.B. das Gauss-Seidel Verfahren? Schreiben Sie die gemessenen Werte als Kommentar in Ihr Programm.

c) Sie haben bei b) die "exakte" Lösung x definiert. Plotten Sie den absoluten Fehler für jedes Vektorelment ihrer drei Lösungsvektoren (d.h. Gauss, Jacobi, Gauss-Seidel). Was stellen Sie fest? Schreiben Sie Ihren Kommentar ins Skript.

[1] In Python gibt es, im Gegensatzu zu Matlab, keinen expliziten Befehl wie z.B. `clear D` , um alloziiertes Memory für eine Matrix D zur Laufzeit wieder freizugeben. Der Befehl `del` in Python löscht lediglich die Verbindung zum Memoryspace, aber nicht die Variable selbst. Was hingegen funktioniert, ist eine nicht mehr gebrauchte Matrix zu überschreiben, z.B. mit `D = 0`, so dass der Garbage Collector das Memory wieder freigibt.