Teil XIV

Lösung linearer Gleichungssysteme

Gauss Algorithmus

- Zwei Schritte: Vorwärtselimination und Rückwärtssubstitution

Vorwärtselimination

- Erzeugen einer Stufenform
- Zeilen dürfen mit Konstanten multipliziert werden
- Zeilen dürfen addiert werden werden

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \quad \begin{matrix} \\ \text{- 1. Zeile} \cdot \frac{a_{21}}{a_{11}} \\ \text{- 1. Zeile} \cdot \frac{a_{31}}{a_{11}} \end{matrix}$$

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & \tilde{a}_{22} & \tilde{a}_{23} \\ 0 & \tilde{a}_{32} & \tilde{a}_{33} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ \tilde{b}_2 \\ \tilde{b}_3 \end{pmatrix} \quad \begin{matrix} \\ \\ \text{- 2. Zeile} \cdot \frac{\tilde{a}_{32}}{\tilde{a}_{22}} \end{matrix}$$

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & \tilde{a}_{22} & \tilde{a}_{23} \\ 0 & 0 & \bar{a}_{33} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ \tilde{b}_2 \\ \bar{b}_3 \end{pmatrix} \quad \begin{matrix} \\ \\ \text{- 2. Zeile} \cdot \frac{\tilde{a}_{32}}{\tilde{a}_{22}} \end{matrix}$$

Rückwärtssubstitution

- LGS muss in Stufenform (obere Dreiechsmatrix) vorliegen
- Die unterste Gleichung hat nur eine Variable
- Deren Wert kann direkt berechnet werden
- Zweitunterste Zeile hat zwei Variablen, eine davon unbekannt
- ...
- nte Zeile von unten hat n Variablen, eine unbekannt

$$x_3 = \bar{b}_3 / \bar{a}_{33}$$

$$x_2 = \tilde{b}_2 - \tilde{a}_{23} \cdot x_3 / \tilde{a}_{22}$$

$$x_1 = \tilde{b}_1 - a_{12} \cdot x_2 - a_{13} \cdot x_3 / a_{11}$$

- Zeile in einer größeren Matrix:

$$x_i = (b_i - \sum_{j=i+1}^{n} a_{ij} \cdot x_j) / a_{ii}$$

Beispiel

$$\begin{pmatrix} 2 & 3 & 1 \\ 4 & 7 & 3 \\ 2 & 4 & 4 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 3 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 3 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 3 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

$$\begin{aligned} x_3 &= 1/2 = 0.5 \\ x_2 &= (0 - 1 \cdot 0.5)/1 = -0.5 \\ x_1 &= (1 - 3 \cdot (-0.5) - 1 \cdot 0.5)/2 = 1.0 \end{aligned}$$

Vorwärtselimination in python

```
def elimination(a,b)
  n = a.shape[0]
  for i in range(n):
    for j in range(i+1,n):
      faktor = a[j,i]/a[i,i]
      b[j] -= b[i]*faktor
      for k in range(i,n):
        a[j,k] -= a[i,k]*faktor
```

- 1. Schleife: Über alle Zeilen/Diagonalelemente
- 2. Schleife: Über alle Zeilen unterhalb des Diagonalelements. Elemente unterhalb des Diagonalelements müssen Null werden
- 3. Schleife: Über alle Elemente der Zeile aus der 2. Schleife Vielfaches der Diagonalelementszeile zur aktuellen Zeile addieren
- `faktor` so gewählt, dass Element unterhalb des Diagonalelements zu Null wird

Rückwärtssubstitution in python

```python
def substitution(a,b):
  n = a.shape[0]
  x = empty(n)
  for i in range(n-1, -1, -1):
    zaehler = b[i]
    for j in range(i+1,n):
      zaehler -= a[i,j] * x[j]
    x[i] = zaehler / a[i,i]
  return x
```

- 1. Schleife: Zeilen von unten nach oben durchlaufen
- 2. Schleife: Bekannte Variablen auf die rechte Seite
- Zum Schluß teilen durch den Koeffizienten des aktuellen x-Werts
- Lösungsvektor `x` zurückgeben

Lösung des vorigen Beispiels

```
>>> from lgs import * # enthaelt beide funktionen
                      # + numpy-import
>>> a = array([[2,3,1],[4,7,3],[2,4,4]], dtype=float)
>>> b = array([1,2,2], dtype = float)
>>> elimination(a,b); a; b
array([[ 2.,  3.,  1.],
       [ 0.,  1.,  1.],
       [ 0.,  0.,  2.]])
array([ 1.,  0.,  1.])
>>> substitution(a,b)
array([ 1. , -0.5,  0.5])
```

Lösung der Wärmeleitungsgleichung 1D

- Stab mit neun Diskretisierungspunkten
- Links beheizt: $T = 0$, rechter Rand: $T = 0$

```
>>> def getwaermelgs1D():
>>>   a = array([[-2, 1, 0, 0, 0, 0, 0, 0, 0],
>>>              [ 1,-2, 1, 0, 0, 0, 0, 0, 0],
...
>>>              [ 0, 0, 0, 0, 0, 0, 0, 1,-2]],
>>>             dtype = float)
>>>   b = array([-1, 0, 0, 0, 0, 0, 0, 0, 0],
>>>             dtype = float)
>>>   return(a,b)
>>>
>>> (a,b) = getwaermelgs1D()
>>> loese(a,b)
array([0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1])
```

Lösung der Wärmeleitungsgleichung 2D

- LGS aus der vorigen Vorlesung
- Diskretisierung des Raums mit 3x3 Punkten
- Unten beheizt: $T = 1$, restlicher Rand: $T = 0$

```
>>> def getwaermelgs2D():
>>>   a = array([[-4, 1, 0, 1, 0, 0, 0, 0, 0],
>>>              [ 1,-4, 1, 0, 1, 0, 0, 0, 0],
...
>>>              [ 0, 0, 0, 0, 0, 1, 0, 1,-4]],
>>>             dtype = float)
>>>   b = array([-1,-1,-1, 0, 0, 0, 0, 0, 0],
>>>             dtype = float)
>>>   return (a,b)
>>>
>>> (a,b) = getwaermelgs2D()
>>> loese(a,b)
array([0.43, 0.53, 0.43, 0.19, 0.25, 0.19, 0.07,  0.10,
0.07])
```

Kosten des Gauss-Algorithmus

Vorwärtselimination

- n sei die Anzahl an Unbekannten
- Drei verschachtelte Schleifen
- Jeweils $\mathcal{O}(n)$ Schleifendurchläufe
- Insgesamt Kosten von $\mathcal{O}(n^3)$

Rückwärtssubstitution

- Zwei verschachtelte Schleifen (Je $\mathcal{O}(n)$)
- Insgesamt Kosten von $\mathcal{O}(n^2)$

Optimierung

- Matrix ist dünn besetzt, viele der Operationen unnötig
- 1D: Matrix tridiagonal
  - Beide inneren Schleifen nur über zwei Elemente
  - $\Rightarrow$ Gesamtkosten $\mathcal{O}(n)$
- 2D: Matrix hat zwei weitere Bänder mit Abstand $\sqrt{n}$
  - Inneren Schleifen über je $\sqrt{n}$ Elemente
  - $\Rightarrow$ Gesamtkosten $\mathcal{O}(n \cdot \sqrt{n} \cdot \sqrt{n}) = \mathcal{O}(n^2)$

Iterative Verfahren

Motivation

- $\mathcal{O}(n^3)$ ist viel zu teuer.
- Optimiert immer noch $\mathcal{O}(n^2)$ (2D) bzw. $\mathcal{O}(n^{2,33})$ (3D)
- Kosten bei 3D-Simulation:

	Mehrgitter	Jacobi	Gauss opt.	Gauss
N Punkte p. Dim.	$n = N^3$	N^5	$n^{2,33} = N^7$	$n^3 = N^9$
10	1.000	$1e5$	$1e7$	$1e9$
20	8.000	$3e6$	$1e9$	$5e11$
50	125.000	$3e8$	$8e11$	$2e15$
100	$1e6$	$1e10$	$1e14$	$1e18$

- Idee: Schrittweise Verbesserung einer Näherungslösung x_i
- $x^{(k+1)} = \Phi(x^{(k)})$
- Bei Konvergenz gibt es einen Fixpunkt $\Phi(x) = x$
- Ein Schritt des Verfahrens kostet $\mathcal{O}(n)$
- Anzahl Schritte möglichst klein, im Ideallfall $\mathcal{O}(1)$

Typen von Iterationsverfahren

Relaxationsverfahren

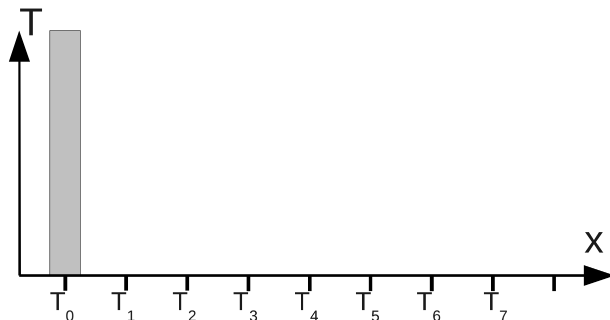
- Ausgangspunkt LGS $Ax = b$
- Ziel: Fehler $e^{(k)} = x^{(k)} - x$ minimieren
- Fehler ist leider nicht messbar, da x unbekannt
- Residuum $r^{(k)} = b - Ax^{(k)} = Ae^{(k)}$ als Richtung des Fehlers
- Neues x unter Verwendung des Residuums
- Z.B. bei Jacobi: Residuum lokal immer auf Null setzen

Minimierungsverfahren

- Ausgang wieder LGS $Ax = b$, A symmetrisch, positiv definit
- Lösung äquivalent zur Minimierung von $f(x) := \frac{1}{2}x^T Ax - b^T x + c$
- $f'(x) := \frac{1}{2}A^T x + \frac{1}{2}Ax - b = Ax - b = -r(x)$
- Minimum von f (Nullstelle von f') ist Lösung des LGS
- Jeder Iterationsschritt läuft einen Schritt (z.B. steilster Abstieg) auf das Minimum zu

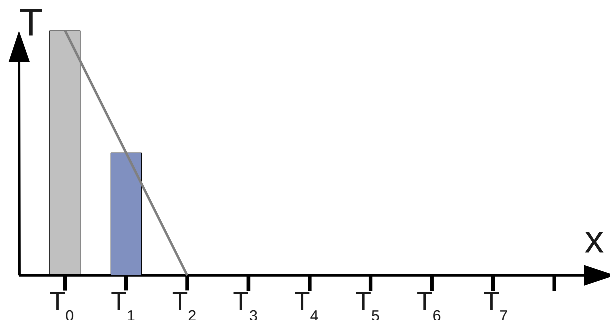
Jacobi-Verfahren (aus PDE-Sicht)

- Ist ein Relaxationsverfahren
- An jedem Punkt wird wird die PDE lokal erfüllt
- Neue Werte $T^{(k+1)}$ aus alten Werte $T^{(k)}$ berechnen
- 1D Wärmeleitung stationär: Zweite Ableigung muß Null sein
- $\Rightarrow T_{i+1} - 2T_i + T_{i-1} = 0 \Rightarrow T_i = \frac{1}{2}(T_{i+1} + T_{i-1})$
- $\Rightarrow T_i^{(k+1)} = \frac{1}{2}(T_{i+1}^{(k)} + T_{i+1}^{(k)})$



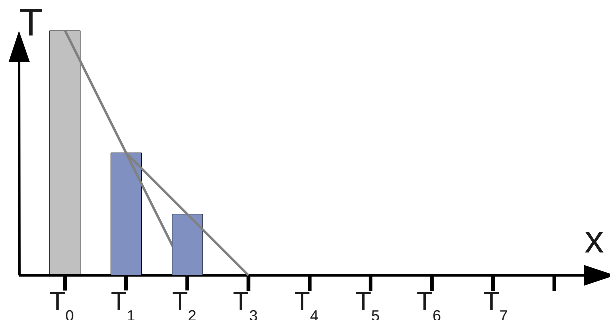
Jacobi-Verfahren (aus PDE-Sicht)

- Ist ein Relaxationsverfahren
- An jedem Punkt wird wird die PDE lokal erfüllt
- Neue Werte $T^{(k+1)}$ aus alten Werte $T^{(k)}$ berechnen
- 1D Wärmeleitung stationär: Zweite Ableigung muß Null sein
- $\Rightarrow T_{i+1} - 2T_i + T_{i-1} \ = \ 0 \Rightarrow T_i \ = \ \frac{1}{2}(T_{i+1} + T_{i-1})$
- $\Rightarrow T_i^{(k+1)} = \frac{1}{2}(T_{i+1}^{(k)} + T_{i+1}^{(k)})$



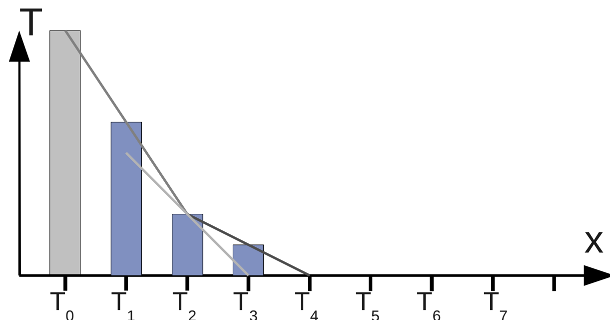
Jacobi-Verfahren (aus PDE-Sicht)

- Ist ein Relaxationsverfahren
- An jedem Punkt wird wird die PDE lokal erfüllt
- Neue Werte $T^{(k+1)}$ aus alten Werte $T^{(k)}$ berechnen
- 1D Wärmeleitung stationär: Zweite Ableigung muß Null sein
- $\Rightarrow T_{i+1} - 2T_i + T_{i-1} = 0 \Rightarrow T_i = \frac{1}{2}(T_{i+1} + T_{i-1})$
- $\Rightarrow T_i^{(k+1)} = \frac{1}{2}(T_{i+1}^{(k)} + T_{i+1}^{(k)})$



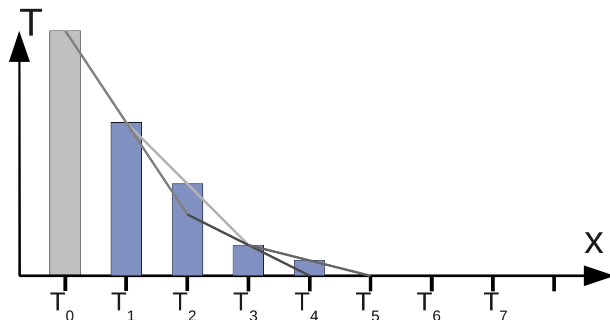
Jacobi-Verfahren (aus PDE-Sicht)

- Ist ein Relaxationsverfahren
- An jedem Punkt wird wird die PDE lokal erfüllt
- Neue Werte $T^{(k+1)}$ aus alten Werte $T^{(k)}$ berechnen
- 1D Wärmeleitung stationär: Zweite Ableigung muß Null sein
- $\Rightarrow T_{i+1} - 2T_i + T_{i-1} = 0 \Rightarrow T_i = \frac{1}{2}(T_{i+1} + T_{i-1})$
- $\Rightarrow T_i^{(k+1)} = \frac{1}{2}(T_{i+1}^{(k)} + T_{i+1}^{(k)})$



Jacobi-Verfahren (aus PDE-Sicht)

- Ist ein Relaxationsverfahren
- An jedem Punkt wird wird die PDE lokal erfüllt
- Neue Werte $T^{(k+1)}$ aus alten Werte $T^{(k)}$ berechnen
- 1D Wärmeleitung stationär: Zweite Ableigung muß Null sein
- $\Rightarrow T_{i+1} - 2T_i + T_{i-1} = 0 \Rightarrow T_i = \frac{1}{2}(T_{i+1} + T_{i-1})$
- $\Rightarrow T_i^{(k+1)} = \frac{1}{2}(T_{i+1}^{(k)} + T_{i-1}^{(k)})$



Jacobi-Verfahren (aus LGS-Sicht)

- Matrix A aufteilen

$$A \cdot x = (M + N) \cdot x = b$$

- Umformen

$$x = N^{-1}(b - M \cdot x)$$

- Vorige Gleichung als Iterationsvorschrift

$$\begin{aligned} x^{(k+1)} &= N^{-1}(b - M \cdot x^{(k)}) \\ &= N^{-1}(b - (A - N) \cdot x^{(k)}) \\ &= N^{-1}(b - A \cdot x^{(k)} + N \cdot x^{(k)}) \\ &= N^{-1}(b - A \cdot x^{(k)}) + x^{(k)} \\ &= x^{(k)} + N^{-1} \cdot r^{(k)} \end{aligned}$$

- N leicht invertierbar: Matrix der Diagonalelemente von A
- $x_i^{(k+1)} = x_i^{(k)} - \frac{1}{2}(-(1 \cdot x_{i-1}^{(k)} - 2 \cdot x_i^{(k)} + 1 \cdot x_{i+1}^{(k)})) = \frac{1}{2}(x_{i-1}^{(k)} + x_{i+1}^{(k)})$
- 2D: $x_{i,j}^{(k+1)} = \frac{1}{4}(x_{i-1,j}^{(k)} + x_{i+1,j}^{(k)} + x_{i,j-1}^{(k)} + x_{i,j+1}^{(k)})$

Jacobi-Verfahren ohne Aufstellung der Matrix

- Die Matrix A hat $\mathcal{O}(n^2)$ Elemente
- Die meisten Elemente sind Null $\Rightarrow$ Verschwendung von Speicher
- Entweder Matrix effizienter speichern, oder gar nicht aufstellen.
- Für jeden neuen x-Wert sind nur jeweils vier alte (2D) nötig
- Koeffizient ist dabei jeweils eins
- Die Matrix ist also überhaupt nicht nötig.

Randbehandlung

- Die Iterationsvorschrift $x_{i,j}^{(k+1)} = ...$ geht nur für innere Punkte
- Am Rand fehlen Nachbarelemente, dafür muss b berücksichtigt werden
- Andere Möglichkeit: Randpunkte in x aufnehmen und nur alle bisherigen x-Werte aktualisieren
- Damit zusätzlich zur Matrix A auch noch b unnötig

Implementierung

Maß für den Fehler

- Perfekt wäre ein Fehler (damit auch Residuum) gleich Null
- Dazu ist eine sehr hohe Zahl an Iterationen nötig
- So hohe Genauigkeit ist meistens nicht erforderlich
- So lange iterieren, bis Residuum unter einer Schranke ist.
- $res = \sqrt{\sum_{i=0}^{n} (res_i)^2 / n}$

Includes

```
from numpy import *
import matplotlib.pyplot as plt
from math import sqrt

class Waermesimulation:
   ...
```

Implementierung

Konstruktor

- Vorgabe von Gebietsbreite, -höhe und Zellbreite h
- Vorgabe der Temperatur an den vier Rändern
- Erstellung eines initialen Arrays für die Temperatur
- Null für innere Zellen, am Rand die gegebenen Werte

```
def __init__(self, lengthx, lengthy, h, leftT, \
             rightT, upperT, lowerT):
    self.zeilen = int(lengthy/h)
    self.spalten = int(lengthx/h)
    self.T = zeros((self.zeilen+2,self.spalten+2))
    for i in range(1, self.zeilen+1):
        self.T[i, 0] = leftT
        self.T[i, self.spalten+1] = rightT
    for i in range(1, self.spalten+1):
        self.T[0, i] = lowerT
        self.T[self.zeilen+1, i] = upperT
```

Iterationsschleife

- Iterieren bis Residuum kleiner als maxres
- Nicht direkt auf `self.T` arbeiten, sondern auf einer Kopie

```
def jacobi(self, maxres):
  res = maxres+1.
  while res > maxres:
    tempT = copy(self.T)
    res = 0
    for i in range(1, self.zeilen+1):
      for j in range(1, self.spalten+1):
        res += (tempT[i-1,j] + tempT[i+1,j] \
             + tempT[i,j-1] + tempT[i,j+1] \
             - 4 * tempT[i,j])**2
        self.T[i,j] = 0.25*(  tempT[i-1,j] \
                            + tempT[i+1,j] \
                            + tempT[i,j-1] \
                            + tempT[i,j+1])
    res = sqrt(res/(self.zeilen*self.spalten))
```

Ergebnis plotten

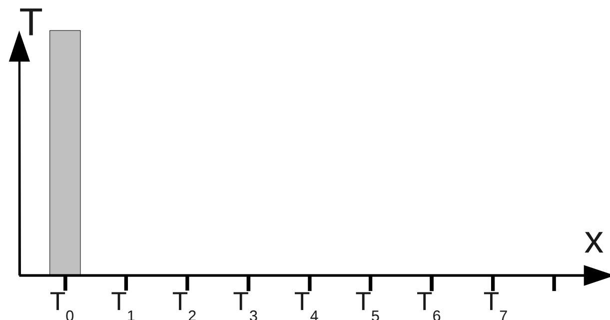
```
def plot(self):
  print self.T
  plt.pcolormesh(self.T)
  plt.show()
```

Beispiel

```
>>> from waermesim import *
>>> lgs = Waermesimulation(50., 30., 1., 0.6, 0., 1., 0.)
>>> lgs.jacobi(0.01)
>>> lgs.plot()
```

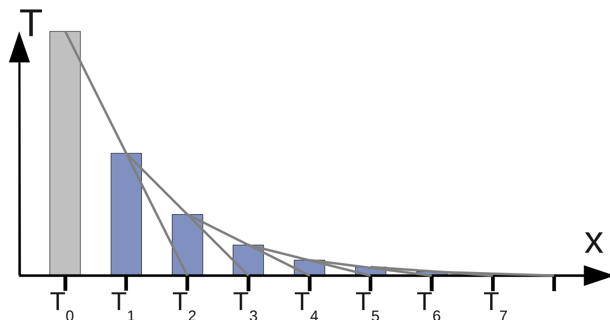
Gauss-Seidel-Verfahren

- Wie Jacobi, es werden aber alle neuen Werte gleich verwendet
- Jacobi 1D: $\Rightarrow T_i^{(k+1)} = \frac{1}{2}(T_{i+1}^{(k)} + T_{i-1}^{(k)})$
- Gauss-Seidel 1D: $\Rightarrow T_i^{(k+1)} = \frac{1}{2}(T_{i+1}^{(k)} + T_{i-1}^{(k+1)})$
- Achtung: Gauss-Seidel abhängig von Durchlaufreihenfolge
- LGS-Sicht: $N = D_A + L_A$ (Diagonale + unteres Dreieck)
  $x^{(k+1)} = x^{(k)} + N^{-1} \cdot r^{(k)}$



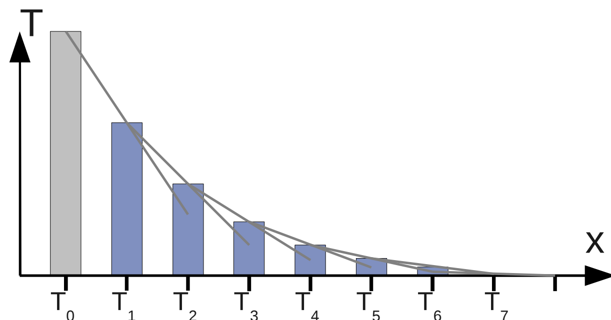
Gauss-Seidel-Verfahren

- Wie Jacobi, es werden aber alle neuen Werte gleich verwendet
- Jacobi 1D: $\Rightarrow T_i^{(k+1)} = \frac{1}{2}(T_{i+1}^{(k)} + T_{i-1}^{(k)})$
- Gauss-Seidel 1D: $\Rightarrow T_i^{(k+1)} = \frac{1}{2}(T_{i+1}^{(k)} + T_{i-1}^{(k+1)})$
- Achtung: Gauss-Seidel abhängig von Durchlaufreihenfolge
- LGS-Sicht: $N = D_A + L_A$ (Diagonale + unteres Dreieck)
  $x^{(k+1)} = x^{(k)} + N^{-1} \cdot r^{(k)}$



Gauss-Seidel-Verfahren

- Wie Jacobi, es werden aber alle neuen Werte gleich verwendet
- Jacobi 1D: $\Rightarrow T_i^{(k+1)} = \frac{1}{2}(T_{i+1}^{(k)} + T_{i-1}^{(k)})$
- Gauss-Seidel 1D: $\Rightarrow T_i^{(k+1)} = \frac{1}{2}(T_{i+1}^{(k)} + T_{i-1}^{(k+1)})$
- Achtung: Gauss-Seidel abhängig von Durchlaufreihenfolge
- LGS-Sicht: $N = D_A + L_A$ (Diagonale + unteres Dreieck)
  $x^{(k+1)} = x^{(k)} + N^{-1} \cdot r^{(k)}$



Implementierung Gauss-Seidel

- Unterschied zu Jacobi: direkt auf `self.T` arbeiten
- Vorteil: schneller und speichereffizienter

```
def gaussseidel(self, maxres):
  res = abs(maxres)+1.
  while abs(res) > abs(maxres):
    #tempT = copy(self.T)
    res = 0
    for i in range(1, self.zeilen+1):
      for j in range(1, self.spalten+1):
        res += (self.T[i-1,j] + self.T[i+1,j] \
              + self.T[i,j-1] + self.T[i,j+1] \
              - 4 * self.T[i,j])**2
        self.T[i,j] = 0.25*(   self.T[i-1,j] \
                             + self.T[i+1,j] \
                             + self.T[i,j-1] \
                             + self.T[i,j+1])
    res = sqrt(res/(self.zeilen*self.spalten))
```