

Humboldt-Universität zu Berlin
Projektpraktikum I
Gruppe 28

Bericht

**Projektpraktikum I -
Numerische Methoden zum Lösen des
Poisson-Problems**

Mai Nguyen, Ele Tarielashvili

Abgabe: 20. Dezember 2024
Betreut von
Prof. Hella Rabus

Inhaltsverzeichnis

1 Einführung und Motivation	3
2 Theorie	4
2.1 Poisson-Problem	4
2.2 Diskretisierung des Gebietes	4
2.3 Diskretisierung des Laplace-Operators	5
2.4 Definition der Block-Diagonal-Matrix	5
2.5 Aufstellen des Linearen Gleichungssystems	6
2.6 LU-Zerlegung	7
2.6.1 LU-Zerlegung mit Spaltenpivotisierung	7
2.6.2 Lösung eines linearen Gleichungssystems mit LU-Zerlegung	8
2.7 Kondition und Speicherverbrauch	8
2.7.1 Kondition des linearen Gleichungssystems	8
2.7.2 Speicherverbrauch	9
3 Experimente	10
3.1 Speichereffizienz durch Sparse-Matrizen	10
3.2 Approximation der Lösung des Poisson-Problems	11
3.3 Genauigkeit der Lösung	13
4 Auswertung	14
5 Zusammenfassung	14
6 Python-Dokumentation	15
6.1 Bedienungsanleitung und Hauptprogramm	15
6.1.1 Bedienungsanleitung	15
6.1.2 Beispieldurchführung des Hauptprogramms	16
6.2 Schnittstellendokumentation	18
6.2.1 <code>block_matrix_2d.py</code>	19
6.2.2 <code>linear_solvers.py</code>	20
6.2.3 <code>poisson_problem_2d.py</code>	21
6.2.4 <code>experiments_lu.py</code>	22
6.2.5 <code>experiments_lu.main()</code>	23
7 Literaturverzeichnis	24
8 Übersicht verwendeter Hilfsmittel	24
9 Selbständigkeitserklärung	25

1 Einführung und Motivation

Das Poisson-Problem ist eine zentrale Gleichung in der angewandten Mathematik und den Naturwissenschaften. Es beschreibt eine partielle Differentialgleichung, die in der Form $-\Delta u = f$ geschrieben wird, wobei der Laplace-Operator Δ die Summe der zweiten Ableitungen einer unbekannten Funktion u darstellt und f eine gegebene Funktion ist. Das Ziel ist es, u zu bestimmen, sodass die Gleichung in einem gegebenen Gebiet Ω erfüllt wird, wobei zusätzliche Randbedingungen wie festgelegte Werte an den Rändern des Gebiets möglich sind [4].

Das Poisson-Problem modelliert viele physikalische Phänomene, wie beispielsweise die Wärmeverteilung in einem Material oder die elektrischen Potentiale in der Elektrostatik, welche durch folgende Gleichung beschrieben wird: [3].

$$\nabla^2 \phi(\mathbf{r}) = -\frac{\rho(\mathbf{r})}{\varepsilon_0}$$

Hierbei bedeuten:

$\phi(\mathbf{r})$: das elektrische Potential, $\rho(\mathbf{r})$: die Ladungsdichte, ε_0 : die elektrische Konstante.

Obwohl das Poisson-Problem eine klare mathematische Formulierung hat, stellen seine Lösungen in der Praxis eine Herausforderung dar. Viele reale Anwendungen betreffen Gebiete mit komplexen Formen oder varierenden Randbedingungen, die analytische Lösungen unzugänglich machen. Numerische Verfahren erlauben die Lösung des Problems für beliebige Geometrien, Randbedingungen und Quellterme. Die Diskretisierung des Problems mithilfe von Verfahren wie der Finite-Differenzen-Methode führt zu großen linearen Gleichungssystemen, die effizient gelöst werden müssen. Diese Systeme sind in der Regel dünnbesetzt (sparse), was bedeutet, dass viele Einträge null sind. Die Nutzung sparsamer Matrizen ermöglicht es, Speicherbedarf und Rechenzeit zu reduzieren. Verfahren wie die LU-Zerlegung oder iterative Lösungsverfahren helfen dabei, diese großen Gleichungssysteme praktischer zu machen [3] [9].

Die numerische Lösung des Poisson-Problems ist somit essenziell für die Simulation von realen Prozessen und die Entwicklung besserer Methoden. Sie ermöglicht, effizient Näherungslösungen für große Problemstellungen zu berechnen, die hinreichend genau für praktische Anwendungen sind. Jedoch treten auch bei numerischen Verfahren Probleme auf, wie Diskretisierungsfehler und hohe Rechenkosten, auf die wir im Verlauf des Berichts zurückkommen werden.

Ein tiefgehendes Verständnis des Poisson-Problems eröffnet daher zahlreiche Möglichkeiten in der Wissenschaft, Technik und Industrie, um physikalische Phänomene zu analysieren, Materialien zu optimieren und präzise Simulationen durchzuführen.

In diesem Bericht werden wir die numerische Lösung des Poisson-Problems anhand der Beispiefunktion $u : (0, 1)^2 \rightarrow \mathbb{R}$ definiert durch $u(x) = x_1 \sin(\kappa \pi x_1) * x_2 \sin(\kappa \pi x_2)$ für $\kappa \in \mathbb{N}$ untersuchen.

2 Theorie

2.1 Poisson-Problem

Der Laplace-Operator Δ einer Funktion $u \in C^2(\mathbb{R}^2; \mathbb{R})$ ist wie folgt definiert:

$$\Delta u := \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2}. \quad (1)$$

Allgemein beschreibt das Poisson-Problem für ein Gebiet $\Omega \subset \mathbb{R}^2$ mit einem Rand $\partial\Omega$, eine Funktion $f \in C(\Omega; \mathbb{R})$ und eine Funktion $g \in C(\partial\Omega; \mathbb{R})$ die Suche nach einer Funktion u als Lösung der partiellen Differentialgleichung (PDE)

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega, \\ u &= g && \text{auf } \partial\Omega. \end{aligned} \quad (2)$$

In diesem Bericht werden wir den Fall $\Omega = (0, 1)^2$ und $g \equiv 0$ betrachten. Zur numerischen Lösung dieser partiellen Differentialgleichung (PDE) werden wir das Gebiet Ω zunächst diskretisieren. Durch die Anwendung der finiten Differenzen zur Approximation des Laplace-Operators entsteht ein lineares Gleichungssystem $A\hat{u} = b$, wobei der Vektor $\hat{u} \in \mathbb{R}^N$ die Näherungswerte der Funktion u an den N inneren Diskretisierungspunkten approximiert, die in 2.2 definiert sind [1].

2.2 Diskretisierung des Gebietes

Das offene Intervall $(0, 1)$ wird äquidistant in $n \in \mathbb{N}$ Teilintervalle zerlegt.

$$\text{Diskretisierungspunkte } X_1 := \left\{ \frac{j}{n} : 1 \leq j \leq n-1 \right\}.$$

Für die $N := (n-1)^2$ inneren Diskretisierungspunkte für das Einheitsquadrat $\Omega = (0, 1)^2$ gilt also:

$$X = X_1 \times X_1 = \left\{ \left(\frac{j}{n}, \frac{k}{n} \right) : 1 \leq j, k \leq n-1 \right\}. \quad (3)$$

In der Abbildung 1 stellen wir als Beispiel das Gitter mit $n = 5$ in der folgenden Anordnung dar:

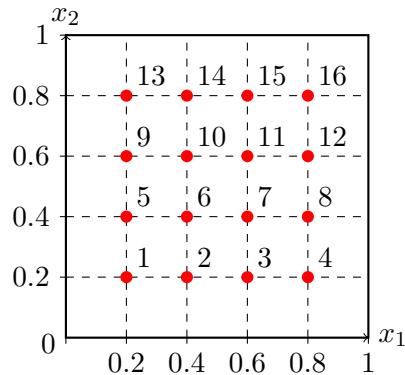


Abbildung 1: Anordnung der diskretisierten Punkte im Einheitsquadrat für $n = 5$

2.3 Diskretisierung des Laplace-Operators

Für die numerische Approximation von u werden wir den Laplace-Operator diskretisieren. Sei $e_\ell \in \mathbb{R}^2$ und $\ell = 1, 2$, dabei bezeichne e_ℓ den ℓ -ten Einheitsvektor. Für $x \in X$ sei die Abbildung $u_{x,\ell} : \mathbb{R} \rightarrow \mathbb{R}$ gegeben durch:

$$u_{x,\ell}(t) := u(x + te_\ell).$$

Seien $x_1, x_2 \in \mathbb{R}$, $x := (x_1, x_2) \in X$. Dann lässt sich die zweite partielle Ableitung von u in Richtung e_ℓ durch die finite Differenz zweiter Ordnung wie folgt approximieren:

$$\frac{\partial^2 u}{\partial x_t^2} = u''_{x,\ell}(0) \approx D_h^{(2)} u_{x,\ell}(0)$$

Daraus erhalten wir die folgende Approximation des Laplace-Operators:

$$\begin{aligned} \Delta_h u(x) &:= D_h^{(2)} u_{x,1}(0) + D_h^{(2)} u_{x,2}(0) \\ &= \frac{u(x_1 + h, x_2) - 2u(x_1, x_2) + u(x_1 - h, x_2)}{h^2} \\ &\quad + \frac{u(x_1, x_2 + h) - 2u(x_1, x_2) + u(x_1, x_2 - h)}{h^2} \\ &= \frac{1}{h^2} (-4u(x_1, x_2) + u(x_1 + h, x_2) + u(x_1 - h, x_2) + u(x_1, x_2 + h) + u(x_1, x_2 - h)) \\ &= \frac{1}{h^2} (-4u(x) + u_{x,1}(h) + u_{x,1}(-h) + u_{x,2}(h) + u_{x,2}(-h)) \end{aligned} \tag{4}$$

Da $u \equiv 0$ auf $\partial\Omega$ gilt, dass wir durch die Werte an den benachbarten Diskretisierungspunkten den diskrete Laplace-Operator von u approximieren können, wobei $n \in \mathbb{N}$ die Anzahl der Teilintervalle aus der Diskretisierung des Gebietes ist. Zudem folgt für $h := \frac{1}{n}$, dass $x \pm he_\ell \in X \cup \partial\Omega$.

Falls also $x_1 \pm h$ bzw. $x_2 \pm h$ in $0, 1$ liegen, werden die entsprechenden Terme in (4) weggelassen. Mögliche weitere Terme, die auf dem Rand liegen sollten, werden ebenfalls weggelassen [1].

2.4 Definition der Block-Diagonal-Matrix

Mithilfe der Diskretisierungen in 2.2 und 2.3 können wir eine Approximation $\hat{u} \in \mathbb{R}^N$ der Lösung u berechnen (siehe (2)). Aus (4) folgt analog, dann für ein fixiertes $n \in \mathbb{N}$ mit $n \geq 2$ die folgende Gleichung:

$$\frac{1}{n^2} f(x_1, x_2) = 4\hat{u}(x) - \hat{u}_{x,1}(h) - \hat{u}_{x,1}(-h) - \hat{u}_{x,2}(h) - \hat{u}_{x,2}(-h) \tag{5}$$

Dadurch erhalten wir ein lineares Gleichungssystem in \hat{u} , die wir als Matrix darstellen können. Dafür definieren wir:

$$\mathcal{C} := \begin{bmatrix} 4 & -1 & 0 & 0 & \cdots & 0 \\ -1 & 4 & -1 & 0 & \cdots & 0 \\ 0 & -1 & 4 & -1 & \cdots & 0 \\ 0 & 0 & -1 & \ddots & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & 4 & -1 \\ 0 & 0 & 0 & \cdots & -1 & 4 \end{bmatrix} \in \mathbb{R}^{(n-1) \times (n-1)} \text{ und} \quad (6)$$

$$A := \begin{bmatrix} \mathcal{C} & -I & 0 & 0 & \cdots & 0 \\ -I & \mathcal{C} & -I & 0 & \cdots & 0 \\ 0 & -I & \mathcal{C} & -I & \cdots & 0 \\ 0 & 0 & -I & \ddots & \ddots & 0 \\ \vdots & \vdots & \vdots & \ddots & \mathcal{C} & -I \\ 0 & 0 & 0 & \cdots & -I & \mathcal{C} \end{bmatrix} \in \mathbb{R}^{N \times N}, \text{ und } \mathcal{I} := \mathcal{I}_{n-1}. \quad (7)$$

Die Matrix \mathcal{C} (siehe (6)) ist mit vielen Nullen besetzt, mit Ausnahme der Hauptdiagonalen sowie der beiden Nebendiagonalen. Sie besitzt $(n-1)^2$ Einträge, wobei $3n-5$ davon Nicht-Null-Einträge sind. Dabei besitzt die Einheitsmatrix $\mathcal{I} \in \mathbb{R}^{(n-1) \times (n-1)}$ nur Einträge auf der Diagonalen, also $n-1$ Nicht-Null-Einträge. Für die Matrix A (siehe (7)) gilt also, dass sie $(n-1)^4$ Einträge hat, wobei $5n^2 - 14n + 9$ davon Nicht-Null-Einträge sind.

Allgemein gilt, dass das Abspeichern von $(N \times N)$ -Matrizen einen Speicherbedarf von $\mathcal{O}(N^2)$ erfordert, falls jeder einzelne Eintrag gespeichert wird. Da aber sehr viele Einträge der N^2 Einträge in A gleich Null sind, wird die Matrix als *sparse* (spärliche besetzt) betrachtet. Aus diesem Grund empfiehlt sich ein spezielles Speicherformat, die CSR-Matrix (Compressed Sparse Row). Diese wird im Python-Modul `scipy` als `scipy.sparse.csr_matrix` implementiert. Dabei werden nur die Werte sowie die Zeilen- und Spaltenindizes der Einträge gespeichert, die von Null verschieden sind. Dadurch wird vermieden, dass unnötig viele Nullwerte gespeichert werden müssen [5] [6].

2.5 Aufstellen des Linearen Gleichungssystems

\hat{u} repräsentiert die Approximation der Lösung der diskretisierten PDE

$$-\Delta_h u(x) = f(x) \quad \forall x \in X, \quad u = 0 \text{ auf } \partial\Omega$$

an den N Diskretisierungspunkten. Es ergeben sich N lineare Gleichungen. Um diese als lineares Gleichungssystem zu schreiben, muss jedem Diskretisierungspunkt $x \in X$ eindeutig eine Gleichungsnummer $m \in \{1, \dots, N\}$ zugeordnet werden[1].

Dazu seien die Diskretisierungspunkte $x = (x_1, x_2) \in X$ und $y = (y_1, y_2) \in X$ durch die Relation

$$x <_X y \iff x_1 n + x_2 n^2 < y_1 n + y_2 n^2$$

geordnet (vgl. Abbildung 1 der Anordnung der inneren Diskretisierungspunkte im Einheitsquadrat für $n = 5$).

Dann werden die Gleichungen aufsteigend nach $<_X$ sortiert und als resultierende Matrix erhalten wir $h^2 A$. Dann ist also der m-te Eintrag der gesuchten Lösung \hat{u} des linearen Gleichungssystems eine Approximation der exakten Lösung u am m-ten Diskretisierungspunkt [1].

Die Gleichungen (5) lassen sich dann durch das folgende Gleichungssystem zusammenfassen für $v_1, \dots, v_N \in X$:

$$A\hat{u} = b := \frac{1}{n^2} \begin{bmatrix} f(v_1) \\ \vdots \\ f(v_N) \end{bmatrix}. \quad (8)$$

Um das Gleichungssystem (8) zu lösen werden wir die *LU*-Zerlegung der Matrix nutzen, die im folgenden Abschnitt erläutert wird.

2.6 LU-Zerlegung

Die *LU*-Zerlegung oder auch *LR*-Zerlegung genannt, ist die Zerlegung einer quadratischen Matrix $A \in \mathbb{R}^{n \times n}$ in ein Produkt:

$$A = P \cdot L \cdot U$$

Dabei haben die anderen Matrizen folgende Eigenschaften:

- P Permutationsmatrix, entsteht durch Vertauschen von Spalten aus der Einheitsmatrix
- L untere Dreiecksmatrix mit Einsen auf der Hauptdiagonalen
- U obere Dreiecksmatrix.

2.6.1 LU-Zerlegung mit Spaltenpivotisierung

Im Folgenden wird der Prozess der LU-Zerlegung mit der Spaltenpivotisierung beschrieben :

1. **Initialisierung:** Setze $k = 1$ und $A^{(1)} = A$.
2. **Pivotwahl:** Bestimme ein $p \in \{k, \dots, n\}$, so dass

$$|a_{pj}^{(k)}| \geq |a_{ij}^{(k)}| \quad \text{für alle } i \in \{k, \dots, n\}.$$

Die Zeile p wird als Pivotzeile gewählt.

3. **Zeilenvertauschung:** Vertausche die Zeilen p und k . Dies bedeutet, dass:

$$a_{ij}^{(k)} \rightarrow \tilde{a}_{ij}^{(k)} = \begin{cases} a_{pj}^{(k)} & \text{für } i = k, \\ a_{kj}^{(k)} & \text{für } i = p, \\ a_{ij}^{(k)} & \text{sonst.} \end{cases}$$

4. **Eliminationsschritt:** Führe den Eliminationsschritt durch und berechne $\tilde{A}(k)$ als $A^{(k+1)}$. Erhöhe den Index k um 1, also setze $k = k + 1$.
5. **Abbruchbedingung:** Wenn $k = n$, stoppe den Algorithmus. Andernfalls gehe zurück zu Schritt 2.

Andrea Walther, 3. Vorlesung Numerische Algebra, LR-Zerlegung, Algorithmus 2.6, Wintersemester 2024/25, Humboldt-Universität zu Berlin

2.6.2 Lösung eines linearen Gleichungssystems mit LU-Zerlegung

Zur Lösung von $A\vec{x} = \vec{b}$ nimmt man folgende Schritte vor:

- 1 Bestimme die LU-Zerlegung von A : $A = PLU$.
- 2 Löse $P\vec{z} = \vec{b}$ durch $\vec{z} = P^\top \vec{b}$.
- 3 Löse $L\vec{y} = \vec{z}$ rekursiv, beginnend mit y_1 .
- 4 Löse $U\vec{x} = \vec{y}$ rekursiv, beginnend mit x_n .

Bei der vereinfachten LU-Zerlegung ist $P = I$, Schritt 2 fällt weg und es gilt $\vec{z} = \vec{b}$.

*Furlan, Peter: Zusätze zum Gelben Rechenbuch LU-Zerlegung.
<https://www.das-gelbe-rechenbuch.de/download/Lu.pdf>, 2024*

Mithilfe der LU-Zerlegung für die Matrix \mathcal{A} und dessen Algorithmus für die Lösung eines linearen Gleichungssystems können wir dann das Gleichungssystem aus (8) numerisch lösen.

2.7 Kondition und Speicherverbrauch

2.7.1 Kondition des linearen Gleichungssystems

Die Kondition eines linearen Gleichungssystems ist ein wichtiger Aspekt bei der numerischen Lösung, da sie angibt, wie empfindlich die Lösung gegenüber Fehlern in den Eingangsdaten oder der numerischen Berechnung ist. Ein schlecht konditioniertes System führt oft zu großen Fehlern bei der Berechnung der Lösung, selbst wenn die Eingabedaten nur geringe Fehler aufweisen [8].

Das lineare Gleichungssystem, das aus der Diskretisierung des Poisson-Problems resultiert, hat die Form $A\hat{u} = b$, wobei A die Matrix des diskretisierten Laplace-Operators darstellt. In diesem Fall ist die Matrix A eine Blockdiagonalmatrix, die durch die Definition der Matrizen \mathcal{C} und \mathcal{A} beschrieben wird (siehe Abschnitt 2.3). Diese Matrix ist spärlich (sparse), was bedeutet, dass viele ihrer Elemente null sind, aber die wichtigen Werte befinden sich auf der Haupt- und Nebendiagonalen.

Die Kondition der Matrix A hängt stark von der Gittergröße n ab. Bei zunehmendem n (d.h., einer feineren Gitterauflösung) wächst die Kondition der Matrix, was zu einer

schlechteren Konditionierung des Systems führt. Dies kann dazu führen, dass numerische Methoden zur Lösung des Systems, wie zum Beispiel der direkte oder iterative Lösungsansatz, ungenau werden oder mit einer höheren Anzahl von Iterationen konvergieren. Besonders bei der Verwendung von iterativen Lösungsverfahren müssen wir darauf achten, dass das System ausreichend gut konditioniert ist, um eine effiziente und präzise Lösung zu gewährleisten.

Die Kondition einer invertierbaren Matrix $A \in \mathbb{R}^{n \times n}$ ist dann wie folgt definiert:

$$\kappa_{\|\cdot\|}(A) = \|A\| \|A^{-1}\|,$$

wobei $\|\cdot\|$ eine submultiplikative Matrixnorm ist [8].

2.7.2 Speicherverbrauch

Der Speicherverbrauch spielt bei der numerischen Lösung des Poisson-Problems eine bedeutende Rolle, insbesondere bei großen Gittergrößen n . Die Matrix A , die das lineare Gleichungssystem beschreibt, hat eine Größe von $N \times N$, wobei $N = (n - 1)^2$ die Anzahl der inneren Gitterpunkte ist. Da die Matrix A jedoch spärlich ist, können wir wie in 2.4 nur die nicht nullen Werte gespeichert werden, inklusive der Stelle in der sie gespeichert werden, was zu einer erheblichen Reduzierung des Speicherbedarfs führt.

Der Speicherbedarf für eine spärliche Matrix wird durch die Anzahl der Nicht-Null-Werte bestimmt. Wenn jede nicht-null Zahl und deren Indizes gespeichert werden, beträgt der Speicherverbrauch für die Matrix A der Ordnung $N \times N$ nur $\mathcal{O}(N)$, da die meisten Einträge null sind. Dies steht im Gegensatz zu einer dichten Matrix, bei der der Speicherbedarf $\mathcal{O}(N^2)$ wäre.

Die Implementierung der Matrix A in einem sparsamen Format, wie der CSR (Compressed Sparse Row)-Matrix, reduziert den Speicherverbrauch weiter. Bei diesem Format werden nur die tatsächlichen Nicht-Null-Werte sowie deren Zeilen- und Spaltenindizes gespeichert. In Python wird dies häufig mit der Klasse `scipy.sparse.csr_matrix` umgesetzt. Durch diese effiziente Speicherung wird der Speicherverbrauch signifikant reduziert, insbesondere bei großen Problemgrößen, und ermöglicht die Arbeit mit größeren Gitterauflösungen ohne unnötigen Speicheraufwand [5] [6].

Insgesamt gilt, dass die Blockstruktur der Matrix A , die durch die Matrizen \mathcal{C} und \mathcal{I} definiert ist, hat Auswirkungen sowohl auf die Kondition als auch auf den Speicherbedarf. Eine strukturierte Matrix, die spärlich ist, kann effizient gespeichert und bearbeitet werden, allerdings kann eine ungünstige Verteilung der Nicht-Null-Werte die Kondition verschlechtern. Der Speicherbedarf und die Kondition hängen also oft zusammen: Während der Speicherbedarf durch die sparsere Struktur verringert wird, können durch die Matrixstruktur auch bestimmte Eigenschaften wie die Kondition beeinflusst werden, die sich wiederum auf die numerische Stabilität und die Fehleranfälligkeit der Berechnungen auswirken.

3 Experimente

Es wurden Experimente durchgeführt, um die Effizienz der LU-Zerlegung als Lösungsansatz des Poisson-Problems zu untersuchen. Ziel war es dabei, Aussagen über die Präzision und Konvergenz der Methode zu treffen. Hierfür wurden die Auswirkungen verschiedener Faktoren wie der Schrittweite h , der Matrixkondition und des Speicherbedarfs betrachtet. Für die Untersuchung betrachten wir für $\kappa \in \mathbb{N}$ die folgende Funktion $u : \Omega \rightarrow \mathbb{R}$

$$u(x) = x_1 \sin(k\pi x_1) * x_2 \sin(k\pi x_2). \quad (9)$$

Die Funktion u (siehe (9)) ist dabei die Lösung des Poisson-Problems für die Funktion $f : \Omega \rightarrow \mathbb{R}$, die gegeben ist durch

$$\begin{aligned} f(x_1, x_2) = & 2\kappa\pi(\kappa\pi x_1 \sin(\kappa\pi x_1)x_2 \sin(\kappa\pi x_2) \\ & - x_2 \sin(\kappa\pi x_2) \cos(\kappa\pi x_1) \\ & - x_1 \sin(\kappa\pi x_1) \cos(\kappa\pi x_2)). \end{aligned}$$

Für diese Untersuchung nutzen wir also f für das numerische Berechnen des Poisson-Problems und überprüfen, wie die Lösung sich der Funktion u annähert und welche Problemstellungen entstehen.

3.1 Speichereffizienz durch Sparse-Matrizen

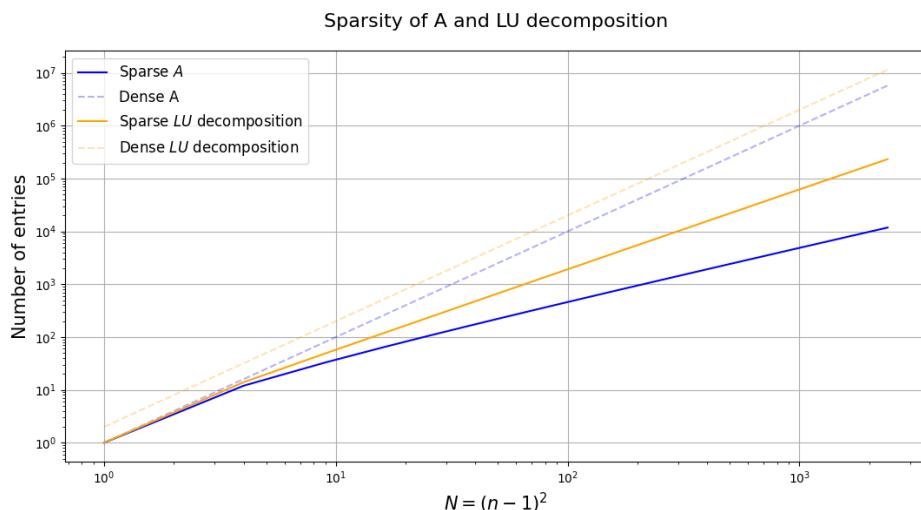


Abbildung 2: Sparsity der Matrix A und ihrer LU-Zerlegung

Ein zentraler Aspekt der Effizienz der LU-Zerlegung ist die Speicherung der entstehenden Matrizen. Abbildung 2 zeigt das Wachstum der Einträge der Block-Diagonal-Matrix, die während der LU-Zerlegung entsteht. Die Anzahl der Einträge der LU-Zerlegung steigt deutlich stärker an im Vergleich zu Block-Diagonal-Matrix A . Grund dafür ist, dass bei

der LU-Zerlegung zusätzliche Füllwerte entstehen (der sogenannte Fill-in Effekt), was bedeutet, dass die Matrizen nun mehr Nicht-Null-Einträge enthalten. Dies führt dazu, dass die L- und U-Matrizen eine höhere Dichte aufweisen als die Ausgangsmatrix A. Trotzdem ist der Speicherbedarf bei der Verwendung von Sparse-Matrizen moderat. Sparse-Matrizen speichern nur die Einträge ungleich Null, was den Speicherbedarf erheblich verringert. Selbst bei sehr großen Matrizen, die bei einer feinen Schrittweite entstehen, wird die Skalierung des Speicherbedarfs handhabbar.

3.2 Approximation der Lösung des Poisson-Problems

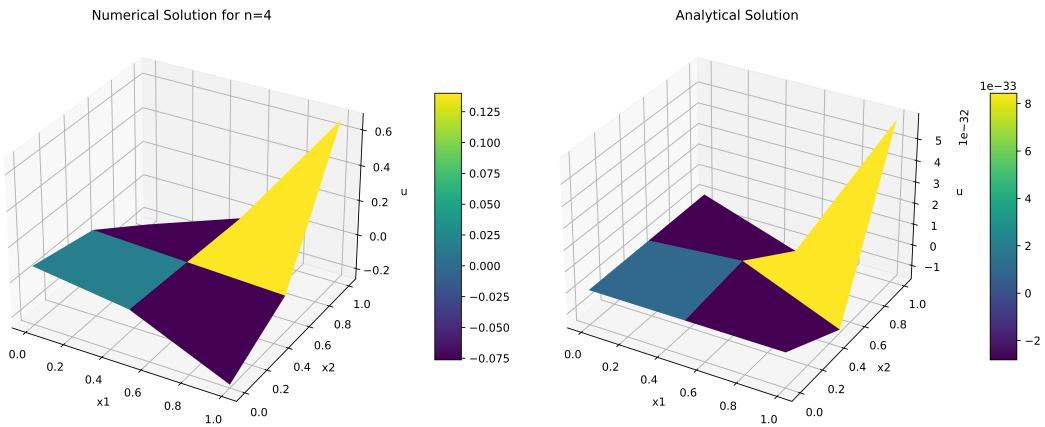


Abbildung 3: Approximation der Lösung und die analytische Lösung mit $n = 4$

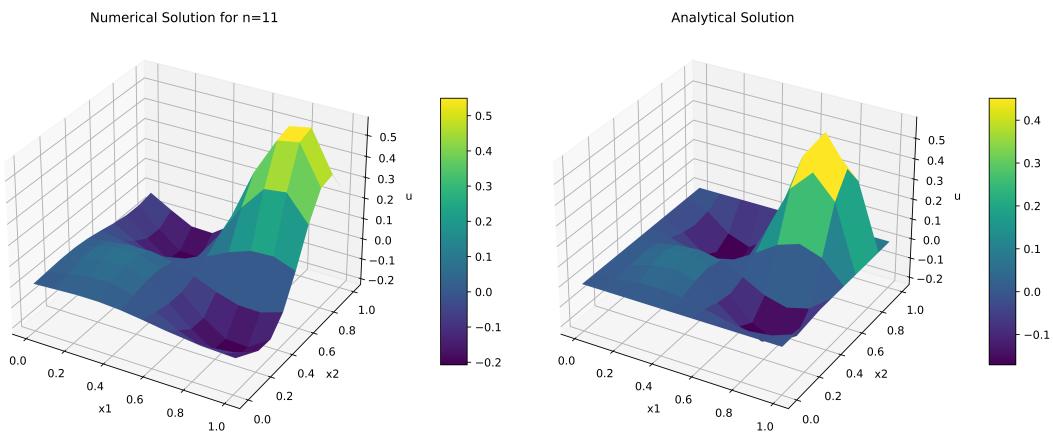


Abbildung 4: Approximation der Lösung und die analytische Lösung mit $n = 11$

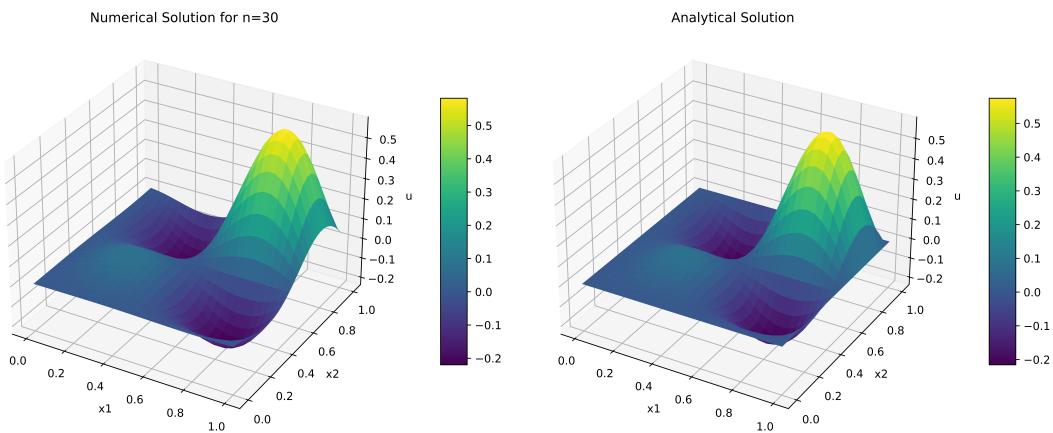


Abbildung 5: Approximation der Lösung und die analytische Lösung mit $n = 30$

Abbildung 3, 4 und 5 visualisieren die approximierten Lösungen des Poisson-Problems mit verschiedenen Schrittweiten im Vergleich zur analytischen Lösung. Beide Lösungen basieren immer auf einem regelmäßigen Gitter mit einer gegebenen Gitterweite. Die Abbildung 3 stellt die numerisch berechneten Ergebnisse mit der Gitterweite 0.25 dar. Es führt zu einer groben Approximation, denn die Lösung zeigt merkliche Abweichungen von der analytischen Lösung, was insbesondere auf die geringe Gitterauflösung $n = 4$ zurückzuführen ist. Abbildung 4 illustriert denselben Vergleich für eine höhere Auflösung,

und zwar $n = 11$. Auffällig ist die deutlich höhere Genauigkeit der Approximation. Trotz der Ähnlichkeit der numerischen Lösung mit der analytischen Referenz, sind Diskretisierungsfehler sichtbar, besonders an den Rändern der Lösung, was erneut auf die begrenzte Gitterauflösung zurückzuführen ist. Jedoch nehmen diese Abweichungen mit einer feineren Schrittweite ab, wie man in Abbildung 5 beobachten kann. Der Unterschied der beiden Lösungen ist gering, was auf eine effektive Approximation durch die Finite-Differenzen-Methode hindeutet. Die Approximationsgenauigkeit für eine zunehmende Schrittweite kann in Abbildung 6 genauer beobachtet werden.

3.3 Genauigkeit der Lösung

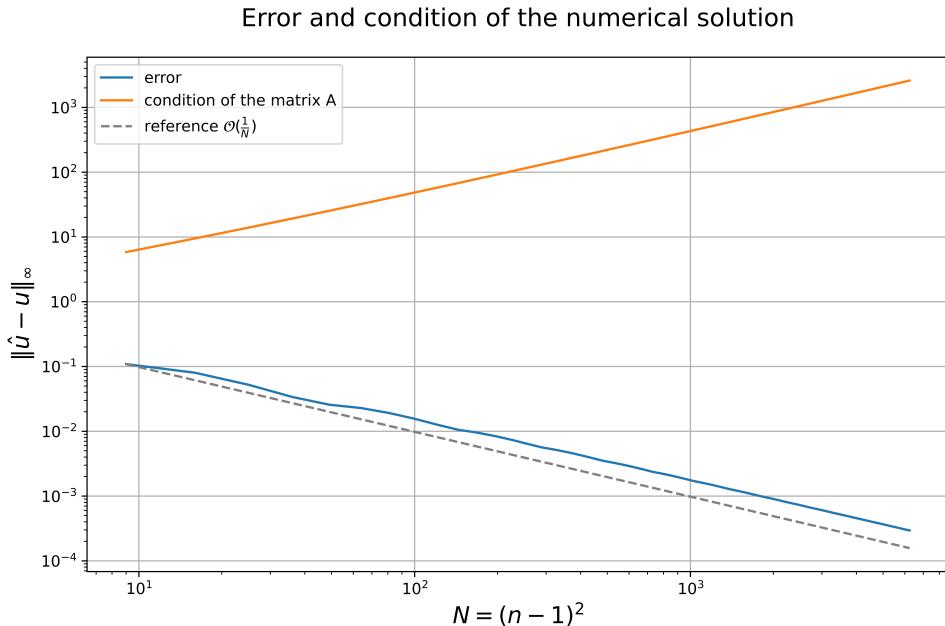


Abbildung 6: Konvergenzverhalten der Approximationsfehler und Kondition von A

In Abbildung 6 wird die Fehleranalyse für verschiedene Gitterauflösungen durchgeführt. Der logarithmische Plot zeigt den Fehler in der $\|\cdot\|_\infty$ (blaue Kurve) und die Kondition des Matrixsystems A (orange Kurve) in Abhängigkeit von der Anzahl der Gitterpunkte $N = (n - 1)^2$.

Die Fehlerkurve fällt proportional zu $\mathcal{O}(N^{-1})$, was eine quadratische Konvergenzordnung bestätigt. Diese Konvergenz lässt sich auf die zentrale finite-Differenzen-Methode zurückführen, dessen Fehlerkonvergenz der Ordnung $\mathcal{O}(h^2)$ ist [1]. Da $h = \frac{1}{n}$, entspricht die Fehlerkonvergenz der approximierten Lösung der Poisson-Gleichung $\hat{u} \in \mathcal{O}\left(\frac{1}{n^2}\right)$. Dies stimmt mit der theoretischen Erwartung überein, da der Fehler durch die Gitterfeinheit h und dadurch auch durch die Anzahl der Gitterpunkte $N = (n - 1)^2$ bestimmt wird.

Zudem lässt sich auch das Verhalten der Konditionszahl der Block-Diagonal-Matrix A beobachten. Diese wächst mit zunehmender Gitterauflösung quadratisch, was auf eine numerische Instabilität bei sehr feiner Diskretisierung weist.

4 Auswertung

Nach der Beobachtung der Experimente, lässt sich einiges der eingeführten Theorie bestätigen. Die zusätzlichen Einträge, die bei der LU-Zerlegung dazukommen, machen die Matrizen dichter und erhöhen somit den Speicherbedarf mit steigendem n . Obwohl dies zunächst gegen die Effizienz der LU-Zerlegung spricht, reduziert die Sparse-Matrix-Darstellung diesen Bedarf erheblich, da nur die Nicht-Null-Einträge gespeichert werden. Dadurch bleibt der Speicherbedarf weiterhin moderat und ermöglicht die Lösung des Poisson-Problems mit einer hohen Gitterauflösung. Mit einer feineren Diskretisierung steigt auch die Approximationsgenauigkeit der finiten Differenzen mit $\mathcal{O}(h^2)$, da die Fehler des Differenzgitters kleiner werden. Obwohl die Lösung der LU-Zerlegung mit kleiner werdender Schrittweite h an Präzision gewinnt, wächst die Kondition der Matrix A quadratisch. Eine schlechte Kondition führt dazu, dass Rundungsfehler in der Lösung verstärkt werden, was die numerische Genauigkeit beeinflusst. Dies limitiert letztlich die Genauigkeit der Lösung, insbesondere bei sehr kleinen Schrittweiten. Der Fehler erreicht eine Grenze, da der Diskretisierungsfehler kleiner wird, aber der Rundungsfehler steigt. Der genaue Punkt bzw. die genaue Schrittweite, bei der es dazu kommt, kann jedoch nicht genau genannt werden. Bereits ab $n = 90$ steigt die Laufzeit des Plots für die Darstellung des Konvergenzverhaltens drastisch an.

5 Zusammenfassung

Letztlich kann gesagt werden die LU-Zerlegung bietet eine hohe Effizienz zur Lösung des Poisson-Problems, insbesondere durch ihre Kombination mit Sparse-Matrizen. Ihre Leistungsfähigkeit und Approximationsgenauigkeit hängt jedoch von mehreren Faktoren ab, die sowohl die Rechenzeit, als auch den Speicherbedarf und die numerische Genauigkeit beeinflussen. Obwohl die Methode genaue Approximationen mit kleiner werdenden Gitterweiten liefert, ist ein kritischer limitierender Faktor jedoch die Kondition der Matrix A, die sich mit zunehmender Gitterfeinheit verschlechtert. Eine hohe Kondition führt dazu, dass Rundungsfehler verstärkt werden, was die Präzision der numerischen Lösung beeinträchtigt. Ab einer bestimmten Gitterweite wird die Lösung durch die Fortpflanzung der entstandenen Rundungsfehler dominiert, unabhängig von der Rechenzeit oder der verfügbaren Speicherkapazität. Dies begrenzt die Effizienz der LU-Zerlegung für große n . Alternative iterative Verfahren könnten hier eine bessere Genauigkeit und Performance bieten.

6 Python-Dokumentation

6.1 Bedienungsanleitung und Hauptprogramm

Das Pythonmodul zum numerischen Lösen des Poisson-Problems besteht aus vier Pythonprogrammen `block_matrix_2d.py`, `experiments_lu.py`, `linear_solvers.py` und `poisson_problem_2d.py`. Das Hauptmodul `experiments_lu.py` ist ein Experimentierskript, welches eine umfassende Möglichkeit bietet, die Lösung der Poisson-Gleichung zu berechnen, zu visualisieren und zu analysieren. Es kombiniert numerische Verfahren (LU-Zerlegung), Fehleranalyse und grafische Darstellung, um die Eigenschaften der Lösung und die Auswirkungen der Diskretisierung zu untersuchen. Das Experimentierskript enthält definierte Funktionen aus `block_matrix_2d.py`, `linear_solvers.py` und `poisson_problem_2d.py`.

In `block_matrix_2d.py` ist die Klasse `BlockMatrix` implementiert, die Funktionen zur Konstruktion und Analyse von dünn besetzten Matrizen (sparse matrices) bietet, die bei der Lösung des Poisson-Problems auf einem Einheitsquadrat mittels finiter Differenzen verwendet werden. Es unterstützt die Erstellung der Matrix, Analyse ihrer Struktur, LU-Zerlegung sowie Visualisierungen der Besetzungsmuster.

Im Programm `linear_solvers.py` ist die Funktion `solve_lu` zur Lösung eines linearen Gleichungssystems $Ax = b$ definiert, basierend auf der LU-Zerlegung der Matrix A . Es verwendet Vorwärts- und Rückwärtssubstitution zur effizienten Berechnung der Lösung.

Das Programm `poisson_problem_2d.py` implementiert numerische Methoden zur Lösung des Poisson-Problems mithilfe der Finite-Differenzen-Methode. Es enthält Funktionen zur Erstellung des rechten Seitenvektors, zur Abbildung zwischen Gleichungsnummern und Gitterpunkten, zur Berechnung numerischer Fehler und zur Visualisierung des Fehlerverhaltens.

Das Hauptprogramm wird nur gestartet, wenn `experiments_lu.py` mittels `python3 experiments_lu.py` gestartet wird.

6.1.1 Bedienungsanleitung

Starten Sie zuerst das Programm `experiments_lu.py`. Es wird die `main()`-Funktion gestartet. Als Funktionen sind $u(x) = x_1 \sin(k\pi x_1) * x_2 \sin(k\pi x_2)$, die Lösung des Poisson-Problems und $f(x_1, x_2) = 2\kappa\pi(\kappa\pi x_1 \sin(\kappa\pi x_1)x_2 \sin(\kappa\pi x_2) - x_2 \sin(\kappa\pi x_2) \cos(\kappa\pi x_1) - x_1 \sin(\kappa\pi x_1) \cos(\kappa\pi x_2))$ gegeben. Es werden drei 3D-Plots wiedergegeben, die die numerische und analytische Lösung des Poisson-Problems für die Gittergrößen $n = 4, 11$ und 30 graphisch darstellen. Diese werden jeweils unter den Namen `3dplot4.png`, `3dplot11.png` und `3dplot30.png` gespeichert. Es wird ebenfalls eine Grafik geplottet, die die Kondition der zu betrachtenden Matrix und die Approximationenfehler des Poissons-Problems für bis zu $n = 10^4$ angibt. Die Plots werden im gleichen Ordner wie `experiments_lu.py` gespeichert. Zu berücksichtigen ist, dass keine PNG-Datei mit dem gleichen Dateinamen im Ordner des Programms vorliegen sollte, da diese sonst überschrieben wird.

6.1.2 Beispielausführung des Hauptprogramms

Zur Veranschaulichung des Programms führen wir eine Beispielausführung durch. Beim Starten des Programms wird dem Nutzer folgendes wiedergegeben:

```
Analyzing for grid size n=4:  
Error: 0.10888318155034138  
Sparsity of the matrix A: 0.4074  
Time taken: 6.604194641113281e-05 seconds
```

Der Plot 3dplot4.png wird angezeigt und im selben Ordner wie das Programm gespeichert.

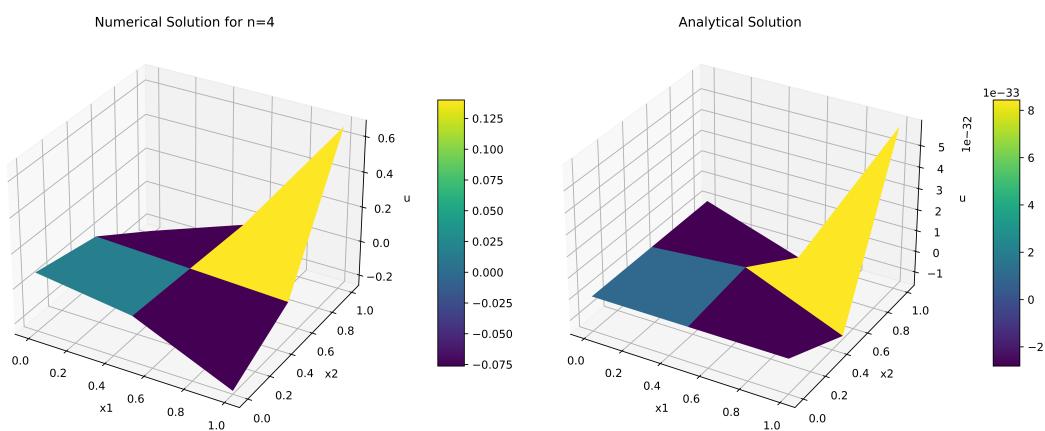


Abbildung 7: 3D-Plot für $n = 4$ der numerischen und analytischen Lösung

```
Analyzing for grid size n=11:  
Error: 0.015592509110129993  
Sparsity of the matrix A: 0.0460  
Time taken: 0.004178047180175781 seconds
```

Der Plot 3dplot11.png wird angezeigt und im selben Ordner wie das Programm gespeichert.

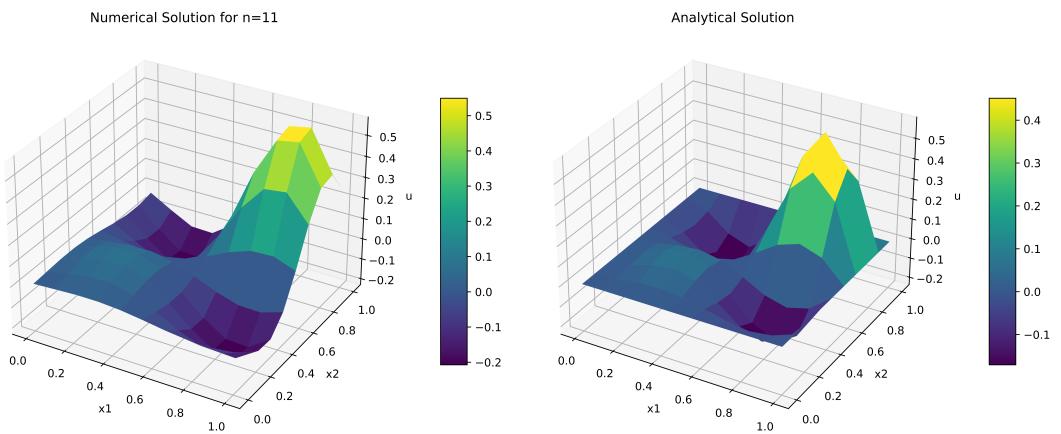


Abbildung 8: 3D-Plot für $n = 11$ der numerischen und analytischen Lösung

Analyzing for grid size n=30:

Error: 0.002094948095691329

Sparsity of the matrix A: 0.0058

Time taken: 0.005517005920410156 seconds

Der Plot 3dplot30.png wird angezeigt und im selben Ordner wie das Programm gespeichert.

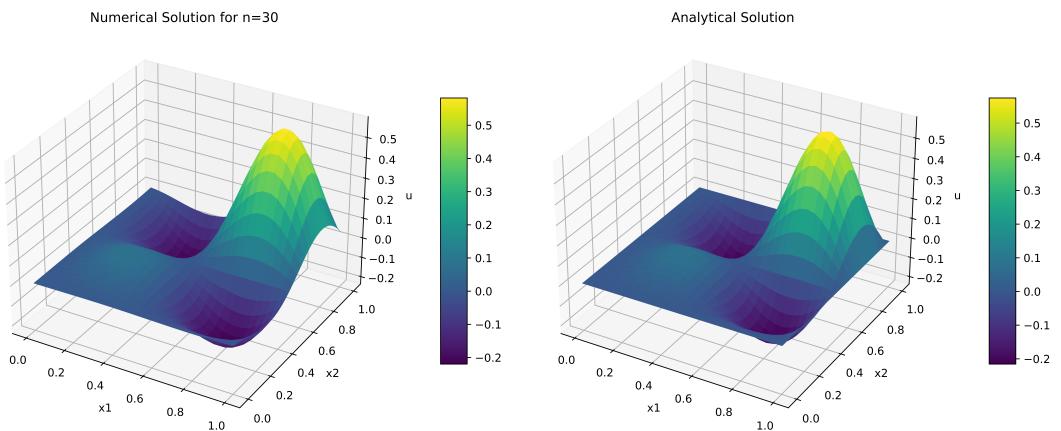


Abbildung 9: 3D-Plot für $n = 30$ der numerischen und analytischen Lösung

Dem Nutzer wird der Fehler der numerischen Lösung des Poisson-Problems wiedergegeben für $n = 50$.

```
For n=50 the error of the numerical solution of the
Poisson problem is 0.0007533860028885453
```

Schließlich wird der Fehlerplot des Poisson-Problems geplottet:

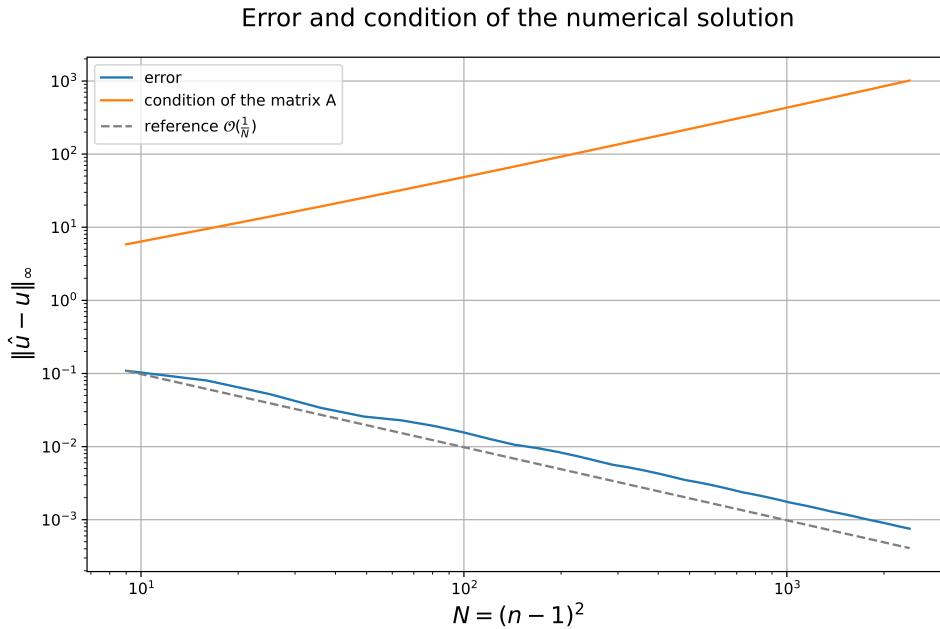


Abbildung 10: Fehlerplot des Poisson-Problems und die Kondition der Matrix

6.2 Schnittstellendokumentation

Das Hauptmodul enthält Funktionen aus den Programmen `block_matrix_2d.py`, `linear_solvers.py` und `poisson_problem_2d.py`. In `block_matrix_2d.py` ist die Klasse `BlockMatrix` definiert, die für die Konstruktion und Analyse von Sparse-Matrizen dient. In `linear_solvers.py` ist eine Funktion für die Berechnung des linearen Gleichungssystems $Ax = b$ definiert. In `poisson_problem_2d.py` sind Funktionen zur Berechnung des Poisson-Problems mithilfe von finiten Differenzen implementiert.

Bemerkung 1: Die Reihenfolge der Eingabeparameter für die Funktionen wurden unter **Parameters** von oben nach unten aufgelistet.

Bemerkung 2: Da in jeder Pythondatei eine `main()` Funktion vorhanden ist, wird vor jeder `main()` Funktion der jeweilige Dateiname dazugeschrieben, um Verwechslungen zu vermeiden.

6.2.1 block_matrix_2d.py

Das Programm enthält die Klasse `BlockMatrix` in der weitere Funktionen definiert sind und einer `main()` Funktion.

Klasse `BlockMatrix`

Repräsentiert Blockmatrizen, die durch finite Differenzen für den Laplace-Operator entstehen.

Parameters:

`n` (*int*) Anzahl der Intervalle in jeder Dimension. Muss mindestens 2 sein.

Attribute:

`n` (*int*) Anzahl der Intervalle in jeder Dimension

`n_inner` (*int*) Anzahl der inneren Punkte, $n - 1$

`capn` (*int*) Gesamtanzahl der Unbekannten $(n - 1)^2$

Methoden:

`get_sparse()`

Erstellt und gibt die Blockmatrix als dünn besetzte Matrix zurück.

Parameters: *None*

Returns:

(*scipy.sparse.csr_matrix*) Blockmatrix als Sparse-Matrix

`eval_sparsity()`

Berechnet die absolute und relative Anzahl von Nicht-Null-Elementen der Matrix.

Parameters: *None*

Returns:

(*int*) Anzahl der Nicht-Null-Einträge

(*float*) Verhältnis der Nicht-Null-Elemente zur Gesamtanzahl der Matrixeinträge als relative Zahl

`get_lu()`

Gibt die LU-Zerlegung der Matrix in der Form $A = PLU$ zurück.

Parameters: *None*

Returns:

(*numpy.ndarray*) `p` als Permutationsmatrix der LU-Zerlegung

(*numpy.ndarray*) `l` als untere Dreiecksmatrix mit Einsen auf der Hauptdiagonalen

(*numpy.ndarray*) `u` als obere Dreiecksmatrix der LU-Zerlegung

`eval_sparsity_lu()`

Berechnet die absolute und relative Anzahl der Nicht-Null-Elemente der LU-Zerlegung.

Parameters: *None*

Returns:

(int) Anzahl der Nicht-Null-Elemente.

(float) Verhältnis der Nicht-Null-Elemente der LU-Zerlegung zur Matrix A .

`plot_sparsity_capn()`

Plottet die Anzahl der Nicht-Null-Elemente der Matrix A und ihrer LU-Zerlegung für verschiedene Werte von n .

Paramters:

max_n *(int)* Maximaler Wert von n .

Returns: *None*, diese Funktion gibt keine Werte zurück, sondern erzeugt und zeigt den Plot an bzw. speichert ihn in einer PNG-Datei ein.

`get_cond()`

Berechnet die Konditionszahl der Matrix in Bezug auf die unendliche Norm.

Parameters: *None*

Returns:

(float) Konditionszahl

`block_matrix_2d.main()`

Demonstriert die Nutzung der BlockMatrix-Klasse und ihrer Methoden.

Parameters: *None*

Returns: *None*, denn diese Funktion gibt keine Werte zurück, sondern erstellt eine Instanz der Klasse `BlockMatrix`, zeigt die dünn besetzte Darstellung der Matrix A und analysiert die Sparsamkeit der Matrix. Zudem erstellt sie ein Sparsamkeitsdiagramm für verschiedene Werte von n .

6.2.2 `linear_solvers.py`

Dieses Modul besteht aus einer Funktion.

`solve_lu()`

Löst das lineare Gleichungssystem $Ax = b$, wenn die Matrix A bereits in der Form $A = PLU$ zerlegt wurde.

Parameters:

p (`numpy.ndarray`) : Permutationsmatrix der LU-Zerlegung

l (`numpy.ndarray`) : Untere Dreiecksmatrix mit Einsen auf der Diagonale der LU-Zerlegung

u (`numpy.ndarray`) : Obere Dreiecksmatrix der LU-Zerlegung

b (*numpy.ndarray*) : Vektor der rechten Seite des Gleichungssystems

Returns:

(*numpy.ndarray*) x als Lösung des linearen Gleichungssystems $Ax = b$

6.2.3 poisson_problem_2d.py

Dieses Modul besitzt 5 Funktionen, die zur Erstellung des rechten Seitenvektors, zur Abbildung zwischen Gleichungsnummern und Gitterpunkten, zur Berechnung numerischer Fehler und zur Visualisierung des Fehlerverhaltens implementiert wurden.

rhs()

Erstellt den rechten Seitenvektor b für das Poisson-Problem basierend auf der Funktion f .

Parameters:

n (*int*) : Anzahl der Intervalle in jeder Dimension

f (*callable*) : Funktion auf der rechten Seite des Poisson-Problems. Der Funktionsaufruf hat die Eigenschaft $f(x_1, x_2)$ und gibt einen Skalar zurück.

Exceptions:

(*ValueError*) : wird ausgelöst, wenn $n < 2$.

Returns:

(*numpy.ndarray*) rechte Seitenvektor b

idx()

Berechnet die Nummer der Gleichung im Poisson-Problem für einen gegebenen Diskretisierungspunkt.

Parameters:

nx (*list[int]*) : Koordinaten eines Diskretisierungspunkts, multipliziert mit n

n (*int*) : Anzahl der Intervalle in jeder Dimension

Returns:

(*int*) : Nummer der entsprechenden Gleichung im Poisson-Problem.

inv_idx()

Berechnet die Koordinaten eines Diskretisierungspunkts für eine gegebene Gleichungsnummer des Poisson-Problems.

Parameters:

m (*int*) : Nummer einer Gleichung im Poisson-Problem

n (*int*) : Anzahl der Intervalle in jeder Dimension

Returns:

(*list[int]*) : Koordinaten des entsprechenden Diskretisierungspunkts, multipliziert mit n .

`compute_error()`

Berechnet den Fehler der numerischen Lösung des Poisson-Problems in Bezug auf die Unendlich-Norm.

Parameters:

- `n` (*int*) : Anzahl der Intervalle in jeder Dimension.
- `hat_u` (*array_like[numpy]*) : Finite-Differenzen-Approximation der Lösung des Poisson-Problems an den Diskretisierungspunkten.
- `u` (*callable*) : Exakte Lösung des Poisson-Problems. Der Funktionsaufruf hat die Eigenschaft $u(x_1, x_2)$ und gibt einen Skalar zurück.

Returns:

- `(float)` : Maximaler absoluter Fehler an den Diskretisierungspunkten.

`compute_error_plot()`

Gibt den Plot der Fehler der numerischen Lösung in Abhängigkeit von $N = (n - 1)^2$.

Parameters:

- `max_n` (*int*) : Maximale Anzahl von n , die berücksichtigt werden soll.
- `hat_u` (*array_like[numpy]*) : Finite-Differenzen-Approximation der Lösung des Poisson-Problems.
- `u` (*callable*) : Exakte Lösung des Poisson-Problems. Der Funktionsaufruf hat die Eigenschaft $u(x_1, x_2)$ und gibt einen Skalar zurück.

Returns: *None*, zeigt den Plot des Fehlers in Abhängigkeit von N an.

6.2.4 experiments_lu.py

Dieses Modul stellt Werkzeuge zur Verfügung, um die 2D-Poisson-Gleichung numerisch zu lösen, Fehler zu berechnen und Ergebnisse sowohl in 3D-Diagrammen als auch in Fehler-Konditions-Diagrammen zu visualisieren und basiert auf der LU-Zerlegung.

`get_u_function()`

Generiert eine analytische Lösungsfunktion $u(x_1, x_2)$ für die Poisson-Gleichung.

Parameters:

- `kappa` (*float*) : Parameter, der das oszillatorische Verhalten der Lösung beeinflusst.

Returns:

- `(callable)` : Funktion $u(x_1, x_2)$, die die Lösung darstellt

`get_f()`

Generiert die Bedingung $f(x_1, x_2)$ der Poisson-Gleichung.

Parameters:

- `kappa` (*float*) : Parameter, der das Verhalten des Terms beeinflusst.

Returns:

- `(callable)` : Funktion $f(x_1, x_2)$, die den Term darstellt.

plot_solution_3d()

Graphische Darstellung der numerischen und analytischen Lösung der Poisson-Gleichung als 3D-Plot.

Parameters:

`n (int)` : Anzahl der Diskretisierungspunkte (Intervallanzahl).

`hat_u (numpy.ndarray)` : Numerische Lösung als 1D-Array.

`u_function (callable)` : Analytische Lösungsfunktion $u(x_1, x_2)$

Exception:

`(ValueError)` , falls die Größe von `hat_u` nicht in die erwartete Form gebracht werden kann.

Returns: `None`, speichert und gibt die 3D-Plots als PNG-Datei wieder.

6.2.5 experiments_lu.main()

Diese Funktion ist das Hauptprogramm des Experimentierskripts.

Paramters: `None`

Returns: `None`, führt Analysen durch und speichert Diagramme ein, Ergebnisse werden auf der Konsole ausgegeben.

7 Literaturverzeichnis

Literatur

- [1] DR. RABUS, HELLA: *Praxisübung NLA - Projektpraktikum I - WiSe 24/25, Serie 2*, 2024. Humbold-Universität Berlin - Institut für Mathematik.
- [2] FURLAN, PETER: *Zusätze zum Gelben Rechenbuch LU-Zerlegung*. <https://www.das-gelbe-rechenbuch.de/download/Lu.pdf>, 2024. letzter Zugriff am 19. Dezember 2024.
- [3] REDAKTIONSTEAM, STUDYSMARTER: *Poisson-Gleichung: Definition, Anwendung & Beispiele*. <https://www.studysmarter.de/studium/mathematik-studium/analysis-2/poisson-gleichung/>, 2024. letzter Zugriff am 10. Dezember 2024.
- [4] SCHWEIZER, BEN: *Partielle Differentialgleichungen: Eine anwendungsorientierte Einführung*. Springer Spektrum Berlin, Heidelberg, 2024. ISBN: 978-3-662-67187-0, Seite 5-9, 19.
- [5] SCIPIY DEVELOPERS: *SciPy Sparse Matrices and Linear Algebra*. <https://docs.scipy.org/doc/scipy/reference/sparse.html>, 2024. letzter Zugriff am 19. Dezember 2024.
- [6] SCIPIY DEVELOPERS: *scipy.sparse.csr_matrix*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html, 2024. letzter Zugriff am 19. Dezember 2024.
- [7] WALTHER, ANDREA: *3. Vorlesung Numerische Algebra, LR-Zerlegung, Algorithmus 2.6*, 2024. Humbold-Universität Berlin - Institut für Mathematik, Serie 2.
- [8] WIKIPEDIA-CONTRIBUTORS: *Kondition (Mathematik)*. [https://de.wikipedia.org/wiki/Kondition_\(Mathematik\)](https://de.wikipedia.org/wiki/Kondition_(Mathematik)), 2024. letzter Zugriff am 19. Dezember 2024.
- [9] WIKIPEDIA-CONTRIBUTORS: *Poisson-Gleichung*. <https://de.wikipedia.org/wiki/Poisson-Gleichung>, 2024. letzter Zugriff am 10. Dezember 2024.

8 Übersicht verwendeter Hilfsmittel

- <https://overleaf.com>: für die Erstellung der LATEX Dokumente: Bericht und Python Dokumentation
- JupyterNotebook: Erstellung der Plots

9 Selbständigkeitserklärung

Wir versichern, dass wir in dieser schriftlichen Studienarbeit alle von anderen Autor:innen wörtlich übernommenen Stellen wie auch die sich an die Gedankengänge anderer Autor:innen eng anlehnenden Ausführungen unserer Arbeit besonders gekennzeichnet und die entsprechenden Quellen angegeben haben. Zusätzlich führen wir den Einsatz von IT-/KI-gestützten Schreibwerkzeugen zur Anfertigung dieser Arbeit im Abschnitt „Übersicht verwendeter Hilfsmittel“ vollständig auf. Hierin haben wir die Verwendung solcher Tools vollständig durch Angabe ihres Produktnamens, unserer Bezugsquelle (z.B. URL) und Angaben zu genutzten Funktionen der Software sowie zum Nutzungsumfang dokumentiert. Bei der Erstellung dieser Studienarbeit haben wir durchgehend eigenständig und beim Einsatz IT-/KI-gestützter Schreibwerkzeuge steuernd gearbeitet.