

Humboldt-Universität zu Berlin
Projektpракtikum I
Gruppe 28

Handout

Iterative vs. direkte Lösungsverfahren für lineare Gleichungssysteme - Das SOR-Verfahren im Vergleich

Mai Nguyen, Ele Tarielashvili

Abgabe: 31. Januar 2025
Betreut von
Dr. Hella Rabus

Inhaltsverzeichnis

1 Einführung und Motivation	3
2 Theorie	3
2.1 Poisson-Problem	3
2.2 Diskretisierung des Gebietes	4
2.3 Aufstellen des Lineares Gleichungssystem	4
2.4 Splitting	5
2.5 Gauß-Seidel-Verfahren	5
2.6 SOR-Verfahren	5
2.7 Konvergenz konsistenter linearer Verfahren	6
3 Experimente	6
3.1 Einfluss des Relaxationsfaktors	6
3.2 Approximationsfehler	8
3.3 Abbrückkriterium eps	9
4 Zusammenfassung	12
5 Literaturverzeichnis	13
6 Übersicht verwendeter Hilfsmittel	13
7 Selbständigkeitserklärung	13
8 Python-Dokumentation	14
8.1 Bedienungsanleitung und Hauptprogramm	14
8.1.1 Bedienungsanleitung	15
8.2 Schnittstellendokumentation	22
8.2.1 <code>block_matrix_2d.py</code>	22
8.2.2 <code>hilfsfunktionen_plot.py</code>	24
8.2.3 <code>hilfsmittel.py</code>	25
8.2.4 <code>linear_solvers.py</code>	26
8.2.5 <code>poisson_problem_2d.py</code>	27
8.2.6 <code>experiments_it.py</code>	29
8.2.7 <code>experiments_it.main()</code>	29

1 Einführung und Motivation

Das Lösen großer linearer Gleichungssysteme ist ein zentrales Problem in der numerischen Mathematik und findet breite Anwendung in der Wissenschaft und Technik, beispielsweise bei der Simulation physikalischer Prozesse oder Optimierungsproblemen. Besonders bei der Diskretisierung partieller Differentialgleichungen, wie dem Poisson-Problem, entstehen sehr große, oft dünnbesetzte Matrizen, die effiziente Algorithmen zur Lösung erfordern.

Direkte Lösungsverfahren wie die LU-Zerlegung haben den Vorteil, präzise und deterministische Lösungen zu liefern. Allerdings ist die LU-Zerlegung von dünnbesetzten Matrizen im Allgemeinen nicht dünn besetzt, was den Speicherplatzbedarf und die Rechenzeit signifikant erhöht [1].

Iterative Verfahren bieten eine attraktive Alternative, da sie den Speicherplatzbedarf durch die Nutzung der dünnbesetzten Struktur der Matrix minimieren und pro Iteration einen geringen Rechenaufwand aufweisen. Diese Verfahren konvergieren jedoch nur unter bestimmten Bedingungen und erfordern die sorgfältige Wahl von Abbruchkriterien, um eine effiziente und genaue Lösung zu gewährleisten.

In diesem Handout werden wir die numerische Lösung des Poisson-Problems anhand der Beispiefunktion $u : \Omega = (0, 1)^2 \rightarrow \mathbb{R}$ definiert durch $u(x) = x_1 \sin(\kappa\pi x_1)x_2 \sin(\kappa\pi x_2)$ für $\kappa \in \mathbb{N}$ mithilfe des iterativen SOR-Verfahrens (Successive-Over-Relaxation) untersuchen und vergleichen die Ergebnisse mit der LU-Zerlegung.

Zielstellung dieser Untersuchung ist es, ein grundlegendes Verständnis bezüglich des iterativen Lösungsverfahrens SOR zu entwickeln und dessen Effizienz im Vergleich zur LU-Zerlegung zu untersuchen.

2 Theorie

2.1 Poisson-Problem

Das Poisson-Problem beschreibt die Suche nach einer Funktion u für ein Gebiet $\Omega \subset \mathbb{R}^2$ mit einem Rand $\partial\Omega$, wobei $f \in C(\Omega; \mathbb{R})$ und $g \in C(\partial\Omega; \mathbb{R})$ gegeben sind. Die Funktion u soll die partiellen Differentialgleichung (PDE)

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega, \\ u &= g && \text{auf } \partial\Omega \end{aligned} \tag{1}$$

lösen. Wir betrachten den Fall $\Omega = (0, 1)^2$ und $g \equiv 0$. Zur numerischen Lösung der Gleichung (1) diskretisieren wir das Gebiet Ω und approximieren den Laplace-Operator mittels der finiten Differenzen. Dies führt zu einem linearen Gleichungssystem $A\hat{u} = b$, wobei der Vektor $\hat{u} \in \mathbb{R}^N$ die Näherungswerte der Funktion u an den N inneren Diskretisierungspunkten (2) darstellt [3].

2.2 Diskretisierung des Gebietes

Das offene Intervall $(0, 1)$ zerlegen wir äquidistant in $n \in \mathbb{N}$ Teilintervalle

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

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

$$X := X_1 \times X_1 = \left\{ \left(\frac{j}{n}, \frac{k}{n} \right) : 1 \leq j, k \leq n - 1 \right\}. \quad (2)$$

2.3 Aufstellen des Lineares Gleichungssystem

Mittels der Diskretisierungen in Abschnitt 2.2 und der Diskretisierung des Laplace-Operators durch die finite Differenz zweiter Ordnung, berechnen wir eine Approximation $\hat{u} \in \mathbb{R}^N$ der Lösung u (siehe (1)). Durch die Approximation des Laplace-Operators ergeben sich N lineare Gleichungen. Wir definieren jedem Diskretisierungspunkt $x \in X$ eine eindeutige Nummer $m = 1, \dots, N$. Die Diskretisierungspunkte $x = (x_1, x_2) \in X$ und $y = (y_1, y_2) \in X$ werden durch die Relation

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

geordnet. Sei $z_m \in X$ der m -te Diskretisierungspunkt, dann gilt:

$$\frac{1}{n^2} f(z_m) = 4\hat{u}_m - \hat{u}_{m+1} - \hat{u}_{m-1} - \hat{u}_{m+n} - \hat{u}_{m-n}, \quad (3)$$

was zu einem linearen Gleichungssystem in \hat{u} führt. Dies lässt sich als Matrix darstellen:

$$\mathcal{C} := \begin{bmatrix} 4 & -1 & 0 & \cdots & 0 \\ -1 & 4 & -1 & \cdots & 0 \\ 0 & -1 & 4 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & -1 \\ 0 & 0 & \cdots & -1 & 4 \end{bmatrix}, \quad A := \begin{bmatrix} \mathcal{C} & -I & 0 & \cdots & 0 \\ -I & \mathcal{C} & -I & \cdots & 0 \\ 0 & -I & \mathcal{C} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & -I \\ 0 & 0 & \cdots & -I & \mathcal{C} \end{bmatrix}. \quad (4)$$

wobei $I \in \mathbb{R}^{(n-1) \times (n-1)}$ die Einheitsmatrix ist. Für $z_1, \dots, z_N \in X$ definieren wir das folgende Gleichungssystem:

$$A\hat{u} = b := \frac{1}{n^2} \begin{bmatrix} f(z_1) \\ \vdots \\ f(z_N) \end{bmatrix}. \quad (5)$$

Um das Gleichungssystem (5) zu lösen, nutzen wir das iterative Successive-Over-Relaxation (kurz: SOR) Verfahren.

2.4 Splitting

Das sogenannte Splitting ermöglicht die Herleitung iterativer Verfahren, indem das Lösen eines linearen Gleichungssystems $Ax = b$ in eine Fixpunkttaufgabe umgewandelt wird. Die exakte Lösung des Gleichungssystems lässt sich schreiben als:

$$Ax = b \Leftrightarrow x = A^{-1}b.$$

Da die Berechnung der Inversen aufwendig ist, zerlegen wir die Matrix A in:

$$A = M - N,$$

wobei M eine reguläre Matrix ist und $N = M - A$ gilt. Damit folgt :

$$Ax = b \Leftrightarrow Mx = Nx + b \Leftrightarrow x = M^{-1} \underbrace{(Nx + b)}_{=F(x)}$$

Je nach Wahl von M und N entstehen verschiedene iterative Verfahren.

2.5 Gauß-Seidel-Verfahren

Wir zerlegen A in

$$A = D + L + U,$$

wobei D die Diagonalmatrix mit Diagonalelementen von A ist, L die strikte untere Dreiecksmatrix von A und U die strikte obere Dreiecksmatrix von A .

Für das Gauß-Seidel-Verfahren wählen wir $M = D + L$, somit folgt für die iterative Gleichung des Gauß-Seidel-Verfahrens [2]:

$$\begin{aligned} &\Rightarrow N = (D + L) - (D + L + U) = -U \\ &\Rightarrow B_{GS} = -(D + L)^{-1}U, \quad C_{GS} = (D + L)^{-1} \\ &\Rightarrow x_{k+1} = -(D + L)^{-1}Ux_k + (D + L)^{-1}b, \end{aligned}$$

wobei x_k der Näherungswert für die Lösung des Gleichungssystems nach k -Iterationen ist und ein Startwert $x^{(0)} \in \mathbb{R}^N$ gegeben ist. Wir wählen 0 für alle Einträge des Startwertes $x^{(0)}$ und betrachten expliziten Formulierung der i -ten Komponente des Lösungsvektors[2]:

$$x_{k+1}^{(i)} = x_k^{(i)} + \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_{k+1}^{(j)} - \sum_{j=i+1}^n a_{ij}x_k^{(j)} \right). \quad (6)$$

2.6 SOR-Verfahren

Das SOR-Verfahren (Successive-Over-Relaxation) ist ein iteratives Verfahren, welches auf das Gauß-Seidel-Verfahren basiert. Es verwendet einen Relaxationsparameter $\omega > 0$, der

für die Optimierung der Konvergenzgeschwindigkeit zuständig ist. Die Iterationsvorschrift für das SOR-Verfahren lautet:

$$x_{k+1}^{(i)} = (1 - \omega)x_k^{(i)} + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_{k+1}^{(j)} - \sum_{j=i+1}^n a_{ij}x_k^{(j)} \right). \quad (7)$$

Der Relaxationsparameter ω spielt eine entscheidende Rolle. Wenn $\omega = 1$ entspricht das dem Gauß-Seidel-Verfahren. Falls $\omega < 1$ gilt die Unterrelaxation, bei der die Konvergenz langsamer ist, aber das Verfahren stabiler wird. Falls $\omega > 1$ sprechen wir von einer Überrelaxation, bei der die Konvergenz schneller sein kann, wobei wir dafür einen optimalen Wert für ω experimentell bestimmen müssen [4].

Da unsere Matrix A symmetrisch und positiv definit ist, gilt, dass das SOR-Verfahren genau dann konvergiert, wenn ω im Intervall $(0, 2)$ liegt [2].

2.7 Konvergenz konsistenter linearer Verfahren

Sei $x_{k+1} = Bx_k + c$ eine Iterationsformel, wobei $B \in \mathbb{R}^{n \times n}$ die Iterationsmatrix und $c \in \mathbb{R}^n$ ein Vektor ist. Sei x^* die zugehörige exakte Lösung.

$$\|x_{k+1} - x^*\| \leq \rho(B)\|x_k - x^*\| \quad \text{mit} \quad \rho(B) = \max\{|\lambda| : \lambda \text{ ist Eigenwert von } B\}. \quad (8)$$

Das Verfahren konvergiert linear, wenn $\rho(B) < 1$, und die Konvergenzrate ist umso schneller, je kleiner $\rho(B)$ ist.

3 Experimente

Für die Untersuchung betrachten wir für $\kappa \in \mathbb{N}$ die folgende Funktion $u : \Omega \rightarrow \mathbb{R}$

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

Ziel der Experimente ist der Vergleich zwischen dem SOR-Verfahren und der LU-Zerlegung hinsichtlich des Lösens großer bzw komplexer linearer Gleichungssysteme. Das SOR-Verfahren ist anders als die LU-Zerlegung ein iteratives Verfahren und ist somit von mehreren Parametern abhängig, welche die Effizienz beeinflussen. Das heißt um einen fairen Vergleich der beiden Verfahren zu ermöglichen, müssen zunächst Experimente durchgeführt werden, um die optimalen Parametereinstellungen zu garantieren.

3.1 Einfluss des Relaxationsfaktors

Die Wichtigkeit der richtigen Wahl des Relaxationsfaktor lässt sich an der Abbildung 1 erkennen.

Für kleine Werte von ω konvergiert das Verfahren sehr langsam. Der Fehler nimmt nur langsam ab, und viele Iterationen sind erforderlich, um eine akzeptable Genauigkeit zu erreichen. Mit einem steigenden Relaxationsfaktor verbessert sich die Konvergenzrate

Error Development for Different ω ($n=40$)

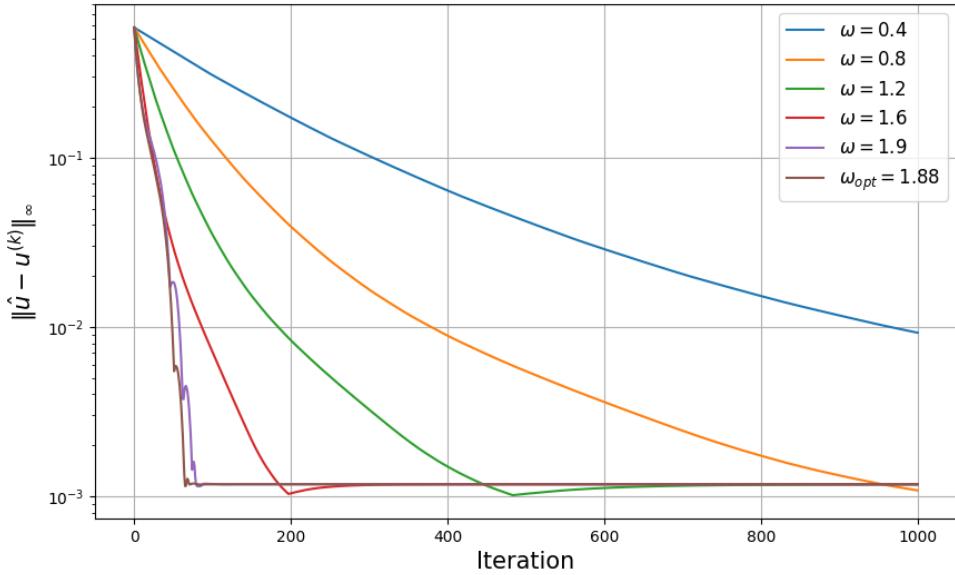


Abbildung 1: Approximationsfehler mit $n = 40$ und verschiedenen ω

deutlich, das heißt der Fehler sinkt schneller. Dies gilt bis der optimale Wert ($\omega \approx 1.88$) erreicht wird, mit dem das Verfahren in unter 100 Iterationen an seine höchste Genauigkeit gelangt. Für zu große Werte von ω verschlechtert sich die Konvergenz jedoch wieder im Vergleich zum optimalen Relaxationsfaktor (ω_{opt}).

Die Ergebnisse des Plots zeigen deutlich, dass die Wahl des Relaxationsparameters ω einen erheblichen Einfluss auf die Konvergenzgeschwindigkeit des SOR-Verfahrens hat. Insbesondere der optimale Wert liefert die schnellste Konvergenz, während zu kleine oder zu große Werte von ω die Leistung des Verfahrens verschlechtern können. Um den idealen Wert ω_{opt} zu bestimmen, betrachten wir im nächsten Experiment 2 den Spektralradius(ρ_ω) der Iterationsmatrix in Abhängigkeit von ω . Der Spektralradius ist ein direktes Maß für die Konvergenz, denn je kleiner der Spektralradius, desto kleiner die Konvergenzrate. Somit entspricht sein Minimum dem optimalen Relaxationsparameter (8) .

Für die genauere Untersuchung des Spektralradius wurde die Funktion `compute_spectral_radius` implementiert, welche den Spektralradius der Iterationsmatrix des SOR-Verfahrens:

$$B_{GS}^\omega = -(\omega L + D)^{-1}((\omega - 1)D + \omega U)$$

berechnet [2]. Um den Spektralradius zu bestimmen, werden die Eigenwerte der Matrix B_{GS}^ω berechnet. Hierzu wird die Funktion `eigvals` aus `scipy.linalg` verwendet, die direkt alle Eigenwerte einer quadratischen Matrix bestimmt. Schließlich wird der Spektralradius als der größte Betrag der berechneten Eigenwerte zurückgegeben.

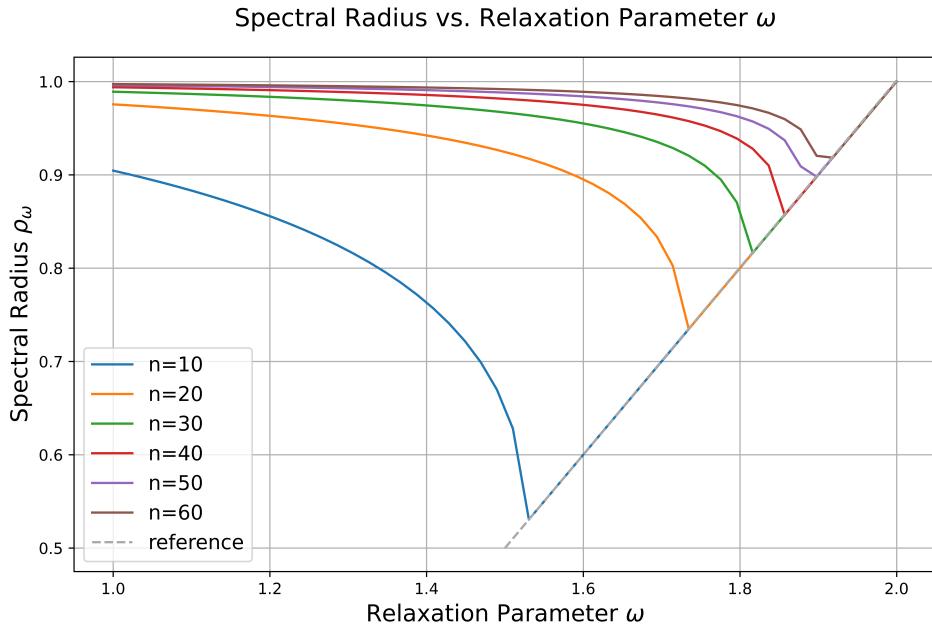


Abbildung 2: Spektralradius ρ_ω für verschiedene n

Der Spektralradius erreicht für alle verschiedenen Schrittweiten einen Tiefpunkt zwischen 1.5 und 2 und fängt daraufhin an linear anzusteigen.

Es lässt sich somit behaupten ω_{opt} nimmt Werte in $(1, 2)$ an und steigt linear mit zunehmender Gittergröße n an. Das heißt für eine steigende Feinheit der Diskretisierung nähert sich der ideale Relaxationsfaktor der 2 an.

Diese Erkenntnis kann genutzt werden um die nachfolgenden Experimente mit den berechneten Relaxationsfaktoren durchzuführen und die höchstmögliche Effizienz des Verfahrens zu nutzen.

3.2 Approximationsfehler

Abbildung 3 zeigt die Entwicklung des maximalen absoluten Fehlers für das SOR-Verfahren und die LU-Zerlegung bei verschiedenen Diskretisierungen ($n = 30, n = 40, n = 50$) 150 Iterationen. Der maximale absolute Fehler nimmt mit zunehmender Anzahl der Iterationen ab. Das ist ein erwartetes Verhalten bei iterativen Verfahren, da sie darauf abzielen, sich der exakten Lösung schrittweise zu nähern. Die Konvergenzrate scheint für alle Diskretisierungen ähnlich zu sein, wobei der Fehler für $n = 50$ zu Beginn am größten ist und am langsamsten abnimmt, d.h. mehr Iterationen benötigt. Dennoch bieten Approximationen mit größeren n Lösungen mit einer höheren Genauigkeit.

Die LU-Zerlegung liefert eine sehr genaue Lösung mit einem geringen maximalen absoluten Fehler. Das SOR-Verfahren benötigt mehrere Iterationen, um die gleiche Genauigkeit zu erreichen. Der maximale absolute Fehler bleibt dann jedoch konstant für die

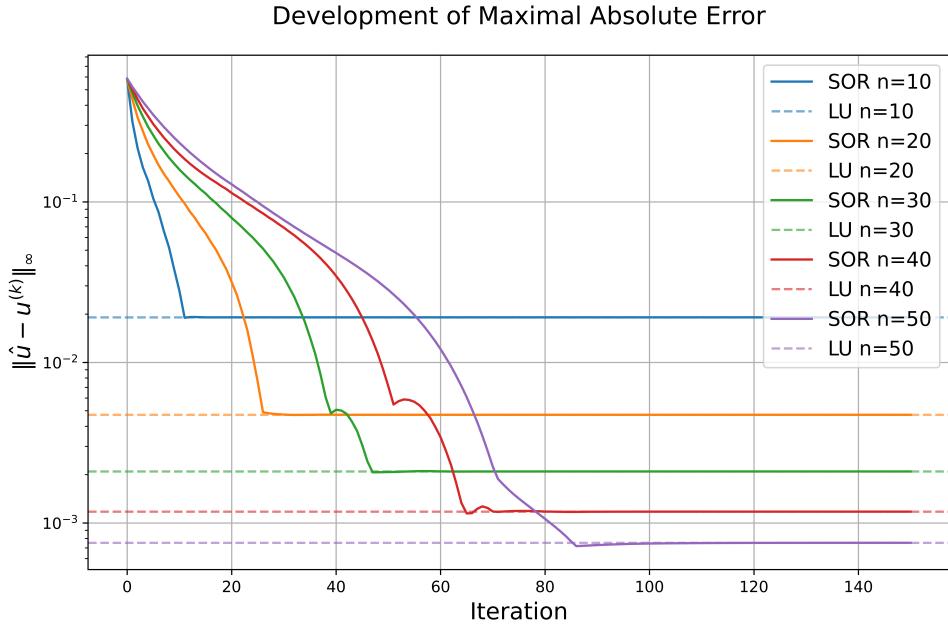


Abbildung 3: Approximationenfehler für ω_{opt}

restlichen Iterationen. Grund dafür ist der Diskretisierungsfehler. Der Diskretisierungsfehler entsteht, weil das kontinuierliche Problem durch ein diskretes Gitter angenähert wird. Die exakte Lösung der diskreten Gleichung ist nämlich nur eine Näherung der exakten Lösung des kontinuierlichen Problems und hängt von der Schrittweite ab ($\frac{1}{n}$). Zu Beginn der Iterationen dominiert der Iterationsfehler. Der Diskretisierungsfehler setzt jedoch eine untere Grenze für den maximalen absoluten Fehler. Dies erklärt den Knick im Graphen: Sobald der Iterationsfehler kleiner wird als der Diskretisierungsfehler, stagniert der Gesamtfehler.

3.3 Abbruckriterium eps

Für die Abbildungen 4, 5 und 6 wählen wir allgemein $\text{max_iter} = 5000$, $\text{var_x} = 1e - 10$ und ein festes $\varepsilon = 1e - 8$ als Abbruckriterien, wobei wir zusätzlich ε in Abhängigkeit von h betrachten. Allerdings gilt, dass max_iter und var_x zu ignorieren sind, da für alle ausgerechneten Werte für die genannten Abbildungen das Abbruckriterium ε erfüllt wird.

In der Abbildung 4 wird der maximale Fehler in Abhängigkeit von $N = (n - 1)^2$ für das SOR-Verfahren mit ε in Abhängigkeit von h als Abbruckriterium. Wir sehen, dass die Fehler für $\varepsilon = h^{-2}$ und $\varepsilon = h^0$ nahezu konstant bleiben, unabhängig von der Anzahl der Gitterpunkte N . Die Plots verlaufen entlang der Referenzlinie $\mathcal{O}(\frac{1}{N^0}) = \mathcal{O}(1)$. Für $\varepsilon = h^2$ zeigt sich eine stärkere Schwankung im Fehlerverlauf, was auf eine instabilere Konvergenz bei diesem Parameter hinweist. Es ist zu erkennen, dass die Plots für $\varepsilon = h^4$

und $\varepsilon = h^6$ nahezu identisch sind. Beide weisen ein deutlich kleineres Fehlerniveau auf und verlaufen parallel zur Referenzlinie $\frac{1}{N^2}$. Insbesondere wird deutlich, dass $\varepsilon = h^4$ und $\varepsilon = h^6$ die kleinsten maximalen Fehler aufweisen, sodass sie besonders geeignet für präzise Berechnungen im Rahmen des SOR-Verfahrens sind. Da $\varepsilon = h^4$ eine geringere Genauigkeit benötigt als $\varepsilon = h^6$ als Schwellenwert, wählen wir für unsere weitere Untersuchung $\varepsilon = h^4$ als h -abhängiges ε .

Error vs Number of Inner Points for Different $\varepsilon(k)$

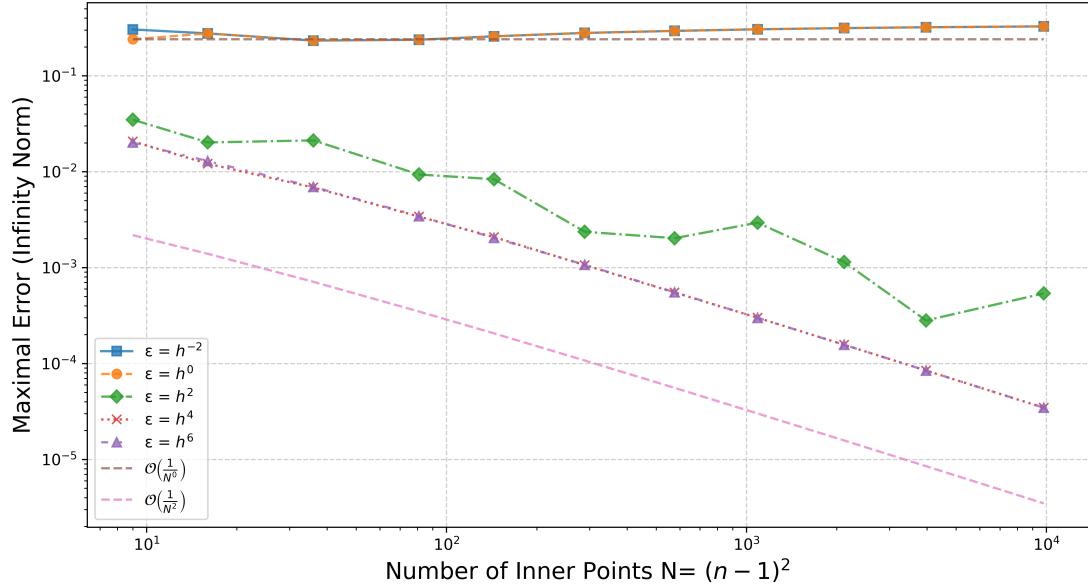


Abbildung 4: Fehlerplot für h -abhängige ε mit $\text{max_iter} = 5000$, $\text{var_x} = 1e - 10$ und $\omega = 1.88$

In der Abbildung 5 wird der maximale Fehler in Abhängigkeit von N für die LU-Zerlegung und das SOR-Verfahren für das fixierte $\varepsilon = 1e - 8$ und h -abhängige $\varepsilon = h^4$ untersucht. Die drei Graphen für die LU-Zerlegung und das SOR-Verfahren sind nahezu identisch zueinander, dies deutet darauf hin, dass beide Verfahren in Bezug auf die Genauigkeit gleichwertig sind für $N \leq 10^4$. Zudem verlaufen die Plots parallel zur Referenzlinie $\mathcal{O}(\frac{1}{N})$, was mit der theoretischen Konvergenzrate übereinstimmt. Das bedeutet, dass das feste ε und das h -abhängige $\varepsilon = h^4$ in Bezug auf die Genauigkeit stabile und zuverlässige Ergebnisse liefern. Für eine erweiterte Bewertung beider Verfahren betrachten wir die Laufzeiten der Algorithmen, um neben der Genauigkeit auch die Effizienz in Bezug auf Rechenzeit zu beurteilen.

In der Abbildung 6 werden die Laufzeiten der LU-Zerlegung sowie des SOR-Verfahrens mit festem $\varepsilon = 1e - 8$ und h -abhängigem $\varepsilon = h^4$ untersucht. Um Ausreißer zu minimieren, wurden 5 Durchläufe durchgeführt. Es ist zu sehen, dass die Laufzeit des SOR-Verfahrens insgesamt am höchsten ist. Für $N < 10^3$ steigen die Plots des SOR-Verfahrens fast parallel zur Referenzlinie $\mathcal{O}(N)$, was der erwarteten Skalierung des Rechenaufwands entspricht, wobei der Graph hier mit dem h -abhängigen ε schneller ist als der festgelegte

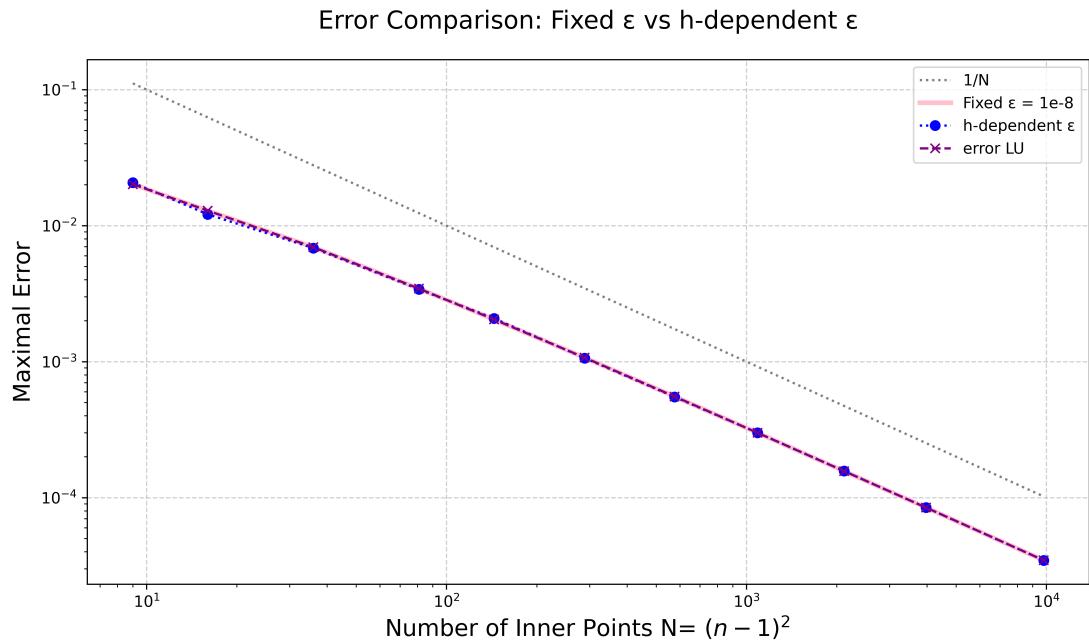


Abbildung 5: Fehlerplot LU und SOR mit 5 Durchläufe
 $\max_iter = 5000$, $\text{var_x} = 1e - 10$ und $\omega = 1.88$

ε . Für $N > 10^3$ verlaufen die beiden Plots des SOR-Verfahrens fast parallel zu $\mathcal{O}(N^2)$, hier sind die Laufzeiten der beiden Graphen fast identisch. Die LU-Zerlegung hat für $N \leq 10^2$ die geringsten Laufzeiten, wobei diese sich der Referenzlinie $\mathcal{O}(N^{1.3})$ nähert. Ab $N > 10^2$ ist eine abrupte Steigung der Laufzeit zu beachten von ca. 0.5 ms zu 10 ms. Die Laufzeiten bei der LU-Zerlegung für $10^2 < N < 10^4$ verlaufen fast gleich der Referenzlinie $\mathcal{O}(N^{0.6})$, wobei es bei $N = 10^4$ erneut zu einer abrupten Steigung auf 800 ms kommt. Die Abbildung 6 zeigt allgemein, dass die Laufzeit für die LU-Zerlegung für $N \leq 10^4$ am kleinsten ist.

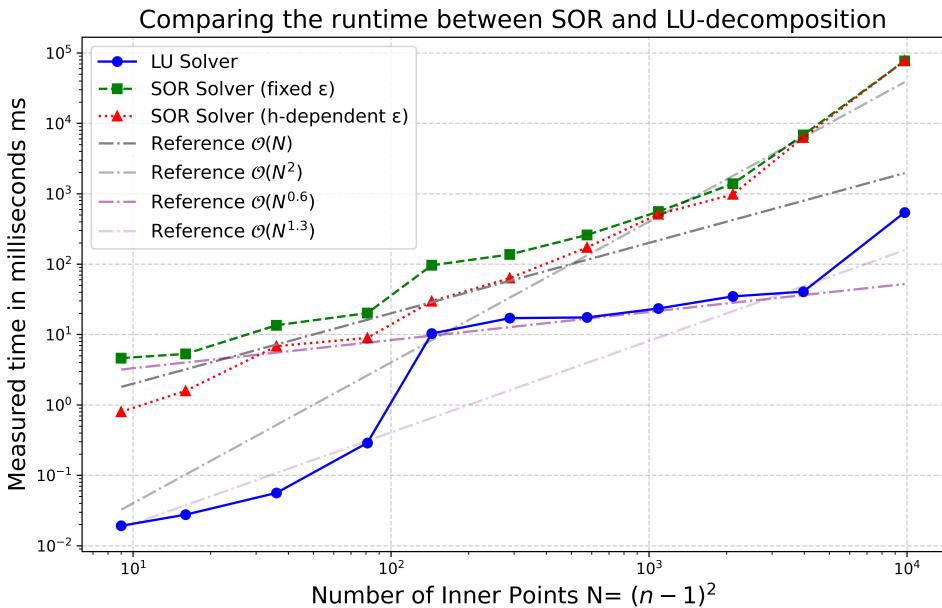


Abbildung 6: Laufzeit mit 5 Durchgängen, $\text{max_iter} = 5000$, $\text{var_x} = 1e - 10$ und $\omega = 1.88$

4 Zusammenfassung

Das Ziel unserer Untersuchung war es, eine effizientere Alternative zur LU-Zerlegung für das Lösen komplexer linearer Gleichungssysteme zu finden. Die LU-Zerlegung ist bei dünnbesetzten Matrizen aufgrund des Fill-in-Effekts speicherineffizient und für sehr große Matrizen nicht praktikabel. Daher haben wir das SOR-Verfahren (Successive Over-Relaxation) als iterative Methode untersucht, um mögliche Vorteile zu identifizieren.

Unsere Experimente zeigten, dass die Effizienz des SOR-Verfahrens stark von der Wahl des Relaxationsparameters ω abhängt. Der optimale Relaxationsfaktor liegt zwischen 1,5 und 2 und steigt mit zunehmender Gittergröße n an, wobei er sich dem Wert 2 annähert. Ein weiterer wichtiger Faktor ist das Abbruchkriterium ϵ , das sorgfältig gewählt werden muss, um eine effiziente Konvergenz zu gewährleisten. Insbesondere ein h -abhängiges $\epsilon = h^4$ erwies sich als besonders geeignet, um präzise Ergebnisse zu erzielen.

Beim Vergleich der Laufzeiten zeigte sich, dass die LU-Zerlegung für $N \leq 10^4$ schneller ist als das SOR-Verfahren. Allerdings konnten wir den Punkt, an dem das SOR-Verfahren effizienter wird, aufgrund der begrenzten Rechenleistung nicht erreichen. Unsere Hoffnung ist, dass bei noch größeren Matrizen (z. B. $n \geq 10^4$) ein Sprung in der Effizienz des SOR-Verfahrens auftritt, was jedoch weiterer Untersuchungen auf leistungsstärkeren Rechnern bedarf. Für kleinere Systeme bleibt die LU-Zerlegung jedoch die schnellere Methode. Weitere Forschung könnte sich auf die Optimierung des SOR-Verfahrens für extrem große Matrizen und die Nutzung leistungsfähigerer Hardware konzentrieren, um den erwarteten Effizienzsprung zu bestätigen.

5 Literaturverzeichnis

Literatur

- [1] LANG, STEFAN: *Lösung tridiagonaler und dünnbesetzter linearer Gleichungssysteme*. https://conan.iwr.uni-heidelberg.de/old-site/teaching/phlr_ws2012/lecture21.pdf, 2012. letzter Zugriff am 27. Januar 2025.
- [2] MEISTER, ANDREAS: *Numerik linearer Gleichungssysteme*. Springer Spektrum Wiesbaden, Wiesbaden, 5. Auflage , 2015. ISBN: 978-3-658-07199-8, Seite 89-90.
- [3] RABUS, HELLA: *Praxisübung NLA - Projektpraktikum I - WiSe 24/25, Serie 2*, 2024. Humboldt-Universität Berlin - Institut für Mathematik.
- [4] RICHTER, THOMAS, HENRY VON WAHL THOMAS WICK: *Einführung in die Numerische Mathematik*. Springer Spektrum Berlin, Heidelberg, Berlin, Heidelberg, 2. Auflage , 2024. ISBN: 978-3-662-69581-4, Seite 113-117.

6 Übersicht verwendeter Hilfsmittel

- <https://overleaf.com>: für die Erstellung der L^AT_EX Dokumente: Handout und Python Dokumentation
- Visual Studio Code: Erstellung der Pythondateien und Plots

7 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.

8 Python-Dokumentation

8.1 Bedienungsanleitung und Hauptprogramm

Das Pythonmodul zur Untersuchung des SOR-Verfahrens durch numerisches Lösen des Poisson-Problems und Laufzeitenvergleich mit der LU-Zerlegung besteht aus den Modulen `block_matrix_2d.py`, `experiments_it.py`, `experiments_sor_final.py`, `hilfsfunktionen_plot.py`, `hilfsmittel.py`, `linear_solvers.py` und `poisson_problem_2d.py`. Dieses Pythonmodul enthält zwei Experimentierskripts.

`experiments_sor_final.py`

Das Modul `experiments_sor_final.py` dient der Untersuchung des SOR-Verfahrens zur Lösung der Poisson-Gleichung. Es ist ein Experimentierskript, welches die Möglichkeit bietet, die Konvergenz des Verfahrens sowie die Bestimmung des optimalen Relaxationsparameters ω zu untersuchen.

Das Skript enthält definierte Funktionen aus `block_matrix_2d.py`, `hilfsmittel.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.

In `hilfsmittel.py` sind Funktionen definiert, die die Nutzereingabe überprüft und den Nutzer fragt, ob dieser die angezeigten Plots speichern möchte.

Im Programm `linear_solvers.py` sind die Funktionen `solve_lu` und `solve_sor` zur Lösung eines linearen Gleichungssystems $Ax = b$ definiert.

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 Programm wird nur gestartet, wenn `experiments_sor_final.py` mittels `python3 -m experiments_sor_final.py` gestartet wird.

`experiments_it.py`

Das Modul `experiments_it.py` ist ein Experimentierskript. Es dient der experimentellen Untersuchung der Poisson-Gleichung mit Hilfe von LU-Zerlegung und Successive Over-Relaxation (SOR). Es bietet Werkzeuge zum numerischen Lösen der 2D-Poisson-Gleichung, zur Berechnung von Fehlern und zur Visualisierung der Ergebnisse in Diagrammen. Das Experimentierskript enthält definierte Funktionen aus `block_matrix_2d.py`, `hilfsfunktionen_plot.py`, `hilfsmittel.py`, `linear_solvers.py` und `poisson_problem_2d.py`.

Die Programme `block_matrix_2d.py`, `hilfsmittel.py` `linear_solvers.py` und `poisson_problem_2d.py` und deren Funktionalitäten wurden bereits in 8.1 erwähnt, wobei in `hilfsmittel.py` für dieses Experimentierskript zusätzlich die Funktion `measure_time` genutzt wird, um die Laufzeiten der Algorithmen gemessen wird. Zusätzlich wird aus `hilfsmittel.py` die Funktion `zahlenliste` genutzt, um eine Zahlenliste zu erstellen.

Das Programm `hilfunktionen_plot.py` enthält Funktionen, um Plots zu erstellen bzgl. der maximalen Fehler und der Laufzeiten.

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

8.1.1 Bedienungsanleitung

`experiments_sor_final.py` - Bedienungsanleitung

Starten Sie zuerst das Programm `experiments_sor_final.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. Der Nutzer hat die Möglichkeiten auszuwählen, was er untersuchen möchte. Dafür tippe er:

'1' für die Untersuchung der Approximation der Lösung des Poisson Problems mit dem SOR-Verfahren,

'2' für die Untersuchung der Konvergenz des Verfahrens hinsichtlich der Gitterweite,

'3' für die Untersuchung der Konvergenz hinsichtlich der Wahl von Omega,

'4' für die Untersuchung der Entwicklung des optimalen Relaxationsfaktors,

'5' für das Ausrechnen des optimalen Relaxationsfaktors.

Bei einer leeren Eingabe wird das Programm abgebrochen. Für jeden ausgegebene Plot wird der Nutzer gefragt, ob er diesen speichern möchte. Falls sie gespeichert werden soll, werden diese im gleichen Ordner wie `experiments_sor_final.py` gespeichert. Zu berücksichtigen ist, dass keine PNG-Datei mit dem gleichen Dateinamen im Ordner des Programms vorliegen sollte, da diese sonst überschrieben wird.

`experiments_sor_final.py` - Beispielausführung

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

Hallo, dies ist ein Experimentierskript für die Untersuchung
des SOR-Verfahrens.

Ein besonderer Schwerpunkt hierbei ist die Untersuchung des optimalen
Relaxationsfaktors (Omega).

Sie können folgendes untersuchen für ein tieferes Verständnis:

1. Approximation der Lösung des Poisson Problems mit dem SOR-Verfahren.
2. Konvergenz des Verfahrens hinsichtlich der Gitterweite.

3. Konvergenz hinsichtlich der Wahl von Omega.
4. Entwicklung des optimalen Relaxationsfaktors.
5. Ausrechnen des optimalen Relaxationsfaktors.

Ihre Wahl (1 - 5):

Tippe der Nutzer '2'. Dann wird ihm folgendes angezeigt:

Ihre Wahl (1 - 5): 2

Wollen Sie den Plot unter dem Namen konvergenz.png speichern?

Es sollte keine Datei mit dem gleichen Dateiname im Ordner des Programms vorliegen, da diese sonst überschrieben wird.
(1 für ja, 2 für nein):

Zum Speichern des Plottes, tippe der Nutzer die '1'. Dann wird folgendes wiedergegeben:

(1 für ja, 2 für nein): 1

Plot gespeichert als konvergenz.png.

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

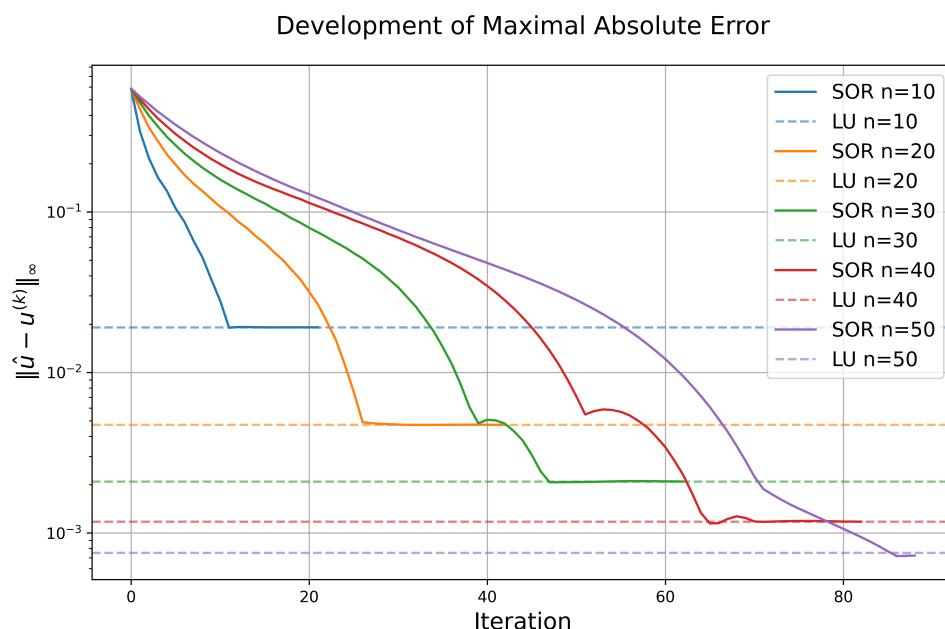


Abbildung 7: Untersuchung der Konvergenz hinsichtlich der Gitterweite

experiments_it.py - Bedienungsanleitung

Starten Sie zuerst das Programm `experiments_it.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.

Der Nutzer hat die Möglichkeit eine Liste für n -Werte zu erstellen, in dem er jede Zahl einzeln zur Liste hinzufügt und kann sich den Relaxationparameter ω wählen, wobei im Intervall $(1, 2)$ liegen soll. Das Modul gibt die Laufzeiten für die einzelnen Berechnungen aus. Falls bei der Eingabe des Relaxationsparameter eine leere Eingabe getätigt wurde, wird das Programm abgebrochen, ansonsten werden drei Plots wiedergegeben, wobei der Nutzer die Wahl hat, ob er diese speichern möchte. Die Plots werden im gleichen Ordner wie `experiments_it.py` gespeichert. Zu berücksichtigen ist, dass keine PNG-Datei mit dem gleichen Dateinamen im Ordner des Programms vorliegen sollte, da diese sonst überschrieben wird.

experiments_it.py - Beispielausführung

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

```
Wir erstellen eine Liste von Zahlen n, die wir untersuchen wollen.  
Bitte beachten Sie, dass Berechnungen für Zahlen größer als 50  
höhere Rechenzeiten benötigen (möglicherweise mehr als 2 Minuten).  
Die eingegebenen Zahlen werden in N=(n-1)^2 umgerechnet.  
Bitte geben Sie eine positive Zahl ein:  
Um die Liste der Zahlen zu beenden, lassen Sie die Eingabe frei
```

Als Beispielausführung erstellt der Nutzer eine Liste mit den Zahlen $[4, 5, 7, 10, 13, 18, 25, 34, 47, 64, 100]$ und wählt $\omega = 1.88$. Für das Vereinfachen des Lesens wurden wiederholende Aussagen gekürzt bzw. mit '...' gekennzeichnet. Der Nutzer sieht folgende Aussgabe und wählt jeweils '1' für das Speichern der drei Plots:

```
Bitte geben Sie eine positive Zahl ein:  
Um die Liste der Zahlen zu beenden, lassen Sie die Eingabe frei  
4  
...  
Die Zahl 64.0 wurde hinzugefügt.  
Bitte geben Sie eine positive Zahl ein:  
Um die Liste der Zahlen zu beenden, lassen Sie die Eingabe frei  
100  
Die Zahl 100.0 wurde hinzugefügt.  
Bitte geben Sie eine positive Zahl ein:  
Um die Liste der Zahlen zu beenden, lassen Sie die Eingabe frei
```

Ihre Eingegebene Liste lautet:

[4.0, 5.0, 7.0, 10.0, 13.0, 18.0, 25.0, 34.0, 47.0, 64.0, 100.0]

Sie sollen nun ein Omega für die Untersuchung

bestimmen. Bitte wählen Sie eine Zahl im

Intervall (1,2). Bei einer Leeren Eingabe wird

das Programm abgebrochen. 1.88

Wir untersuchen das Poisson-Problem mit einer Beispielfunktion

Dafür werden wir drei Plots untersuchen.

N=9.0:

LU-Zeit: 0.064309 ms

SOR-Zeit (festes): 5.484508 ms

Abbruchkriterium: eps

SOR-Zeit (h-abhängiges): 1.105092 ms

Abbruchkriterium: eps

N=16.0:

LU-Zeit: 0.032625 ms

SOR-Zeit (festes): 4.715142 ms

Abbruchkriterium: eps

SOR-Zeit (h-abhängiges): 1.396292 ms

Abbruchkriterium: eps

N=36.0:

LU-Zeit: 0.051717 ms

SOR-Zeit (festes): 7.943075 ms

Abbruchkriterium: eps

SOR-Zeit (h-abhängiges): 3.221933 ms

Abbruchkriterium: eps

N=81.0:

LU-Zeit: 0.105758 ms

SOR-Zeit (festes): 17.378267 ms

Abbruchkriterium: eps

SOR-Zeit (h-abhängiges): 8.462242 ms

Abbruchkriterium: eps

N=144.0:

LU-Zeit: 22.012109 ms

SOR-Zeit (festes): 128.818842 ms

Abbruchkriterium: eps

SOR-Zeit (h-abhängiges): 47.979700 ms

Abbruchkriterium: eps

N=289.0:

LU-Zeit: 25.682867 ms

SOR-Zeit (festes): 172.397592 ms

Abbruchkriterium: eps

SOR-Zeit (h-abhängiges): 122.457208 ms

Abbruchkriterium: eps

```

N=576.0:
    LU-Zeit: 26.660442 ms
    SOR-Zeit (festes ): 372.851108 ms
        Abbruchkriterium: eps
    SOR-Zeit (h-abhängiges ): 214.690917 ms
        Abbruchkriterium: eps
N=1089.0:
    LU-Zeit: 36.894008 ms
    SOR-Zeit (festes ): 614.453508 ms
        Abbruchkriterium: eps
    SOR-Zeit (h-abhängiges ): 467.702808 ms
        Abbruchkriterium: eps
N=2116.0:
    LU-Zeit: 49.613558 ms
    SOR-Zeit (festes ): 1431.163300 ms
        Abbruchkriterium: eps
    SOR-Zeit (h-abhängiges ): 1164.638367 ms
        Abbruchkriterium: eps
N=3969.0:
    LU-Zeit: 52.395241 ms
    SOR-Zeit (festes ): 7272.299683 ms
        Abbruchkriterium: eps
    SOR-Zeit (h-abhängiges ): 6611.775175 ms
        Abbruchkriterium: eps
N=9801:
    LU-Zeit: 542.257066 ms
    SOR-Zeit (festes ): 77017.061850 ms
        Abbruchkriterium: eps
    SOR-Zeit (h-abhängiges ): 77950.757008 ms
        Abbruchkriterium: eps
Wollen Sie den Plot unter dem Namen
plot_error_for_eps_variants.png speichern?
Es sollte keine Datei mit dem gleichen Dateiname im
Ordner des Programms vorliegen, da diese sonst
überschrieben wird.
(1 für ja, 2 für nein): 1
Plot gespeichert als plot_error_for_eps_variants.png.

```

Der Plot `plot_error_for_eps_variants.png` wird angezeigt und im selben Ordner wie das Programm gespeichert.

Error vs Number of Inner Points for Different $\varepsilon(k)$

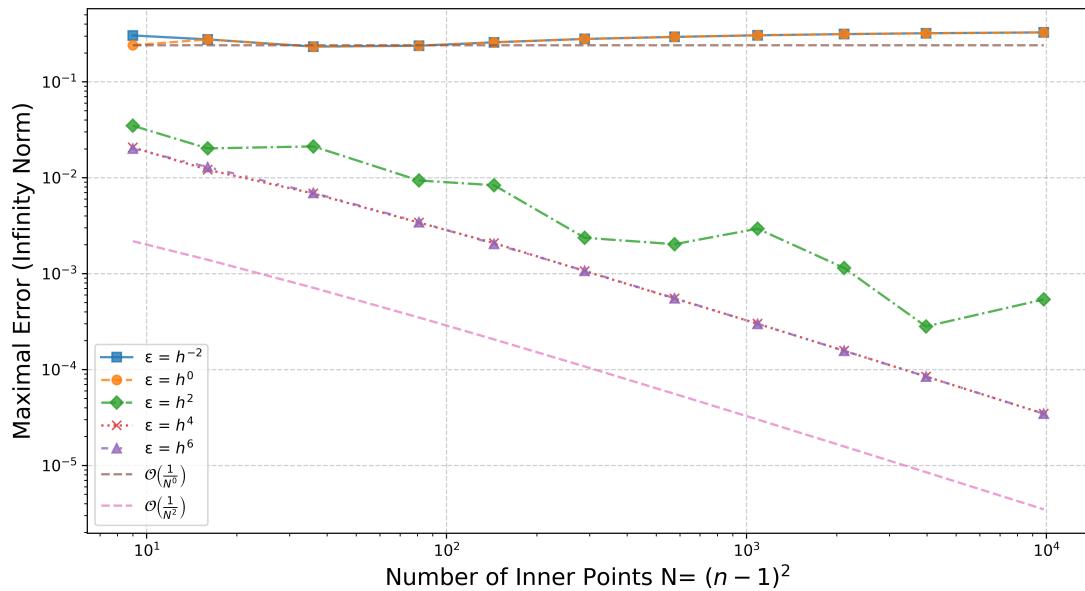


Abbildung 8: Maximaler Fehler in Abhangigkeit von N mit h -abhangigen ε

Wollen Sie den Plot unter dem Namen `plot_fixed_and_h_dependent_errors.png` speichern?

Es sollte keine Datei mit dem gleichen Dateinamen im Ordner des Programms vorliegen, da diese sonst uberschrieben wird.

(1 fur ja, 2 fur nein): 1

Plot gespeichert als `plot_fixed_and_h_dependent_errors.png`.

Der Plot `plot_fixed_and_h_dependent_errors.png` wird angezeigt und im selben Ordner wie das Programm gespeichert.

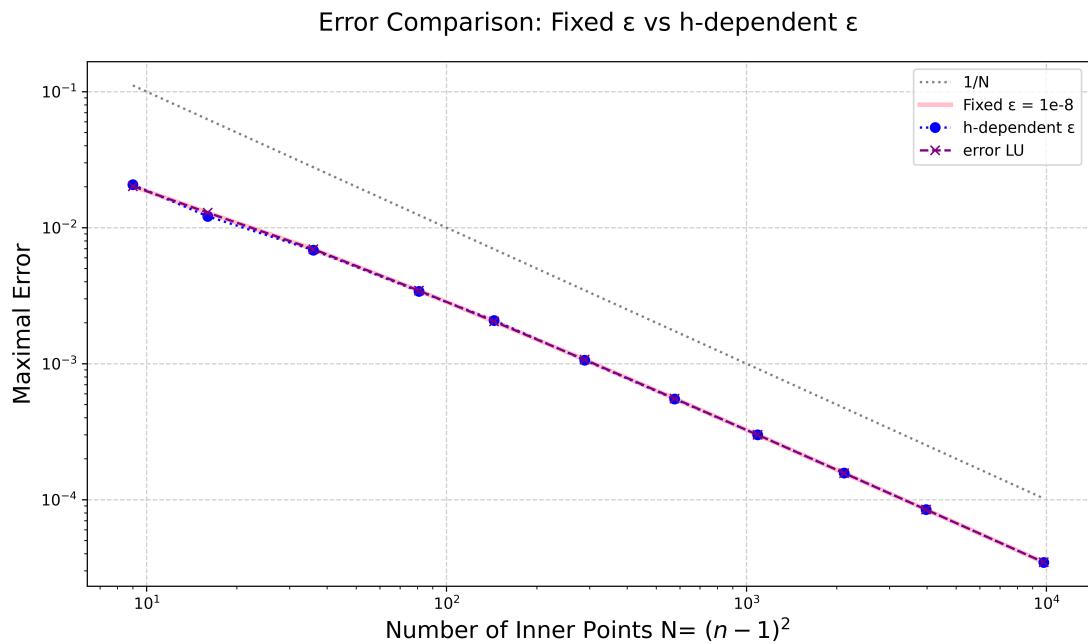


Abbildung 9: Maximaler Fehler in Abhängigkeit von N Vergleich LU, SOR mit festem und h -abhängigen ε

Wollen Sie den Plot unter dem Namen `runtime_comparison.png` speichern?
 Es sollte keine Datei mit dem gleichen Dateinamen im Ordner des Programms vorliegen, da diese sonst überschrieben wird.
 (1 für ja, 2 für nein): 1
 Plot gespeichert als `runtime_comparison.png`.

Der Plot `runtime.png` wird angezeigt und im selben Ordner wie das Programm gespeichert.

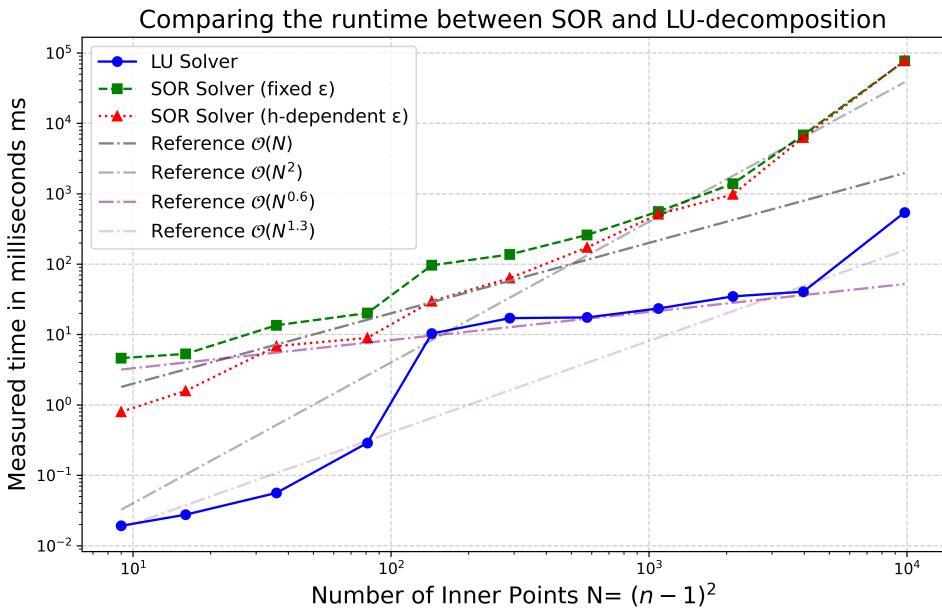


Abbildung 10: Laufzeitvergleich LU, SOR mit festem und h -abhängigem ε

8.2 Schnittstellendokumentation

Die Experimentierskripte enthalten Funktionen aus den Programmen `block_matrix_2d.py`, `hilfsfunktionen_plot.py`, `hilfsmittel.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 `hilfsfunktionen_plot.py` sind Hilfsfunktionen zum Plotten der Untersuchungen definiert und in `hilfsmittel.py` sind allgemeine Hilfsfunktionen definiert. In `linear_solvers.py` sind Funktionen 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.

8.2.1 `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 .

Parameters:

`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 .

8.2.2 hilfsfunktionen_plot.py

Dieses Modul besteht aus drei Funktionen zum Plotten von Untersuchungen.

`plot_error_for_eps_variants()`

Plotten den maximalen Fehler mit der Unendlichkeitsnorm für verschiedene h -abhängige Epsilon.

Parameters:

`Ns (list of int)` : Anzahl der inneren Gitterpunkte

`errors (dict)` : *dictionary*, dessen Schlüssel die Epsilon-Werte (k) sind und dessen Werte Listen von Fehlern sind, die den jeweiligen Gittergrößen entsprechen

`epsilon_values (list of int)` : Liste von Ganzzahlen, die die verschiedenen k -Werten für $\varepsilon = h^k$ darstellen

Returns: `None`, denn es wird ein Plot ausgegeben.

`plot_fixed_and_h_dependent_errors()`

Plotten den maximalen Fehler mit der Unendlichkeitsnorm für ein h -abhängige Epsilon, ein festes Epsilon und für die LU-Zerlegung.

Parameters:

Ns (*list of int*) : Anzahl der inneren Gitterpunkte
errors_fixed (*list of floats*) : Liste von Fehlern mit SOR für ein festes ε
errors_h (*list of floats*) : Liste von Fehlern mit SOR für ein h -abhängiges ε
errors_lu (*list of floats*) Liste von Fehler von der LU-Zerlegung

Returns: *None*, denn es wird ein Plot ausgegeben.

plot_fixed_and_h_dependent_errors()

Plotten die Laufzeit für ein h -abhängige ε , ein festes ε und für die LU-Zerlegung.

Parameters:

numbers (*list of int*) : Anzahl der inneren Gitterpunkte
lu_times (*list of floats*) : Liste der gemessenen Zeit für die LU-Zerlegung
sor_times_fixed (*list of floats*) : Liste der gemessenen Zeit mit SOR für ein festes Epsilon
sor_times_h (*list of floats*) : Liste der gemessenen Zeit mit SOR für ein h -abhängiges Epsilon

Returns: *None*, denn es wird ein Plot ausgegeben.

hilfsfunktionen_plot.main()

Demonstriert die Nutzung der Funktionenplots.

Parameters: *None*

Returns: *None*, denn es werden nur Plots ausgegeben.

8.2.3 hilfsmittel.py

Dieses Modul enthält allgemeine Funktionen, die genutzt werden können.

read_number()

Überprüfung der Nutzereingaber auf ihre Gültigkeit.

Parameters:

question (*str*) : Frage, die der Nutzer beantworten soll
lower_limit (*float*) untere Grenze, Standardwert: *-math.inf*
upper_limit (*float*) obere Grenze, Standardwert: *math.inf*
data_type (*type*) : Eingabe des Nutzers soll als dieser Datentyp eingelesen werden.

Returns:

(*data_type*) nutzereingabe als eingegebene Zahl wird im Datentyp *data_type* zurückgegeben

zahlenliste()

Erstellt eine Zahlenliste.

Paramters: *None*

Returns:

(list) list als erstellte Zahlenliste

save_plot()

Speichert den aktuellen Plot unter einem angegebenen Dateinamen.

Parameters:

filename (str) Dateiname, unter dem der Plot gespeichert wird

Returns: *None*

measure_time()

Misst die durchschnittliche Ausführungszeit einer Funktion.

Parameters:

func (function) : Funktion, deren Ausführungszeit gemessen werden soll

**args (tuple)* : Argumente, die an die Funktion übergeben werden soll

num_repeats (int, optional) Anzahl der Wiederholungen der Funktion, Standardwert
= 5

***kwargs (dict)* zusätzlich benannte Argumente für die Funktion

Returns:

(float) durchschnittliche Ausführungszeit in Millisekunden

hilfsmittel.main()

Demonstriert die Nutzung der Hilfsfunktionen.

Parameters: *None*

Returns: *None*, denn es werden nur Werte in der Konsole ausgegeben.

8.2.4 linear_solvers.py

Dieses Modul besteht aus zwei Funktionen zum Lösen des linearen Gleichungssystem $Ax = b$.

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 $Ax = b$

`solve_sor()`

Löst das lineare Gleichungssystem $Ax = b$ mittels des SOR-Verfahrens.

Parameters:

- `A (numpy.ndarray)` : Die Matrix A des linearen Gleichungssystems
- `b (numpy.ndarray)` : Vektor der rechten Seite des Gleichungssystems
- `x0 (numpy.ndarray)` : Ausgewählter Startvektor bzw. geratene Lösung
- `params (dict, optional)` : *dictionary* enthält Informationen der Abbruchbedingungen, wobei
 - `'eps' (float)`: Toleranz für den Residualfehler, der in der Unendlichen Norm (maximale absolute Differenz) gemessen wird. Wenn der Residualfehler kleiner als `eps` wird, bricht das Verfahren ab.
 - `'max_iter' (int)`: Maximale Anzahl der Iterationen. Wenn diese Zahl erreicht wird, bricht das Verfahren ab.
 - `var_x (float)`: Minimale Änderung des Lösungsvektors `x`, wenn der Unterschied zwischen aufeinander folgenden Iterationen kleiner als `var_x` ist, bricht das Verfahren ab.
- `omega (float, optional)` : Relaxationsparamter ω

Returns:

- `(str)` Grund für den Abbruch des Verfahrens
- `(list of numpy.ndarray)` Liste der Vektoren, die in jeder Iteration berechnet wurden
- `(list of float)` eine Liste der Residuen

`linear_solvers.main()`

Demonstriert die Nutzung der Funktionen für die LU-Zerlegung und des SOR-Verfahrens

Parameters: `None`

Returns: `None`, denn diese Funktion gibt Werte im Konsole für ein Beispiel aus.

8.2.5 `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.

8.2.6 experiments_it.py

Dieses Modul stellt Werkzeuge zur Verfügung, um den maximalen Fehler des SOR-Verfahrens und der LU-Zerlegung zu untersuchen.

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 $f(x_1, x_2)$, das den Term darstellt.

8.2.7 experiments_it.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.