Lehrstuhl für Systemsimulation

Sommersemester 2025

Prof. Dr. Harald Köstler, Frederik Hennig, Michael Zikeli

Algorithmik kontinuierlicher Systeme
Aufgabenblatt 9 — Numerische Löser in der Praxis

- Zu diesem Übungsblatt steht im StudOn-Kurs ein Zip-Verzeichnis mit Material bereit. Laden Sie dieses herunter und entpacken Sie es, bevor Sie mit den Aufgaben beginnen.
- Zu jeder Aufgabe gehört ein Python-Modul, dessen Name im Aufgabentitel gelistet ist. Füllen Sie die darin enthaltenen Funktions-Stubs mit Ihren Lösungen und geben Sie die Dateien über StudOn ab. Es handelt sich hierbei um **Einzelabgaben**. Sie können Ihre abgegebenen Dateien beliebig oft aktualisieren – nur die letzte abgegebene Version wird gewertet. **Laden Sie ihre Lösungen einzeln, oder als Zip-Archiv hoch, und ändern Sie nicht die Namen der Python-Dateien.**
- Das Material zum Übungsblatt enthält außerdem ein Jupyter Notebook zur interaktiven Entwicklung Ihrer Lösungen, sowie eine automatisierte Test-Suite. Jede Teilaufgabe wird anhand von einer Reihe an Tests automatisch bewertet. Die *öffentlichen* Testfälle können Sie entweder in dem mitgelieferten Notebook, oder durch das Skript `blatt**_tests.py` auf der Konsole ausführen. Denken Sie daran, dass das Bestehen der öffentlichen Tests keine Garantie für Korrektheit ist.

Aufgabe 1 — Dünnbesetzte Lineare Algebra (12 Punkte) `sparse_linalg.py`

In dieser Aufgabe befassen wir uns mit iterativen Lösungsverfahren für dünnbesetzte Matrizen. Zu diesem Zweck stellen wir Matrizen im Compressed Row Storage (CRS)-Format dar. In den ersten Teilaufgaben realisieren Sie die Konstruktion und Matrix-Vektor-Multiplikation für CRS-Matrizen. Anschließend implementieren Sie das Jacobi- und das SOR-Verfahren, sowie das Verfahren der Konjugierten Gradienten auf Basis der CRS-Datenstruktur.

Hinweis: In dieser und der folgenden Aufgabe werden CRS-Matrizen durch den durch die vorgegebene Klasse `CrsMatrix` repräsentiert. Auf die Felder `val`, `col_idx` und `row_ptr` kann dabei namentlich (z.B. `A.col_idx`) zugegriffen werden. Bei `val`, `col_idx` und `row_ptr` handelt es sich wiederrum um NumPy-Arrays vom Elementtyp `np.float64` beziehungsweise `np.int64`. Eine neue `CrsMatrix` muss stets mit dem Konstruktor `CrsMatrix(val, col_idx, row_ptr)` erzeugt werden.

a) CRS-Matrix erzeugen (2 Punkte) Implementieren Sie die Funktion `crs_assemble` zur Erzeugung der CRS-Datenstruktur einer quadratischen Matrix. Sie erhält als Parameter die Zeilen- und Spaltenanzahl `n`, sowie eine Callback-Funktion `row_callback`, welche pro Zeile die Einträge der Matrix liefert. `row_callback` nimmt als einziges Argument einen Zeilenindex `j` entgegen, und gibt die Einträge der `j`-ten Zeile als Liste von Tupeln der Gestalt `(val, col_idx)` zurück.

b) CRS Matrix-Vektor-Multiplikation (1 Punkt) Die Funktion `crs_mvm` erhält die Matrix $\boldsymbol{A}$ im CRS-Format, sowie einen Vektor $\boldsymbol{x}$ als eindimensionales NumPy-Array, und soll das Matrix-Vektor-Produkt $\boldsymbol{Ax}$ berechnen. Implementieren Sie die Matrix-Vektor-Multiplikation mit dem für CRS aus der Vorlesung bekannten Algorithmus. Die Matrix $\boldsymbol{A}$ soll dabei explixit nicht rekonstruiert werden!

c) Jacobi-Iterationsschritt (3 Punkte) Implementieren Sie die Funktion `jacobi_step`, welche einen Iterationsschritt des Jacobi-Verfahrens durchführt. Die Funktion erhält die Systemmatrix $\boldsymbol{A}$ als CRS-Datenstruktur, sowie die rechte Seite $\boldsymbol{b}$ und den Startvektor $\boldsymbol{x}_0$ als 1D NumPy-Arrays. Ihre Implementierung soll die Approximationslösung nach einem Iterationsschritt zurückgeben.

d) SOR-Iterationsschritt (3 Punkte) Realisieren Sie nun die Funktion `sor_step`, welche einen Iterationsschritt des *Successive Over-Relaxation*-Verfahrens ausführt. Die Funktion erhält die Systemmatrix $\boldsymbol{A}$ als CRS-Datenstruktur; die rechte Seite $\boldsymbol{b}$ und den Startvektor $\boldsymbol{x}_0$ als 1D NumPy-Arrays; sowie das Relaxationsgewicht $w \in (0.5, 2)$. Ihre Implementierung soll die Approximationslösung nach einem Iterationsschritt zurückgeben.

e) CG-Verfahren (3 Punkte) Das aus der Vorlesung bekannte Verfahren der Konjugierten Gradienten (CG-Verfahren) kann als iteratives Verfahren zum Lösen großer linearer Gleichungssysteme $\boldsymbol{Ax} = \boldsymbol{b}$ mit symmetrisch positiv definiter Systemmatrix $\boldsymbol{A}$ interpretiert werden. Anders als für Fixpunktverfahren (wie Jacobi und Gauss-Seidel) kann für CG garantiert werden, dass die exakte Lösung (bei fehlerfreier Rechnung) nach spätestens n Schritten erreicht wird. Bei vielen Systemen ist aber schon eine geringere Iterationszahl für eine brauchbare Lösung ausreichend.

Das CG-Verfahren kann als Dreiterm-Rekursion mit den Iterationsvariablen Näherungslösung $\boldsymbol{x_i}$, Suchrichtung $\boldsymbol{d_i}$ und Residuum $\boldsymbol{r_i}$ beschrieben werden. Initial sind

$$\boldsymbol{d_0} = \boldsymbol{r_0} = \boldsymbol{b} - \boldsymbol{Ax_0}.$$

Die Iterationsvorschrift beinhaltet neben skalarer Arithmetik lediglich Matrix-Vektor-Multiplikationen und Skalarprodukte, und lässt sich wiefolgt kompakt schreiben:

$$\begin{aligned}
\alpha_i &= \frac{\boldsymbol{r_i}^\top \boldsymbol{r_i}}{\boldsymbol{d_i}^\top \boldsymbol{A} \boldsymbol{d_i}} \\
\boldsymbol{x_{i+1}} &= \boldsymbol{x_i} + \alpha_i \boldsymbol{d_i} \\
\boldsymbol{r_{i+1}} &= \boldsymbol{r_i} - \alpha_i \boldsymbol{A} \boldsymbol{d_i} \\
\beta_i &= \frac{\boldsymbol{r_{i+1}}^\top \boldsymbol{r_{i+1}}}{\boldsymbol{r_i}^\top \boldsymbol{r_i}} \\
\boldsymbol{d_{i+1}} &= \boldsymbol{r_{i+1}} + \beta_{i+1} \boldsymbol{d_i}
\end{aligned}$$

Implementieren Sie das CG-Verfahren in der Funktion `cg`, welche die Systemmatrix $\boldsymbol{A}$ im CRS-Format, sowie die rechte Seite $\boldsymbol{b}$ und den Startpunkt der Iteration $\boldsymbol{x}_0$ als eindimensionale NumPy-Arrays erhält. Darüberhinaus erhält Ihre Funktion zwei Parameter `epsilon` und `max_iter`, welche die Anzahl der ausgeführten Iterationen steuern: Die CG-Iteration soll so lange ausgeführt werden, bis entweder eine maximale Anzahl von `max_iter` Iterationen erreicht wird, oder für die euklidische Norm $\|\boldsymbol{r}_i\|_2$ des Residuums $\boldsymbol{r_i}$ gilt: $\|\boldsymbol{r_i}\|_2 <$ `epsilon`.

Ihre Funktion soll sowohl die berechnete Näherungslösung, als auch die Anzahl der ausgeführten Iterationen zurückgeben.

Hinweis: Sie *können* die Norm mithilfe von `np.linalg.norm` berechnen, aber eventuell gibt es einen effizienteren Weg.

Aufgabe 2 — Poisson-Randwertproblem (8 Punkte) `poisson.py`

Ein Hauptanwendungsgebiet der iterativen Lösungsverfahren aus Aufgabe 1 ist das numerische Lösen von Randwertproblemen partieller Differentialgleichungen. Auf ein solches Randwertproblem, das Poisson-Problem, wollen wir nun unseren CG-Löser anwenden.

Es sei $\Omega = [0,1] \times [0,1] \subset \mathbb{R}^2$ das Einheitsquadrat. Gesucht ist eine Funktion $u : \Omega \to \mathbb{R}$, deren Verhalten wiefolgt bestimmt ist:

(i) Auf dem Rand von Ω ist u durch *Dirichlet-Randbedingungen* gegeben, d.h. für $\boldsymbol{x} \in \partial\Omega$ ist $u(\boldsymbol{x}) = \phi(\boldsymbol{x})$, für eine vorgegebene Funktion $\phi : \partial\Omega \to \mathbb{R}$;

(ii) Im Inneren von Ω soll u die *Poisson-Gleichung*

$$-\Delta u(\boldsymbol{x}) = f(\boldsymbol{x})$$

für ein vorgegebenes $f : \Omega \to \mathbb{R}$ erfüllen. In zwei Dimensionen nimmt diese folgende Gestalt an:

$$-\frac{\partial^2 u}{\partial x^2}(\boldsymbol{x}) - \frac{\partial^2 u}{\partial y^2}(\boldsymbol{x}) = f(\boldsymbol{x})$$

Um dieses Problem numerisch zu lösen, wird der Definitionsbereich Ω durch ein kartesisches Gitter von $N \times N$ Punkten diskretisiert. Die Abstände der Punkte untereinander sind durch die Schrittweite $h := \frac{1}{N-1}$ gegeben. Eine solche Diskretisierung ist in Abb. 1 für $N = 5$ dargestellt. Wir bezeichnen die Gitterpunkte mit $\boldsymbol{x}_{ij}$, und die Funktionswerte von u und f an diesen Punkten mit u_{ij} und f_{ij}. Diese sind definiert als

$$\boldsymbol{x}_{ij} := (i \cdot h, j \cdot h), \quad u_{ij} := u(\boldsymbol{x}_{ij}), \quad f_{ij} = f(\boldsymbol{x}_{ij}) \qquad (i, j \in \{0, 1, \ldots, N-1\})$$

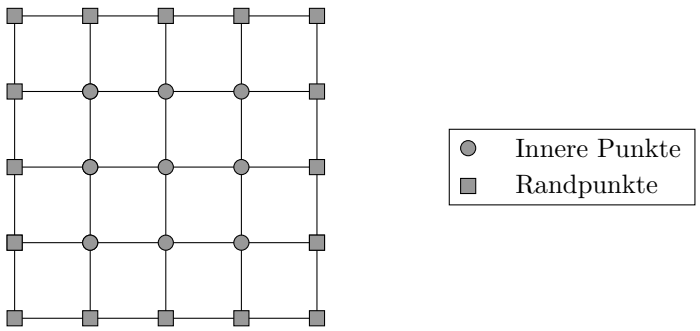

Abbildung 1: Das Diskretisierungsgitter für $N = 5$.

Nun übertragen wir die Forderungen (i) und (ii) auf das diskrete Gitter:

(i) Auf den Randpunkten gelten nach wie vor die Dirichlet-Randbedingungen; d.h. $u_{ij} = \phi(\boldsymbol{x}_{ij})$ für $i \in \{0, N-1\}$ oder $j \in \{0, N-1\}$;

(ii) Im Inneren, d.h. für $(i, j) \in \{1, 2, \ldots, N-2\}^2$, gilt die Finite-Differenzen-Diskretisierung der Poisson-Gleichung:

$$\frac{1}{h^2}(4u_{ij} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1}) = f_{ij}. \qquad (*)$$

Nur die Werte von u an den inneren Punkten sind also unbekannt. Glücklicherweise liefert Forderung (ii) genau eine lineare Gleichung für jeden inneren Punkt! Sammeln wir also die $(N-2)^2$ inneren Punkte in einen Vektor

$$\boldsymbol{u} := (u_{11}, \ldots, u_{1,N-2}, u_{2,1}, \ldots, u_{2,N-2}, \ldots, u_{N-2,1}, \ldots, u_{N-2,N-2})^\top,$$

so können wir das diskrete Randwertproblem als lineares Gleichungssystem, mit dünnbesetzter Systemmatrix $\boldsymbol{A}$ der Größe $(N-2)^2 \times (N-2)^2$, sowie einer rechten Seite $\boldsymbol{b} \in \mathbb{R}^{(N-2)^2}$ darstellen:

$$\boldsymbol{A}\boldsymbol{u} = \boldsymbol{b}$$

Die rechte Seite $\boldsymbol{b}$ entsteht dabei aus der Kombination von f und den Randbedingungen.

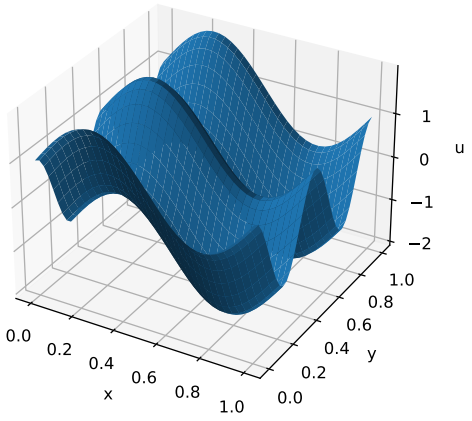

Abbildung 2: Numerische Lösung der Poisson-Gleichung für $f(x,y) = 4\pi^2 \sin(2\pi x) + 16\pi^2 \cos(4\pi y)$ und $\phi(x,y) = \sin(2\pi x) + \cos(4\pi y)$, auf einem Gitter mit $N = 34$.

a) Systemmatrix aufstellen (3 Punkte) Die Gleichungen $(*)$ bestimmen, für jeden inneren Punkt u_{ij}, jeweils eine Zeile von $\boldsymbol{A}$ und den entsprechenden Eintrag von $\boldsymbol{b}$. Überlegen Sie sich, welche Einträge $\boldsymbol{A}$ enthält, und welcher Anteil der Gleichung jeweils zur rechten Seite $\boldsymbol{b}$ gehört (Stichwort: Randpunkte!). Implementieren Sie die Funktion `assemble_poisson_matrix`, welche die Anzahl N von Gitterpunkten in jeder Richtung erhält, und die Systemmatrix $\boldsymbol{A}$ im CRS-Format erzeugt. Nutzen Sie dazu `build_crs_matrix` aus Aufgabe 2a). Achten Sie darauf, dass Ihre Callback-Funktion die Einträge jeder Zeile aufsteigend nach dem Spaltenindex sortiert zurückgibt.

b) Rechte Seite aufstellen (2 Punkte) Implementieren Sie nun die Funktion `assemble_rhs`, welche den Parameter N sowie die Funktionen f und ϕ erhält, und daraus die rechte Seite $\boldsymbol{b}$ des Gleichungssystems als eindimensionales NumPy-Array erzeugt.

c) Poisson-Problem lösen (3 Punkte) Schreiben Sie die Funktion `solve_poisson`, welche das Poisson-Problem für gegebenen Diskretisierungsgrad N, Quellenfeld f und Randbedingung ϕ löst. Stellen Sie dazu das lineare Gleichungssystem mithilfe ihrer Funktionen aus a) und b) auf, und lösen Sie es mit dem CG-Verfahren aus Aufgabe 2c). Ihre Funktion soll schließlich ein $N \times N$-Gitter `u_grid` (NumPy-Array) zurückgeben, welches die Werte u_{ij} der numerischen Lösung u an den Gitterpunkten `u_grid[j,i]` enthält (Achtung: x und y sind vertauscht!).