Lösung linearer Gleichungssysteme mit iterativen Methoden

vorgelegt von

Sascha Richter
Matrikelnummer: 5001431
Seminargruppe: CS18-2

Leipzig der

29. September 2020

Prüfer: Prof. Dr. Holger Perlt

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

Die vorliegende Ausarbeitung dient der wissenschaftlichen Betrachtung dreier Verfahren zur iterativen Lösung linearer Gleichungssystemen. Dies sind die Lösung mithilfe des Jacobi-, Gauß-Seidel- und Successive Overrelaxation-Verfahrens. Im restlichen Teil der Arbeit wird die Kurzbezeichnung SOR für das Successive Overrelaxation-Verfahren verwendet.

Im theoretischen Teil der Arbeit werden die Verfahren grundsätzlich beschrieben und die wesentlichen Eigenschaften erläutert. Es wird untersucht, welche Bedingungen das lineare Gleichungssystem erfüllen muss, damit die Iteration konvergiert. Nur wenn ein iteratives Verfahren gegen einen Wert konvergiert, kann eine Lösung nach endlicher Zeit gefunden werden.

Im praktischen Teil wird die Problemstellung sowie die Implementierung der Methoden in der Programmiersprache Python beschrieben. Die mit Kommentaren versehenen Quelltextdokumente befinden sich im Anhang dieser Arbeit. Im Anschluss werden die Methoden bezüglich der Konvergenz gegenübergestellt. Abgeschlossen wird die Ausarbeitung mit einem Fazit zu den Methoden sowie einer kurzen Zusammenfassung.

Kapitel 2

Theorie

Ein lineares Gleichungssystem (abgekürzt LGS) ist eine Menge linearer Gleichungen mit einer oder mehreren Unbekannten, die alle gleichzeitig erfüllt sein sollen. Die Problemstellung der linearen Gleichungssysteme ist ein Teilgebiet der linearen Algebra und im alltäglichen Leben allgegenwärtig. Mögliche Anwendungsbereiche befinden sich im medizinischen, marktwirtschaftlichen und wissenschaftlichen Sektor.

$$\begin{aligned} 3x + 2y &= 1 \\ 2x - 2y + 4z &= -2 \\ -x + 1/2y - z &= 0 \end{aligned} \tag{2.1}$$

In 2.1 ist ein einfaches Beispiel für ein LGS dargestellt. Die Lösung des LGS erfordert das Finden der Variablen x, y, z, sodass alle Gleichungen wahr sind. Für das genannte Beispiel ist die Lösung wie folgt:

$$\begin{aligned} x &= 1 \\ y &= -2 \\ z &= -2 \end{aligned} \tag{2.2}$$

Mathematisch gibt es Methoden, die bei Systemen bis zu einer gewissen Größe eine Lösung in akzeptabler Zeit direkt finden. Für sehr große Gleichungssysteme, die über eine Größe von 1000 unbekannten hinausgehen, sind diese Verfahren nur schwer zu bewältigen. Der Rechenaufwand wird zu groß und es wird zuviel Speicherplatz benötigt. Aus diesem Grund werden Iterationsverfahren angewendet, diese benötigen weniger Speicher, liefern aber nur ein angenähertes Ergebnis. Deshalb wird die Konvergenz, also die Abweichung zwischen aktuellem Ergebnis und vorherigem Ergebnis, betrachtet
Im Folgenden werden drei dieser iterativen Methoden, das Jacobi-Verfahren, das Gauß-Seidel-

Verfahren und die Succesive Over Relaxation Methode, näher betrachtet.

2.1 Jacobi-Verfahren

Das Jacobi-Verfahren, benannt nach Carl Gustav Jacob Jacobi, ist ein Algorithmus zur näherungsweisen Lösung von linearen Gleichungssystemen. Die Problemstellung ist, wie im Beispiel 2.1, ein LGS der Form Ax = b. Die Matrix A wird nun in eine Diagonalmatrix D, eine untere Dreiecksmatrix L und eine obere Dreickesmatrix U zerlegt.

$$A = L + D + U \tag{2.3}$$

Anhand dieser Werte kann die Jacobi-Matrix J gebildet werden:

$$J = -D^{-1} * (L + R) \tag{2.4}$$

Für x kann ein beliebiger Vektor als Startpunkt für die Iteration gewählt werden. Idealerweise wird ein Vektor gewählt, der nahe an der Lösung liegt. Die Iteration erfolgt nun durch das berechnen neuer Werte für den Vektor x [1].

$$x^t = J * x^{t-1} + D^{-1}b \tag{2.5}$$

Als Abbruchkriterium des Verfahrens kann die Konvergenz des x-Vektors betrachtet werden. Sobald die absolute Differenz der x-Vektoren der letzten Iterationsschritte einen gewissen Wert unterschreitet, wird das Verfahren beendet. Dieser Wert wird häufig als Toleranz bezeichnet.

2.2 Gauß-Seidel-Verfahren

Das Gauß-Seidel-Verfahren, benannt nach Carl Friedrich Gaus und Ludwig Seidel, verhält sehr ähnlich zum Jacobi-Verfahren. Es erfolgt wie in der Gleichung 2.3 eine Zerlegung der Matrix. Es wird nun die Gauß-Seidel-Matrix H gebildet und anhand dessen die Iteration durchgeführt.

$$H = -(D + X)^{-1}R \tag{2.6}$$

$$x^t = Hx^{t-1} + (D + L)^{-1}b \tag{2.7}$$

Beide Verfahren verwenden eine Iterationsmatrix und iterieren Anhand eines Vektors x. Damit diese Verfahren eine Lösung liefern, ist es notwendig, dass die Vektoren x linear konvergent

sind. Dies ist der Fall, wenn der Spektralradius der Iterationsmatrix kleiner 1 ist. Je geringer der Spektralradius, desto schneller konvergiert das Verfahren [2].

2.3 SOR

In der Praxis sind die Matrizen groß und dünn besetzt. Daraus resultiert, dass der Spektralradius gegen 1 geht. Wie bereits zum Ende des Kapitels 2.1 dargelegt, konvergieren die behandelten Verfahren somit nur langsam. Es wird nun versucht, die Konvergenz durch Relaxationsparameter $0 < \omega < 2$ zu beschleunigen. Wählt man $\omega = 1$, erhält man das Gauß-Seidel-Verfahren. Die Iterationsmatrix wird nun aus der Beziehung von ω und der zerlegten Matrix A gebildet und folgendermaßen iteriert:

$$H_\omega = (D + \omega L)^{-1}((1 - \omega)D - \omega R) \tag{2.8}$$

$$x^t = H\omega x^{t-1} + \omega(D + \omega L)^{-1}b \tag{2.9}$$

Ein ideales ω lässt sich durch einen Zusammenhang zwischen den Eigenwerten der Matrix $H\omega$ sowie den Eigenwerten der Jacobi-Matrix J herleiten. Der Spektralradius der Matrix J wird als ρ bezeichnet. Gilt für $\rho_J < 1$, dann gilt für den Spektralradius der SOR-Matrix:

$$spr(H_\omega) = \begin{cases} \omega - 1 & \omega \leq \omega_{opt}, \\ \frac{1}{4}(\rho_j^2\omega^2 + \sqrt{\rho_j^2\omega^2 - 4(\omega - 1)})^2 & \omega \geq \omega_{opt} \end{cases} \tag{2.10}$$

Der optimale Parameter ω_{opt} kann als Schnittpunkt der beiden Funktionen gefunden werden. Daraus resultiert folgende Berechnung [3]:

$$\omega_{opt} = \frac{2(1 - \sqrt{1 - \rho_j^2})}{\rho_j^2} \tag{2.11}$$

Kapitel 3

Praxis und Lösungsentwicklung

Nach der theoretischen Beschreibung der Methoden werden im Folgenden die praktischen Aspekte der Verfahren thematisiert. Die Ausarbeitung erfolgt anhand der Aufgabenstellung. Die Methoden wurden mit der Programmiersprache Python realisiert. Für alle Verfahren wurde als Abbruchkriterium eine Toleranz von $|x^k - x^{k-1}| \leq 10^{-3}$ definiert.
Zusätzlich soll die Konvergenz der Iteration betrachtet werden. Die Umsetzung, wie die Verfolgung und das Plotten der Konvergenz in Python realisiert wurde, wird im Umfang der Projektarbeit nicht näher betrachtet, kann aber im Anhang unter Quellcode nachgeschlagen werden.

3.1 Lösungsentwicklung

Zur Realisierung der Aufgabe wurden die Python Pakete math, numpy und matplotlib importiert. Math dient dem Lösen von Wurzln, numpy dem verwenden von Matritzenoperationen und matplotlib zum plotten der Konvergenz. Alle Verfahren wurden im gleichen Python Script realisiert und als Funktionen angelegt. Da für alle Verfahren die gleiche Matrix verwendet werden soll, wird diese importiert und entsprechend in Diagonal-, Links- und Rechtsmatrix zerlegt. Zusätzlich wird der b Vektor importiert und ein x Vektor als Startpunkt der Iteration gewählt. Für alle Verfahren wurde ein Startvektor mit ausschließlich Nullen gewählt. Die Berechnungen wurden gemäß der theoretischen Grundlagen aus dem ersten Teil der Projektarbeit umgesetzt. Die verwendeten Numpy Operationen sind als folgende Matritxoperationen zu verstehen:

```
1 np.dot() => Berechnung des Skalarprodukts. Wird haeufig auch als Inneres
     Produkt oder Punktprodukt bezeichnet.
2 np.linalg.inv() => Invertierung der Matrix
3 np.linalg.norm(, ) => Bildung von Matrixnorm. np.linalg.norm(A,2) bildet
     den Spektralradius der Matrix A.
```

Alle weiteren verwendeten Funktionen sind selbsterklärend. Die elementaren Teile der Funktionen, inklusive Lösung, sind die folgenden:

```
#Jacobi Matrix
Jm = np.dot(-np.linalg.inv(Dm), (Lm + Rm))
#Iteration
c = np.dot(np.linalg.inv(Dm), b)
x = np.dot(Jm, x0) + c

#Ausgabe
Jacobi Verfahren
Loesung gefunden nach: 9 Iterationen
x1 = 0.9996741452148709
x2 = 2.0004476715450097
x3 = -1.0003691576845715
x4 = 1.0006191901399695

# Berechnung Gauss-Seidel-Matrix
gsM = np.dot(-np.linalg.inv((Dm + Lm)), Rm)
#Iteration
x = np.dot(gsM, x0) + np.dot(np.linalg.inv(Dm + Lm), b)

#Ausgabe
Gauss-Seidel Verfahren
Loesung gefunden nach: 4 Iterationen
x1 = 1.0008609786250942
x2 = 2.000298250656547
x3 = -1.0003072761017007
x4 = 0.9998497464910825
```

Die Ausarbeitung des SOR-Verfahrens besteht aus zwei Funktionen; die Hauptfunktion zur Iteration des Fixpunktes und einer Unterfunktion zur Berechnung des idealen ω.

```
# Berechnung p aus Jacobi-Matrix
Jm = np.dot(-np.linalg.inv(Dm), (Lm + Rm))
#Berechnung des Spektralradius
p = np.linalg.norm(Jm, 2)
#Berechnung omega
omega = (2*(1 - math.sqrt(1-p**2))) / p**2
#Berechnung SOR-Matrix hw
Hw = np.dot(np.linalg.inv((Dm + (omega * Lm))), ((1 - omega) * Dm - omega
    * Rm))
#Iteration
x = np.dot(Hw, x0) + np.dot(omega * (np.linalg.inv(Dm + (omega * Lm))), b)

#Ausgabe
SOR-Verfahren
```

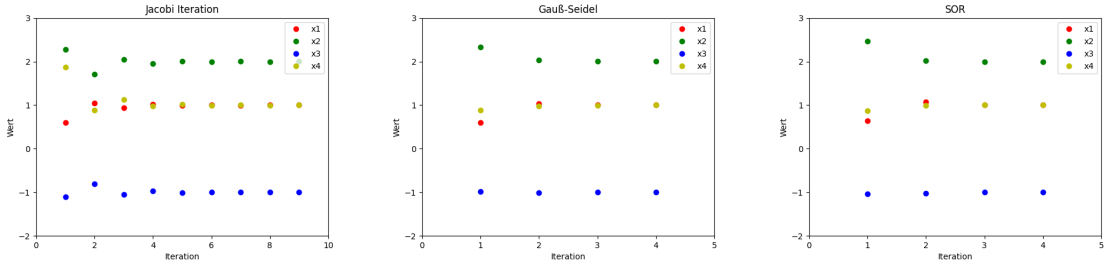
```
Spektralradius der Jacobi-Matrix: 0.4520573859824952
Ideales Omega berechnet: 1.0570886785782487
Loesungswerte
x1 = 0.999478360339346
x2 = 1.9997939409736603
x3 = -0.9998102250796905
x4 = 1.0000388396361288
Loesung gefunden nach Iterationen: 4
```

3.2 Auswertung

Alle Verfahren haben für die gegebene Aufgabe in angemessener Zeit ein Ergebnis geliefert. In Anbetracht der Resultate, kann davon ausgegangen werden, dass die exakte Lösung $x1 = 1, x2 = 2, x3 = -1, x4 = 1$ ist. Werden die Werte zum Überprüfen in das x der Ausgangsgleichung $Ax = b$ eingesetzt, wird dies bestätigt.

$$\begin{aligned} 10 - 2 - 2 &= 6 \\ -1 + 22 + 1 + 3 &= 25 \\ 2 - 2 - 10 - 1 &= -11 \\ 6 + 1 + 8 &= 15 \end{aligned} \tag{3.1}$$

Um die Konvergenz der Verfahren zu vergleichen, wurden die Werte der Iterationen abgespeichert und als Plot visualisiert.



(a) Jacobi-Verfahren **(b)** Gauß-Seidel-Verfahren **(c)** SOR-Verfahren

Abbildung 3.1: Konvergenz der Verfahren

Gemäß Abbildung 3.1 zeigt sich, dass das Gauß-Seidel- und SOR-Verfahren deutlich schneller konvergieren als das Jacobi Verfahren. Aufgrund der theoretischen Grundlage des SOR-Verfahrens besteht die Vermutung, dass das SOR-Verfahren schneller als das Gauß-seidel Verfahren konvergieren sollte. Betrachtet man jedoch den Spektralradius der Jacobi-Matrix liegt dieser bei ungefähr 0,45. Eine schnellere Konvergenz würde vorliegen, wenn dieser Wert sich 1 annähert. Zusätzlich wurde als $\omega = 1.0570886785782487$ berechnet. Die theoretische

Grundlage besagt, dass $\omega = 1$ exakt das Gauß-Seidel-Verfahren ist. Somit ist die Ähnlichkeit der Konvergenz zwischen Gauß-Seidel- und SOR-Verfahren offensichtlich. Sobald man eine größere, dünn besetzte Matrix verwendet und der Spektralradius gegen 1 strebt, erweist sich das SOR-Verfahren als effizienter.

Kapitel 4

Zusammenfassung

In der vorliegenden Arbeit wurden das Jacobi-Verfahren, Gauß-Seidel-Verfahren und SOR-Verfahren zum iterativen Lösen linearer Gleichungssysteme theoretisch betrachtet und praktisch angewendet.
Alle drei Methoden wurden anhand der theoretischen Grundlagen untersucht und erläutert. Als Konvergenzbedingung wurde der Spektralradius gegeben, sobald sich der Wert der 1 annähert, sollte das SOR-Verfahren verwendet werden. Wenn die 1 überschritten wird, konvergiert keines der aufgeführten Verfahren.
Alle drei Verfahren wurden in Python praktisch umgesetzt und die elementaren Punkte des Quelltextes in der Projektarbeit aufgeführt und erläutert.
Der optimale Overrelaxationsparameter, $\omega_{opt} = 1.0570886785782487$, wurde anhand der Formel 2.11 berechnet.
Bei einem direkten Vergleich des Konvergenzverhaltens hat sich gezeigt, dass Gauß-Seidel und SOR besser konvergieren als Jacobi. Gauß-Seidel und SOR verhalten sich für dieses spezifische Beispiel fast identisch. Das bestätigt die theoretischen Grundlagen, dass das SOR-Verfahren bei einem Spektralradius nahe 1 besser konvergiert (in diesem Beispiel bei ca. 0,5) und bei einem $\omega = 1$ mit dem Gauß-Seidel-Verfahren übereinstimmt.
In Anbetracht der gelieferten Ergebnisse der Verfahren wurde folgendes Ergebnis vermutet:

$$x1 = 1$$
$$x2 = 2$$
$$x3 = -1$$
$$x4 = 1$$

Dies hat sich, durch das Einsetzen in die ursprüngliche Formel, als richtig erwiesen.

Literaturverzeichnis

[1] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[2] A. Meister, *Numerik linearer gleichungssysteme*. Springer, 2011, vol. 5.

[3] T. Richter and T. Wick, *Einführung in die Numerische Mathematik: Begriffe, Konzepte und zahlreiche Anwendungsbeispiele*. Springer-Verlag, 2017.

Anhang

Quellcode

```
import math

import matplotlib.pyplot as plt
import numpy as np

# Zu loesendes LGS aus Aufgabe
A = np.array([[10, -1, 2, 0],
              [-1, 11, -1, 3],
              [2, -1, 10, -1],
              [0, 3, -1, 8]])

# Hilfsmatritzen zur Berechnung der Dreicksmatritzen
L = np.array([[0, 0, 0, 0],
              [1, 0, 0, 0],
              [1, 1, 0, 0],
              [1, 1, 1, 0]])

R = np.array([[0, 1, 1, 1],
              [0, 0, 1, 1],
              [0, 0, 0, 1],
              [0, 0, 0, 0]])

# Berechnung Diagonalmatrix
Dm = np.diag(np.diag(A, 0))

# Dreiecksmatrix Links und Rechts
Lm = A * L
Rm = A * R

# Loesung aus Aufgabe
b = np.array([6, 25, -11, 15])

# Anfangspunkt fuer Iteration
x = np.array([0, 0, 0, 0])
```

```
35
36 # Toleranz aus Aufgabenstellung
37 tol = 0.001
38
39
40 def jacobi(A, b, x, Dm, Lm, Rm, tol, maxiter=200):
41     iteration = 1
42     x0 = x.copy()
43     # Berechnung Jacobimatrix
44     Jm = np.dot(-np.linalg.inv(Dm), (Lm + Rm))
45
46     yIterat = []
47     x1lJ = []
48     x2lJ = []
49     x3lJ = []
50     x4lJ = []
51
52     while iteration <= maxiter:
53         # Berechnung Fixpunkt
54         c = np.dot(np.linalg.inv(Dm), b)
55         x = np.dot(Jm, x0) + c
56         # Vorbereitung zur Untersuchung ob Abbruchkriterium erreicht
57         var = abs(x0 - x)
58         # Pruefung ob Toleranzgrenze erreicht ist
59         if var[0] < tol and var[1] < tol and var[2] < tol and var[3] < tol
    :
60             break;
61         # Wert vorgehenden Iteration sichern
62         x0 = x.copy()
63         # Erzeugen einer Liste zum Plotten des Konvergenzverhaltens
64         x1lJ.append(x0[0])
65         x2lJ.append(x0[1])
66         x3lJ.append(x0[2])
67         x4lJ.append(x0[3])
68         yIterat.append(iteration)
69         iteration = iteration + 1
70     iteration = iteration -1
71     print("Jacobi Verfahren")
72     print("Loesung gefunden nach: " + str(iteration) + " Iterationen")
73
74     print("x1 = " + str(x0[0]))
75     print("x2 = " + str(x0[1]))
76     print("x3 = " + str(x0[2]))
77     print("x4 = " + str(x0[3]))
78
```

```
    # Plotten der Konvergenz
    p1 = plt.plot([yIterat], [x1lJ], "ro")
    p2 = plt.plot([yIterat], [x2lJ], "go")
    p3 = plt.plot([yIterat], [x3lJ], "bo")
    p4 = plt.plot([yIterat], [x4lJ], "yo")

    plt.axis([0, 10, -2, 3])
    plt.title('Jacobi Iteration')
    plt.xlabel('Iteration')
    plt.ylabel('Wert')
    plt.legend((p1[0], p2[0], p3[0], p4[0]), ('x1', 'x2', 'x3', 'x4'))
    plt.show()


def gauss_seidel(A, b, x, Dm, Lm, Rm, tol, maxiter=200):
    iteration = 1
    yIterat = []
    x1lgs = []
    x2lgs = []
    x3lgs = []
    x4lgs = []
    x0 = x.copy()
    # Berechnung Gauss-Seidel-Matrix h1
    gsM = np.dot(-np.linalg.inv((Dm + Lm)), Rm)
    while iteration <= maxiter:

        # Berechnung Iterationsmatrix
        x = np.dot(gsM, x0) + np.dot(np.linalg.inv(Dm + Lm), b)
        # Vorbereitung zur Untersuchung ob Abbruchkriterium erreicht
        var = abs(x0 - x)
        # Pruefung ob Toleranzgrenze erreicht ist
        if var[0] < tol and var[1] < tol and var[2] < tol and var[3] < tol
    :
            break;
        # Sichern vorheriger Iterationsschritt
        x0 = x.copy()
        # Erzeugen einer Liste zum Plotten des Konvergenzverhaltens
        x1lgs.append(x0[0])
        x2lgs.append(x0[1])
        x3lgs.append(x0[2])
        x4lgs.append(x0[3])
        yIterat.append(iteration)
        iteration = iteration + 1
    iteration = iteration - 1
    print("Gauss-Seidel Verfahren")
```

```
123     print("Loesung gefunden nach: " + str(iteration) + " Iterationen")
124 
125     print("x1 = " + str(x0[0]))
126     print("x2 = " + str(x0[1]))
127     print("x3 = " + str(x0[2]))
128     print("x4 = " + str(x0[3]))
129 
130     # Plotten der Konvergenz
131     p1 = plt.plot([yIterat], [x1lgs], "ro")
132     p2 = plt.plot([yIterat], [x2lgs], "go")
133     p3 = plt.plot([yIterat], [x3lgs], "bo")
134     p4 = plt.plot([yIterat], [x4lgs], "yo")
135 
136     plt.axis([0, 5, -2, 3])
137     plt.title('Gauss-Seidel')
138     plt.xlabel('Iteration')
139     plt.ylabel('Wert')
140     plt.legend((p1[0], p2[0], p3[0], p4[0]), ('x1', 'x2', 'x3', 'x4'))
141     plt.show()
142 
143 
144 def SOR(A, b, x, Dm, Lm, Rm, tol, maxiter=100):
145     iteration = 1
146     yIterat = []
147     x1lgs = []
148     x2lgs = []
149     x3lgs = []
150     x4lgs = []
151     x0 = x.copy()
152     # Berechnung p aus Jacobi-Matrix
153     Jm = np.dot(-np.linalg.inv(Dm), (Lm + Rm))
154     p = np.linalg.norm(Jm, 2)
155     print("Spektralradius der Jacobi-Matrix: " +str(p))
156     omega = omega_opt(p)
157     print("Ideales Omega berechnet: " + str(omega))
158     #Berechnung SOR-Matrix hw
159     Hw = np.dot(np.linalg.inv((Dm + (omega * Lm))), ((1 - omega) * Dm -
    omega * Rm))
160     while iteration <= maxiter:
161         # Berechnung Iterationsmatrix
162         x = np.dot(Hw, x0) + np.dot(omega * (np.linalg.inv(Dm + (omega *
    Lm))), b)
163         # Vorbereitung zur Untersuchung ob Abbruchkriterium erreicht
164         var = abs(x0 - x)
165         # Pruefung ob Toleranzgrenze erreicht ist
```

```
166         if var[0] < tol and var[1] < tol and var[2] < tol and var[3] < tol
    :
167             break;
168 
169         x0 = x.copy()
170         # Erzeugen einer Liste zum Plotten des Konvergenzverhaltens
171         x1lgs.append(x0[0])
172         x2lgs.append(x0[1])
173         x3lgs.append(x0[2])
174         x4lgs.append(x0[3])
175         yIterat.append(iteration)
176         iteration = iteration + 1
177     iteration = iteration - 1
178     print("Loesungswerte")
179     print("x1 = " + str(x0[0]))
180     print("x2 = " + str(x0[1]))
181     print("x3 = " + str(x0[2]))
182     print("x4 = " + str(x0[3]))
183     print("Loesung gefunden nach Iterationen: " + str(iteration))
184     # Plotten der Konvergenz
185     p1 = plt.plot([yIterat], [x1lgs], "ro")
186     p2 = plt.plot([yIterat], [x2lgs], "go")
187     p3 = plt.plot([yIterat], [x3lgs], "bo")
188     p4 = plt.plot([yIterat], [x4lgs], "yo")
189 
190     plt.axis([0, 5, -2, 3])
191     plt.title('SOR')
192     plt.xlabel('Iteration')
193     plt.ylabel('Wert')
194     plt.legend((p1[0], p2[0], p3[0], p4[0]), ('x1', 'x2', 'x3', 'x4'))
195     plt.show()
196 
197 
198 def omega_opt(p):
199     # result = 2 / (1 + math.sqrt(1 - p**2))
200     #result = 1 + (p / (1 + math.sqrt(1 - p ** 2))) ** 2
201     result = (2*(1 - math.sqrt(1-p**2))) / p**2
202     return result
203 
204 print("Willkommen zum Numerik Script zum interativen L s e n von LGS")
205 print("Folgende Eingaben sind moeglich:")
206 print("1 f r Jacobi")
207 print("2 f r Gauss-Seidel")
208 print("3 f r SOR")
209 
```

```
210 userInput = input("Eingabe int 1-3: ")
211
212 if userInput == '1':
213     jacobi(A, b, x, Dm, Lm, Rm, tol)
214
215 if userInput == '2':
216     gauss_seidel(A, b, x, Dm, Lm, Rm, tol)
217
218 if userInput == '3':
219     SOR(A, b, x, Dm, Lm, Rm, tol)
```

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder veröffentlicht noch einer anderen Prüfungsbehörde vorgelegt.

Leipzig, den 29. September 2020

Sascha Richter