universitätbonn

Institut für Numerische Simulation der Universität Bonn
Prof. Dr. Mario Bebendorf

Praktikum im Wintersemester 2013/2014

Programmierpraktikum numerische Algorithmen (P2E1)
(Numerische Lösung der Wärmeleitungsgleichung)

Betreuer: Christian Kuske[1]

Blatt 2

1 Einführung

Auf diesem Blatt werden weitere Verfahren zur Lösung linearer Gleichungssysteme der Form

$$L_h u_h = f_h \tag{1}$$

vorgestellt, wobei L_h eine schwach-besetzte (ggf. symmetrische) Matrix ist. Eine solche Matrix entsteht beispielsweise durch eine Diskretisierung des LAPLACE-Operators, wie sie auf Blatt 1 hergeleitet wurde.

Die Implementierung des GAUSS-SEIDEL-Verfahrens auf dem letzten Aufgabenblatt hat gezeigt, dass bei einer Verringerung der Gitterweite der Algorithmus immer langsamer konvergiert.

Konvergenzrate

Es lässt sich zeigen, dass das GAUSS-SEIDEL-Verfahren für die stationäre Wärmeleitungsgleichung mit

$$\rho(G) = 1 - \mathcal{O}(n^{-2}) < 1$$

konvergiert, wobei $G = D^{-1}(L + R)$ die Iterationsmatrix, $\rho(G)$ der Spektralradius und n die Anzahl der Eigenwerte dieser Matrix ist. Bei einer Verringerung der Gitterweite $h = \frac{1}{n-1}$ nähert sich diese Konvergenzrate $\rho(G)$ Eins an, so dass für große Gleichungssysteme der Zeitaufwand immer weiter ansteigt. Das SOR-Verfahren (*successive over-relaxation*) kann einen solchen Effekt vermindern, allerdings bleibt die Konvergenzordnung von zwei erhalten.

Vorkonditionierung

Eine weitere Möglichkeit besteht darin, das Gleichungssystem mit dem CG-Verfahren (*conjugated gradients*) zu lösen. Da die Konvergenzgeschwindigkeit des CG-Verfahrens von der Kondition der Matrix abhängig ist, stellt sich die Frage, wie diese verringert werden kann. Diese Idee wird durch das PCG-Verfahren (*preconditioned conjugated gradients*) realisiert.

[1]We6 4.002; Tel.: 0228/73-60454; Email: kuske@ins.uni-bonn.de

2 SOR-Verfahren

Das SOR-Verfahren ist eine Erweiterung des Gauss-Seidel-Verfahrens und kombiniert den letzten mit dem aktuellen Iterationsschritt um eine bessere Konvergenzrate zu erzielen. Für ein Gleichungssystem $Ax = b$ ist

$$x_i^{(k+1)} = \frac{\omega}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) + (1 - \omega) x_i^{(k)}$$

die Iterationsvorschrift des SOR-Verfahrens, wobei $\omega \in (0, 2)$ als Relaxationsparameter bezeichnet wird.

Relaxationsparameter

Für $\omega = 1$ ergibt sich das bereits bekannte Gauss-Seidel-Verfahren.

Wird $\omega < 1$ gewählt, so ist das Verfahren unterrelaxiert. Da jeder Iterationsschritt dazu dient, sich der exakten Lösung anzunähern, wird hierbei lediglich ein Teil des neuen Schritts genutzt und der vorherige bleibt teilweise vorhanden. Dies verlangsamt in den meisten Fällen das Verfahren weiter und wird daher zum direkten Lösen von linearen Gleichungssystemen nicht verwendet.

Dem entgegen steht die Wahl von $\omega > 1$. In diesem Fall wird das Verfahren Überrelaxationsverfahren oder SOR-Verfahren genannt. Der neue Iterationsschritt wird mit mehr als Eins gewichtet, sodass eine schnellere Annäherung an die Lösung stattfindet.

Optimierung

Es lässt sich zeigen, dass das SOR-Verfahren für positiv definite Matrizen und $\omega \in (0, 2)$ konvergiert [Beb10, 12.15]. Hierbei kann der optimale Relaxationsparameter durch

$$\omega_{\text{opt}} := \frac{2}{1 + \sqrt{1 - \rho(G)}}$$

gewählt werden, wobei für den diskretisierten Laplace-Operator $\rho(G) = \cos^2(\frac{\pi}{n})$ gilt. Dass die Konvergenzrate des SOR-Verfahren für ω_{opt} optimal ist, wird in [Hac91, Satz 6.6.5] bewiesen.

Da $\rho(G)$ nicht für jede Iterationsmatrix bekannt ist, wird oft ein ω zwischen 1,2 und 1,6 gewählt. Dadurch ergibt sich bereits eine Beschleunigung des Gauss-Seidel-Verfahrens.

Ebenfalls ist im Fall von schwach-besetzten Matrizen darauf zu achten, dass lediglich die Einträge, die nicht Null sind, berechnet werden müssen.

3 CG-Verfahren

Beim CG-Verfahren handelt es sich zunächst um ein direktes Verfahren, das heißt, dass das Verfahren nach n Iterationsschritten die exakte Lösung ermittelt hat. Der Beweis dazu ist in [Beb10, Lemma 12.22] zu finden. Bei großen n ist dies wenig hilfreich. Allerdings ist eine geringere Anzahl von CG-Schritten ausreichend, um den Fehler unter eine gegebene Schranke zu drücken. Der Fehler wird maßgeblich von der Kondition beeinflusst und wird durch [Beb10, Satz 12.27] abgeschätzt. Der Algorithmus ist ein Spezialfall vom vorkonditionierten CG-Verfahren 4.1, bei dem $C = I$ gesetzt wird.

Matrix-Vektor-Multiplikation

Da in jedem Iterationsschritt die Suchrichtung orthogonal zu den vorherigen Richtungen bezüglich der Matrix gewählt werden, ist es nötig, eine Matrix-Vektor-Multiplikation durchzuführen. Diese Operation kostet $\mathcal{O}(n^2)$ Schritte. Bei schwach-besetzten Matrizen, die bei der Diskretisierung durch Finite Differenzen auftreten, lässt sich der Aufwand auf $\mathcal{O}(n)$ reduzieren. Dadurch wird der Algorithmus beschleunigt und ein Aufstellen oder Abspeichern der gesamten Matrix entfällt.

4 Vorkonditionierung

Da die Konvergenzrate vieler Iterationsverfahren zum Lösen linearer Gleichungssysteme von der Kondition der Matrix A abhängt, ist es von Vorteil, diese zu verringern. Diese Idee wird durch die Vorkonditionierung verwirklicht. Im folgenden wird das Gleichungssystem

$$Ax = b \tag{2}$$

betrachtet, wobei das PCG-Verfahren hierfür wie folgt ausgeführt wird:

$r_0 = b - Ax_0,\ d_0 = h_0 = Cr_0$
do
  $x_{k+1} = x_k + \alpha_k d_k,\ \alpha_k = \frac{r_k^T h_k}{d_k^T A d_k}$
  $r_{k+1} = r_k - \alpha_k A d_k$
  $h_{k+1} = C r_{k+1}$
  $d_{k+1} = h_{k+1} + \beta_k d_k,\ \beta_k = \frac{r_{k+1}^T h_{k+1}}{r_k^T h_k}$
until $\|r_{k+1}\| < \varepsilon$

Algorithmus 4.1: PCG-Verfahren

Links-Vorkonditionierung

Bei dieser Vorkonditionierung wird eine reguläre Matrix C auf das Gleichungssystem (2) multipliziert, so dass ein neues System

$$CAx = Cb$$

entsteht. Dabei soll C so gewählt werden, dass CA eine geringere Kondition als A besitzt, also $C \approx A^{-1}$. Auf dieses veränderte Gleichungssystem lassen sich Iterationsverfahren anwenden. Allerdings ist darauf zu achten, dass sich durch die Multiplikation mit der Matrix C ebenfalls das Residuum zu

$$\tilde{r}_k := Cb - CAx_k = C(b - Ax_k)$$

ändert. Daher ist eine Überprüfung von $r_k := b - Ax_k$ auf die zu erreichende Genauigkeit, im Gegensatz zu $\tilde{r}_k$, sinnvoll.

Rechts-Vorkonditionierung

In diesem Fall wird das lineare Gleichungssystem (2) durch $y := C^{-1}x$ verändert und es ergibt sich

$$ACy = b.$$

Wie für die Links-Vorkonditionierung soll C leicht zu berechnen sein und die Kondition der Matrix A reduzieren. Die Berechnung des Residuums zeigt, dass dieses unverändert bleibt:

$$r_k = b - ACy_k = b - ACC^{-1}x_k = b - Ax_k.$$

Ist die vorgegebene Genauigkeit unterschritten, muss im letzten Schritt $x_k = Cy_k$ berechnet werden, um eine Lösung der Gleichung (2) zu erhalten.

Symmetrische Vorkonditionierung

Durch Kombination der Links- und Rechts-Vorkonditionierung kann (2) mit Hilfe von $C = (KK^T)^{-1}$ durch das folgende Gleichungssystem ersetzt werden:

$$K^T AKy = K^T b, \quad y := K^{-1}x.$$

Der Vorteil der symmetrischen Variante ist, dass die Matrix $K^T AK$ symmetrisch und positiv definit ist, falls dies für A gilt.

Allerdings lässt sich dies auch bei den unsymmetrischen Verfahren CA beziehungsweise AC zeigen, falls A, C beide Eigenschaften besitzen. Es sei $(x, y)_{C^{-1}} := (C^{-1}x, y)$ ein Skalarprodukt. Dann gilt für alle $x, y \in \mathbb{R}^n$

$$(CAx, y)_{C^{-1}} = (Ax, y) = (x, Ay) = (CC^{-1}x, Ay) = (C^{-1}x, CAy) = (x, CAy)_{C^{-1}}.$$

Daraus folgt, dass CA symmetrisch und positiv definit bezüglich $(x, y)_{C^{-1}}$ ist. Analoges gilt für AC bezüglich $(x, y)_C$.

5 Vorkonditionierer

Einige wichtige und häufig Vorkonditionierer sollen kurz beschrieben werden.

Jacobi

Als erstes ist der Jacobi-Vorkonditionierer zu nennen. Dieser besteht, ähnlich zum Jacobi-Verfahren, aus den inversen Diagonaleinträgen der Matrix A, das heißt $C = D^{-1}$. Daher hat die im PCG-Verfahren zusätzlich auftretende Matrix-Vektor-Multiplikation für den Vorkonditionierer einen Rechenaufwand von $\mathcal{O}(n)$. Dieses Verfahren verringert die Kondition vorallem bei Systemmatrizen, die durch eine nicht äquidistante Gitterweite diskretisiert wurden.

Gauß-Seidel

Dieser Vorkonditionierer leitet sich aus dem Gauss-Seidel-Verfahren ab. Dabei wird $C = (D + L)^{-1}$ gesetzt, wobei D die Diagonale und L die strikte untere Diagonalmatrix von A ist. Aufgrund der Dreiecksgestalt werden für die Matrix-Vektor-Multiplikation $\frac{1}{2}n^2$ Schritte benötigt. Im Fall schwach-besetzter Matrizen reduziert sich der Aufwand auf $\mathcal{O}(n)$.

Symmetrisches SOR

Hierbei handelt es sich um eine symmetrische Version des SOR-Verfahren. Für die Matrix des Vorkonditionierers gilt

$$C = (D + \omega R)^{-1} D (D + \omega L)^{-1},$$

wobei R die rechts oberen Einträge der Matrix A darstellt. Ähnlich zum Gauss-Seidel-Verfahren wird der Multiplikationsaufwand auf $n^2 + n$ Schritte berechnet und kann bei schwach-besetzten Matrizen auf $\mathcal{O}(n)$ verringert werden.

ILU

Normalerweise entstehen bei einer LR-Zerlegung vollbesetzte Matrizen, auch wenn eine schwach-besetzte Matrix vorliegt. Dadurch beträgt der Aufwand bei der Matrix-Vektor-Multiplikation durch die Vorkonditionierung $\mathcal{O}(n^2)$ und zusätzlich entsteht ein Aufwand von $\mathcal{O}(n^3)$ zur Berechnung einer LR-Zerlegung. Beide Effekte können durch die ILU-Zerlegung (*incomplete lower upper*) beseitigt werden. Es wird eine Zerlegung der Form

$$A = LR + E$$

gesucht, wobei E die Fehlermatrix ist. Der Fall $(E)_{ij} \neq 0$ wird lediglich zugelassen, falls $(A)_{ij} = 0$ gilt. Genauer bleiben die Nullen in der Matrix A in LR an den gleichen Stellen vorhanden. Dadurch entsteht eine schwach besetzte Zerlegung mit der $\mathcal{O}(n)$ Operationen für die Matrix-Vektor-Multiplikation benötigt werden.

IC

Analog zur LR-Zerlegung ist die CHOLESKY-Zerlegung aufwendig. Mit der gleichen Idee wie bei der ILU-Zerlegung kann die IC-Zerlegung (*incomplete Cholesky*) eine schwach-besetzte Matrix erzeugen. Diese hat die Form

$$A = LL^T + E.$$

Die für die CHOLESKY-Zerlegung vorausgesetzte Symmetrie der Matrix A ist in vielen in der Praxis vorkommenden Problemen gegeben, sodass die IC-Zerlegung meistens anwendbar ist.

6 Aufgaben

a) Implementiere das SOR-Verfahren für die Gleichung (1) und verwende dazu:

- `void SOR(const unsigned int n, const double * const f, double * const u, double * const eps, unsigned int * const iter, const double * const omega = 1)`,

wobei das Gleichungssystem $L_h u = f_h$ gelöst wird. Das Verfahren soll abbrechen, sobald $\|u^{(k+1)} - u^{(k)}\|_\infty < \varepsilon$ ist oder nach maximal `iter` Iterationsschritten. Die erreichte Genauigkeit und die dabei benötigten Iterationsschritte sollen in den jeweiligen Variablen gespeichert werden.

b) Erstelle eine Tabelle für $n = 50$, $\varepsilon = 10^{-4}$, $\omega_i = \frac{i}{100}$, $1 \leq i < 200$ und stelle die Anzahl der Iterationen gegenüber!

c) Implementiere eine Matrix-Vektor-Multiplikation für die schwach-besetzte Diskretisierung des LAPLACE-Operators in symmetrischer Form mit Hilfe der Matrix-Klasse:

- `void LaplaceMatrix::multMatVec(const double * const x, double * const y) const`,

wobei $y = L_h x$ berechnet wird und L_h symmetrisch positiv definit ist. (Hinweis: Verwende die geschriebene Funktion `applyLaplace()` aus Blatt 1.)

d) Implementiere das CG-Verfahren mit Hilfe der Matrix-Klasse und verwende folgende Funktionen:

- `unsigned int CG(const Matrix& L, const double * const f, double * const u, const double eps, const uint iter)`,

wobei das lineare Gleichungssystem $L_h u = f_h$ mit $\|r_{k+1}\| < \varepsilon$ und maximal `iter` Iterationsschritten gelöst wird. Die erreichte Genauigkeit und die dabei benötigten Iterationsschritte sollen in den jeweiligen Variablen gespeichert werden. Der Rückgabewert wird verwendet um die Konvergenz oder nicht Konvergenz anzuzeigen.

- `void scal(const unsigned int n, const double alpha, double * const x)`, wobei x mit α skaliert wird.
- `void copy(const unsigned int n, const double * const x, double * const y)`, wobei x nach y kopiert wird.
- `void axpy(const unsigned int n, const double alpha, const double * const x, double * const y)`, wobei $y + \alpha x$ in y gespeichert wird.
- `double scpr(const unsigned int n, const double * const x, const double * const y)`, wobei der Rückgabewert der Skalarprodukt $\langle x, y \rangle$ entspricht.

Hinweis: Gehe Schritt für Schritt vor und überprüfe an Hand eines kleinen Beispiels ($n = 5$) jede geschriebene Funktion.

e) Implementiere das PCG-Verfahren mit dem symmetrischen Jacobi-Vorkonditionierer und mit Hilfe der Matrix-Klasse:

- `void LaplaceMatrixJacobi::applyPrecond(double * const x) const`, wobei $x = Cx$ berechnet wird und C der Vonkonditioniere des Jacobi-Verfahren ist. (Hinweis: Die Matrix C wird dabei nicht aufgestellt.)

f) Erstelle eine Tabelle in der Rechenzeiten und Iterationsschritte für Gauss-Seidel, SOR, CG und PCG mit unterschiedlichen n verglichen werden.

Literatur

[Beb10] Bebendorf, Mario: *Algorithmische Mathematik 2 – Vorlesungsskript SS2010*. `http://bebendorf.ins.uni-bonn.de/teaching/AlMa2_SS10/AlMa2.pdf`. Version: 2010

[Hac91] Hackbusch, Wolfgang: *Iterative Lösung großer Gleichungssysteme*. Teubner, 1991