Institut für Analysis und Numerik
Universität Magdeburg
Thomas Richter
Henry von Wahl

26.06.2020

thomas.richter@ovgu.de
henry.vonwahl@ovgu.de

Blatt 11 zur Vorlesung Numerische Mathematik
Sommersemester 2020

Abgabe bis 3. Juli, 23:59

Evaluation Wir wurden darauf hingewiesen, dass der Rücklauf der Evaluation bisher sehr gering ist. Falls die Evaluation (link wurde per Moodle verschickt) noch freigeschaltet ist, bitten wir um Teilnahme.

Lineare Gleichungssysteme II In diesem letzten Abschnitt der Vorlesung geht es noch einmal um lineare Gleichungssysteme. Dabei haben wir sehr große aber dünn besetzte Gleichungssysteme im Sinn. Für $A \in \mathbb{R}^{n\times n}$ mit sehr großem n, wir gehen von $n > 1\,000\,000$ aus, wird dennoch nur eine feste Anzahl von Einträgen pro Zeile ungleich Null sein.
Wir fassen zunächst einige Punkte aus der der bisherigen Vorlesung sowie der Algorithmischen Mathematik zusammen

1. Das allgemeinste Verfahren ist die LR-Zerlegung. Sie erfordert jedoch $\mathcal{O}(n^3)$ Operationen. $n = 10^6$ bedeutet hier 10^{18} Operationen und somit eine Rechnenzeit von etlichen Jahren.

2. Auch wenn die Matrix dünn besetzt ist, können die Matrizen L und R voll besetzt sein. Das Überschreiben von Nullen durch Werte wird *fill-ins* genannt. Im ungünstigsten Fall müssen wir somit n^2 also 10^{12} Werte speichern. Im Datentyp `double` sind dies fast 8 Terabyte (Arbeitsspeicher).

3. Wenn wir die Matrix durch Zeilen- und Spaltentausch entsprechend permutieren können wir eine Bandform erreichen. Gelingt es uns, die Bandbreite auf $m \ll n$ zu reduzieren, so sinkt der Aufwand auf $\mathcal{O}(nm^2)$ und der Speicher auf $\mathcal{O}(nm)$. Der Aufwand kann jedoch immer noch sehr groß sein.

4. Weitere direkte Verfahren wir die QR-Zerlegung sind - in Bezug auf den Aufwand - hier nicht überlegen. Ist die Matrix A symmetrisch, so kann der Aufwand und Speicherbedarf mit der Cholesky-Zerlegung halbiert werden.

Eine Alternative sind iterative Verfahren. Ausgehend von einem Startwert $x^0 \in \mathbb{R}^n$ iterieren wir für $l = 1, 2, \ldots$

$$x^l = x^{l-1} + C(b - Ax^{l-1})$$

mit einer Matrix C, die eine Approximation an A^{-1} ist. Wir fassen kurz zusammen

1. Falls x^l die Lösung von $Ax = b$ ist, so gilt für den Defekt $b - Ax^l = 0$ und die Iteration bricht mit der Lösung ab

2. Falls $C = A^{-1}$ gilt, so gilt für beliebigen Startwert

$$x^1 = x^0 + A^{-1}b - A^{-1}Ax^0 = A^{-1}b = x,$$

   d.h. wir haben die Lösung gefunden. Dieses Vorgehen ist jedoch nicht möglich. Könnten wir A^{-1} berechnen, so könnten wir das LGS durch eine einfache Multiplikation mit $C = A^{-1}$ lösen und würden keine weitere Methode mehr brauchen. $C = A^{-1}$ ist also optimal in Bezug auf die Konvergenz, jedoch zu aufwändig. Die Alternative $A = I$ ist optimal in Bezug auf den Aufwand, führt jedoch im Allgemeinen nicht zur Konvergenz.

3. Für den Fehler $e^l = x - x^l$ gilt

$$x^l = x^{l-1} - CAx^{l-1} = (I - CA)x^{l-1}$$

   und wir können ablesen: die Methode konvergiert, falls

$$\rho := \|I - CA\| < 1.$$

   Dabei gibt ρ die Konvergenzrate ab.

4. Da es verschiedene Matrixnormen gibt, kommt es bei der Bestimmung der Konvergenzrate auf die Wahl der Norm an. Wir erhalten das optimale und scharfe Ergebnis mit der Betrachtung des Spektralradius

$$\rho := \operatorname{spr}(I - CA), \quad \operatorname{spr}(I - CA) = \max_{\{} |\lambda|,\ \lambda \text{ ist Eigenwert von } I - CA\}$$

5. Die einfachsten Verfahren sind die Richardson Iteration mit $C = \omega I$ für ein $\omega > 0$ und die Jacobi-Iteration mit $C = D$, wobei D die Diagonale der Matrix A ist. Dem liegt die additive Zerlegung

$$A = L + D + R$$

   zugrunde. Wählt man $C = L + D$ so resultiert das Gauß-Seidel-Verfahren. Alle diese Verfahren konvergieren nur für spezielle Matrizen, z.B. symmetrisch positiv definit oder diagonaldominant.

6. Der Aufwand dieser drei Verfahren ist pro Schritt im Wesentlichen der Aufwand einer Matrix-Vektor Multiplikation. Dies ist der große Vorteil dieser Verfahren: da die Matrix dünn besetzt ist und insgesamt nur $\mathcal{O}(n)$ Elemente ungleich Null hat, ist auch der Aufwand pro Schritt nur $\mathcal{O}(n)$. Das Problem ist, dass die Methoden im Allgemeinen sehr viele Schritte benötigen und dass der Gesamtaufwand üblicherweise nicht geringer ist als der Aufwand der LR-Zerlegung bei guter Sortierung. Man betrachte hierzu z.B. Beispiel *7.12 Jacobi- und Gauß-Seidel Verfahren bei der Modellmatrix*, welches die extrem schnell steigende Anzahl von benötigten Iterationsschritten zeigt.

Zur Erinnerung an die Grundbegriffe überfliege man bei Bedarf die ersten Abschnitte von Kapitel *7 Numerische Iterationsverfahren für lineare Gleichungssysteme* bis einschließlich Abschnitt *7.2 Konvergenzkriterium für Jacobi- und Gauß-Seidel-Iteration*.

Abstiegs- und Gradientenverfahren Man lese Abschnitt *7.5 Abstiegs- und Gradientenverfahren*. Wichtig ist hier zunächst die Charakterisierung der Lösung eines Linearen Gleichungssystems als Lösung einer Minimierungsaufgabe in *Satz 7.17* (*Lineares Gleichungssystem und Minimierung*). Die folgenden Verfahren werden nun nicht mehr darauf abzielen das LGS direkt zu lösen. Stattdessen wird versucht das Minimum von $Q(x) = \frac{1}{2}(Ax, x) - (b, x)$ zu finden.

1. Warum ist es für Satz 7.17 wichtig, dass die Matrix A symmetrisch ist? An welcher Stelle genau geht dies ein?
2. Welcher Schritt geht schief, wenn die Matrix A symmetrisch ist, aber nicht positiv definit?
3. Wie sieht das Energiefunktion aus im Fall $A \in \mathbb{R}^{2\times 2}$ mit $A = I$ (Einheitsmatrix) und $b = (1, 2)$?
4. Beim Abstiegsverfahren wird in jedem Schritt ein eindimensionales Minimierungsproblem gelöst um die optimale Schrittweite zu finden. Warum ist dies für das Funktional $Q(x)$ einfach? Bei allgemeinen Optimierungsproblemen ist doch auch dieses eindimensionale Problem sehr schwer.
5. Was unterscheidet das Gradientenverfahren vom allgemeinen Abstiegsverfahren?
6. Warum wird gerade $-\nabla Q(x)$ als Abstiegsrichtung gewählt?
7. Welche wesentlichen Operationen müssen pro Schritt des Gradientenverfahrens erfolgen?
8. Im Beweis zu Satz 7.20, der die qualitative Konvergenz des Verfahrens zeigt wird am Ende $1/(2\lambda_{max}) > 0$ genutzt um zu zeigen, dass wirklich Konvergenz vorliegt. Ist die andere Schranke, also $\lambda_{min}(A)(d, d) < (Ad, d)$ in diesem Satz auch wichtig? Wofür genau?
9. Von welcher Größe hängt die Konvergenzrate des Gradientenverfahrens wesentlich ab?

Dünn besetzte Matrizen in Python Das Paket `scipy` baut auf `numpy` auf und bietet ein Interface für dünn besetzte Matrizen. Nur wenn wir wirklich ausnutzen, dass die meisten Einträge der Matrix Null sind, können wir Effizienz erlangen, denn auch die Multiplikation mit Null oder die Addition von Null kostet jeweils eine Iteration. Darüber hinaus wollen wir die Nullen natürlich nicht speichern.
Es gibt verschiedene Formate zum Speichern von dünn besetzten Matrizen. Für uns sind zwei interessant:

COO steht für *Coordinate Format* und folgt einer einfachen Idee: anstelle der Matrix werden 3 `arrays` der Länge `nnz` gespeichert. Dabei ist `nnz` die Anzahl der Nicht-Null Einträge (**n**umber of **n**on **z**eros). Die Arrays sind `row`, `col` und `val` und die Bedeutung ist schlicht

$$A_{\texttt{row[i]},\texttt{col[i]}} = \texttt{val[i]}$$

Es muss keine Sortierung eingehalten werden. Die Vektoren müssen nur die richtige Länge haben und `row` und `col` müssen von einem ganzzahligen Typ sein. Ein Beispiel

```python
import numpy as np
from  scipy import sparse

row = np.array([0,1,1,2])        # Zeilenindizes
col = np.array([0,1,2,0])        # Spaltenindizes
val = np.array([0.1,-2.0,0.5,1]) # Eintraege

# Initialisierung der COO-Matrix
A = sparse.coo_matrix( (val, (row,col) ), shape=(3,3) )

# Ausgabe im COO-Format
print(A)

# Leserliche Ausgabe als Matrix
print(A.toarray())
```

Das COO-Format ist sehr gut geeignet um dünn besetzte Matrizen zu erstellen. Es eignet sich jedoch nicht sonderlich gut, um effizient zu Rechnen. Die Multiplikation von Matrizen oder die Matrix-Vektor Multiplikation erfordert das ständige Suchen von Indizes

CSR steht für *Compressed Sparse Row*. Auch hier werden drei Vektoren gespeichert, wieder `val` der Länge `nnz` mit den Einträgen. Weiter wird ein Vektor `col` der Länge `nnz` gespeichert, der zu jedem Eintrage angibt, zu welcher Spalte er gehört. Die Zeile wird jedoch nicht separat gespeichert. Stattdessen wird ein Vektor `rowptr` der Länge `n+1` (n ist Zahl der Zeilen) gespeichert. Die Bedeutung von `rowptr` ist die Folgende: Der Eintrag `rowptr[i]` gibt den ersten Index der Vektoren `col` sowie `val` an, der sich auf die Matrizzeile `i` bezieht. D.h. alle Indizes `j` gemäß

$$\texttt{i} \leqslant \texttt{j} < \texttt{i+1}$$

beziehen sich auf Zeile `i` und es gilt

$$A_{\texttt{i},\texttt{row[j]}} = \texttt{val[j]}, \quad \texttt{j} = \texttt{i}, \ldots, \texttt{i}-1-1.$$

Wir erstellen die Matrizen oft im `COO`-Format und wechseln dann in das effiziente `CSR`-Format. Hierzu setzen wir das Beispiel oben fort

```
#...

# Konvertieren in das CSR-Format
B = sparse.csr_matrix(A)

# Ausgabe im CSR-Format
print(B)

# Ausgabe als Matrix
print(B.toarray())
```

Die Initialisierung der Testmatrix ist in der Vorlage zu dieser Woche vorgegeben. Die Details der Speicherung müssen daher nicht genau verstanden werden. Für weiteres verweisen wir noch auf die entsprechenden Wikipedia-Seiten `https://en.wikipedia.org/wiki/Sparse_matrix#Coordinate_list_(COO)` sowie `https://de.wikipedia.org/wiki/D%C3%BCnnbesetzte_Matrix` und `https://de.wikipedia.org/wiki/Compressed_Row_Storage`

Übungsaufgaben

Aufgabe 11.1

Man beweise, dass die Wahl der Suchrichtung

$$d^l = \frac{b - Ax^l}{\|b - Ax^l\|}$$

lokal optimal ist. D.h. in Richtung d^l gilt

$$\lim_{s\to 0} \frac{Q(x + sd^l) - Q(x)}{s\|d^l\|} \leqslant \lim_{s\to 0} \frac{Q(x + sw) - Q(x)}{s\|w\|} \quad \forall w \in \mathbb{R}^n$$

Programmieraufgabe 11.2

Man implementiere das Gradientenverfahren zur Lösung des LGS $Ax = b$ zur Modellmatrix und der vorgegebenen rechten Seite. Diese rechte Seite wird mittels $b = Ax$ zu einer bekannten Lösung bestimmt, so dass der Fehler des Gradientenverfahrens bestimmt werden kann. Dies ist bereits in `template_11.py` vorgegeben.
a) Für $n = 16$ nutze man das Gradientenverfahren ausgehened vom Startvektor `x=np.zeros(n)` zum Lösen des linearen Gleichungssystems. Man iteriere, bis `np.linalg.norm(d)<1.e-6`, vergleiche *Algorithmus 7.2 Gradientenverfahren*. Man dokumentiere die Lösung und vergleiche mit der vorgegebenen exakten Lösung, d.h. man gebe `np.linalg.norm(x-xex)` aus, wobei `x` die Lösung des Gradientenverahrens ist und `xex` die exakte Lösung.

b) Man löse das LGS nun für große Matrizen, d.h. $n = m^2$ mit $m = 2, 4, 8, 16, \ldots$ und gebe jeweils wieder `np.linalg.norm(x-xex)` aus, zusätzlich noch `np.linalg.norm(b-A.dot(x))`. Sie sollten beobachten, dass der Fehler bei grossen Matrizen steigt, obwohl das Residuum nach wie vor die Schwelle 10^{-6} erfüllt. Haben Sie eine Erklärung?

c) Für $m = 2, 4, 8, 16, \ldots$ stelle man den Aufwand sowie die Anzahl der benötigten Schritte graphisch dar.