Implementierung der Cramer´schen Regel in Python

Autoren

Felix Wolf, Hannes Götz, Lorenz Schech
Fakultät für Informationstechnik, Hochschule Mannheim

Abstract

Das händische Anwenden der Cramer´schen Regel bei Matrizen, die größer als eine 3x3 Matrix sind, ist aufwendig und fehleranfällig. Daher ist es hilfreich, wenn bei derartig großen Matrizen ein Algorithmus genutzt werden kann. Das Ziel dieser Arbeit ist eine eigene Implementierung der Cramer´schen Regel in Python für beliebig große Matrizen ($n < 30$) zu entwickeln, sowie den Rechenaufwand, als auch die numerische Stabilität genauer zu betrachten.
Für die Implementierung des Algorithmus haben wir uns für Python entschieden, da diese eine der gängigsten Programmiersprachen für das Lösen mathematischer Probleme ist. Unteranderem werden wir Jupiter Lab zur Dokumentation und Visualisierung möglicher Ungenauigkeiten verwenden.
Das Ergebnis der Arbeit wird als Paper und als GitHub Repository zur Verfügung gestellt.
Durch diese Arbeit erhoffen wir uns ein besseres Verständnis für das Lösen großer Matrizen. Aber vor allem eine rechenunaufwändige Methode, die trotzdem eine hohe numerische Genauigkeit aufweist, zu entwickeln.

Einleitung

Die Cramer´sche Regel ist eine klassische Methode zur Lösung linearer Gleichungssysteme, die auf der Verwendung von Determinanten basiert. Da dieses Verfahren eine hohe Rechenkomplexität aufweist, ist es für Matrizen, die größer als (3x3) sind, in der Praxis kein sinnvoller Weg, um händisch lineare Gleichungssysteme aufzulösen. Mittels leistungsfähiger Computer und einem effizient implementierten Algorithmus beispielsweise in Python ist es dennoch eine interessante Möglichkeit, größere lineare Gleichungssysteme zu lösen.
Dieses Paper präsentiert eine detaillierte Implementierung der Cramer´schen Regel in Python. Es wird gezeigt, wie lineare Gleichungssysteme mittels eines Algorithmus gelöst werden. Hierbei liegt der Hauptfokus auf der Untersuchung der numerischen Stabilität, als auch dem Rechenaufwand, bei immer größer werdenden Matrizen.

Zunächst wird die technische Umsetzung zur Berechnung der Determinante aufgezeigt. Anschließend wird der Algorithmus auf seine Effizienz untersucht. Abschließend wird der Algorithmus bezüglich Schwächen betrachtet, die Einfluss auf die numerische Stabilität haben. Dies sind beides wichtige Faktoren, um die Praktikabilität in der Praxis beurteilen zu können. Durch die Kombination aus theoretischem Wissen und praktischer Implementierung in Python, zielt dieses Paper darauf ab das Verständnis für die Cramer´sche Regel zu erhöhen und das Lösen von linearen Gleichungssystemen zu erleichtern.

Material und Methoden

Für die Implementierung der Cramer´schen Regel wurde Python 3.8 ebenso wie Jupiter Notebook verwendet.

In Python wurden folgende Bibliotheken genutzt:

- [1]Mathe: Bibliothek für Mathematischer Grundfunktionen
- [2]timeit: Bibliothek zu Messung der Zeit
- [3]matplotlib.pyplot: Bibliothek zu Visualisierung von Daten (Graphen)
- [4]copy: Bibliothek zur Tiefenkopie von Listen
- [5]random: Bibliothek zur Erstellung von willkürlichen Zahlen

Die Berechnung der Determinante wurde mittels LU-Dekomposition umgesetzt. Unter der LU-Komposition versteht man ein mathematisches Verfahren, welches die Matrix in ein Produkt aus einer unteren und einer oberen Dreiecksmatrix aufteilt. Dieses Verfahren hat die Komplexitätsklasse $O(n^3)$. Im Anschluss kann die Determinante als Produkt aus den Diagonalelementen der beiden Dreiecksmatrizen berechnet werden. Zu Beginn wird durch die Berechnung der Determinante überprüft, ob das Gleichungs-System eine Lösung besitzt. Sollte die Determinante der Matrix null sein, so besitzt sie keine Lösung.

Zur finalen Berechnung der Lösungen mittels der Cramer´schen Regel werden Untermatrizen gebildet. Hierfür wird der Lösungsvektor durch die Matrix geschoben und ersetzt jeweils eine Spalte. Nun wird von jeder Untermatrix die Determinante berechnet und im Anschluss durch

[1] Python Software Foundation, (2024, Juni 05). Math - Mathematical functions. https://docs.python.org/3/library/math.html (Zugriff am 2024, Juni 06)
[2] Python Software Foundation, (2024, Juni 05). timeit – Measure execution time of small code. https://docs.python.org/3/library/timeit.html (Zugriff am 2024, Juni 06)
[3] The Matplotlib development team. (o. D.). matplotlib. https://matplotlib.org/stable/api/pyplot_summary.html (Zugriff am 2024, Juni 06)
[4] Python Software Foundation, (2024, Juni 05). copy – Shallow and deep copy operations. https://docs.python.org/3/library/copy.html (Zugriff am 2024, Juni 06)
[5] Python Software Foundation, (2024, Juni 05). random – Generate pseudo-random numbers. https://docs.python.org/3/library/random.html (Zugriff am 2024, Juni 06)

die Determinante der Originalmatrix geteilt. So erhält man für jede Untermatrix eine Lösung. Zur Komplexität der Determinante kommt noch die Komplexität des Verschiebens durch die Matrix, welche O(n) ist, hinzu, womit sich eine Gesamt-Komplexitätsklasse von $O(n^4)$ ergibt.

Beispiel

$$\begin{bmatrix} 3 & -1 & -1 \\ -1 & 3 & -1 \\ -1 & -1 & 3 \end{bmatrix} = \begin{pmatrix} 20 \\ -20 \\ -5 \end{pmatrix} \quad \begin{bmatrix} 3 & -1 & -1 \\ -1 & 3 & -1 \\ -1 & -1 & 3 \end{bmatrix} \rightarrow detA = 16$$

$$\begin{bmatrix} 20 & -1 & -1 \\ -20 & 3 & -1 \\ -5 & -1 & 3 \end{bmatrix} \rightarrow detA_1 = 60 \rightarrow L_1 = \frac{detA_1}{detA} = \frac{60}{16} = 3,75$$

$$\begin{bmatrix} 3 & 20 & -1 \\ -1 & -20 & -1 \\ -1 & -5 & 3 \end{bmatrix} \rightarrow detA_2 = -100 \rightarrow L_2 = \frac{detA_2}{detA} = \frac{-100}{16} = -6,25$$

$$\begin{bmatrix} 3 & -1 & 20 \\ -1 & 3 & -20 \\ -1 & -1 & -5 \end{bmatrix} \rightarrow detA_3 = -40 \rightarrow L_3 = \frac{detA_3}{detA} = \frac{-40}{16} = -2,5$$

Ergebnisse

Um eine genauere Aussagekraft über den Rechenaufwand treffen zu können, haben wir mehrfach Gleichungssysteme aufstellen und lösen lassen, während wir die Laufzeit durch eine Computerfunktion gemessen haben. Im Anschluss wurde daraus der Mittelwert ermittelt und in Grafiken dargestellt.

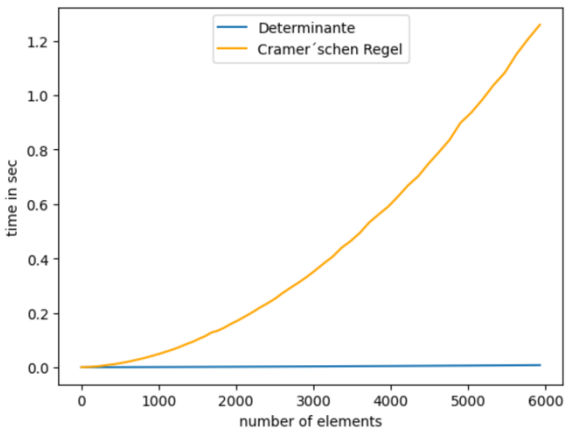

Abb. 1: Laufzeit Berechnung der Determinante und der Cramer´schen Regel

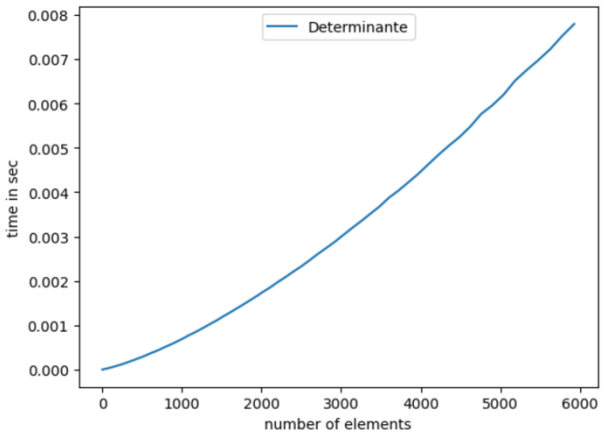

Abb. 2: Laufzeit der Determinantenberechnung

Abb. 1 zeigt die Laufzeit der Berechnung der Determinante und die Lösung mittels Cramer´scher Regel von Matrizen bis zum Typen (77x77) im Vergleich.

Abb. 2 zeigt die Laufzeit der Berechnung der Determinante von Matrizen bis zum Typen (77x77).

```
# Beispielrechung rundungsfehler
matrix = [[3,-1, -1], [-1,3, -1], [-1,-1, 3], ]
loesung = [20,-20, 5]
test = cramerische_regel(matrix, loesung)
print(test)
```

Executed at 2024.06.05 13:50:05 in 19ms

```
[6.25, -3.75, 2.5000000000000004]
```

Abb. 3 zeigt einen Rundungsfehler bei der Berechnung nach Anwendung der Cramer´schen Regel.

Abb. 3: Codeausschnitt

```
# Beispielrechung (Rechenfehler)
matrix = [[92388, 58526, 17401], [75621, 4783, 98154], [26878, 26427, 50936]]
loesung = [39662, 58691, 90459]
print('Originalmatrix: ',matrix, '\nLösungsvektor:', loesung)
for i in range(len(matrix)): # calculate the solution with cramer's rule
    new_matrix = copy.deepcopy(matrix)
    for j in range(len(matrix)): # replace the column with the solution
        new_matrix[i][j] = loesung[j]
    print('\n'+str(i + 1)+'.', 'UnterMatrix:',new_matrix)
    print('DetA_'+str(i + 1)+' =',calc_determinant(new_matrix))
result_cra = cramerische_regel(matrix, loesung)
print('\nLösung:', result_cra)
```

Executed at 2024.06.06 13:58:55 in 17ms

```
Originalmatrix:  [[92388, 58526, 17401], [75621, 4783, 98154], [26878, 26427, 50936]]
Lösungsvektor: [39662, 58691, 90459]

1. UnterMatrix: [[39662, 58691, 90459], [75621, 4783, 98154], [26878, 26427, 50936]]
DetA_1 = 4699982177743.019

2. UnterMatrix: [[92388, 58526, 17401], [39662, 58691, 90459], [26878, 26427, 50936]]
DetA_2 = 70184252097800.01

3. UnterMatrix: [[92388, 58526, 17401], [75621, 4783, 98154], [39662, 58691, 90459]]
DetA_3 = -590834507140697.0

Lösung: [-0.0183858287778426, -0.2745533053469477, 2.3112815482092732]
```

Abb. 4: Lösung mit Rechenfehler

A			x		b
92388	58526	17401	-1.060		39662
75621	4783	98154	1.959	=	58691
26878	26427	50936	1.319		90459

Abb. 5: Lösung ohne Rechenfehler

Abb. 4 zeigt wie bei der Berechnung, der Lösung einer Beispielmatrix, ein Rechenfehler aufgetreten ist, wohingegen Abb. 5 die korrekte Lösung des Gleichungssystems zeigt.

1	$\log n$	n	$n \log n$	n^2	n^3	2^n
konstant	logarithmisch	linear	–	quadratisch	kubisch	exponentiell
1	1	2	2	4	8	4
1	3	8	24	64	512	256
1	6	64	384	4096	262.144	10^{19}
1	8	256	2048	65536	16.777.216	10^{77}
1	10	1024	10.240	1.048.576	10^9	10^{308}
1	12	4096	49.152	16.777.216	10^{10}	$10^{1.230}$
1	14	16384	229.376	268.435.456	10^{12}	$10^{4.932}$
1	16	65536	1.048.576	10^9	10^{14}	$10^{19.728}$
1	18	262.144	4.718.592	10^{10}	10^{16}	$10^{78.913}$
1	20	1.048.576	20.971.520	10^{12}	10^{18}	$10^{315.652}$

Abb. 6: Tabelle von Komplexitätsklassen

Diskussion

Leistung und Effizienz

Aus Abb. 1 lässt sich ableiten, dass die Berechnungsdauer der Determinante im Verhältnis zur Lösung des Gleichungssystems mittels Cramer´scher Regel nahezu konstant ist. Zudem folgt aus der Grafik, dass mit zunehmender Matrizengröße die Laufzeit der Lösung des Gleichungssystems quartisch wächst. Ebenso lässt sich aus der Grafik ablesen, dass der Algorithmus bis zu einer Matrizengröße von (76x76) effizient ist (Berechnungszeit ca. 0,98s). Wie in Abb. 2 zu sehen, steigt die Laufzeit der Berechnung der Determinante dennoch kubisch an [$O(n^3)$]. Zur Lösung mittels Cramer´scher Regel ist die Berechnung der Determinante von n-Untermatrizen erforderlich, hierbei ist n die Anzahl an Spalten/Zeilen. Hier raus folgt die Komplexität $O(n^4)$, welche auch zur Folge hat, dass wie in Abb. 1 zu sehen, die Laufzeit der Berechnung der Determinante konstant wirkt.

Berechnet man nun auf Basis der Komplexitätsklassen [$O(n^3)$, $O(n^4)$] die maximale Größe einer Matrix, stellt man fest, dass für das Lu-Zerlegungsverfahren eine Matrix mit n=114 und für die Cramer´schen Regel n=196 das Maximum ist. [6]Hier beziehen wir uns darauf, dass die maximale Anzahl an Operationen zur Berechnung mit 10^9 Operationen begrenzt ist, bis wohin ein Algorithmus als effizient gilt. Als Funktion zur Berechnung der Schritte wurde $\frac{2}{3}n^4 = 10^9$ genutzt.

[6] Barth P., Bohli J. (2022). Algorithmen und Datenstrukturen (Skript). Hochschule Mannheim [Seite 30]

Nummerische Stabilität

In Abb. 3 ist zu erkennen das bei der dritten Lösung des Gleichungssystems ein Darstellungsfehler entstanden ist (erwünscht wäre der Wert 2,5). Dieser kann Auswirkungen auf Folgerechnungen haben. [7]Dies ist darauf zurückzuführen, dass die Binärdarstellung von Brüchen in vielen Programmiersprachen nicht exakt darstellbar ist.

Aus Abb. 4 und Abb. 5 lässt sich schließen das Rechenfehler und daraus resultierende falsche Lösungen in Python entstehen können da eine Variable vom Speicher überladen wird. Diese Fehler treten vermehrt auf, wenn die Werte der Matrix größer werden.

Bedeutung und Anwendung

Trotz ihrer Einschränkungen bietet die Implementierung der Cramer´schen Regel verschiedene Vorteile. Zum einen hilft sie die Prinzipien der Determinantenberechnung und die Lösung von linearen Gleichungssystemen zu verdeutlichen. Desweitern zeigt sie aber auch, dass selbst diese Implementierung an technische Grenzen kommt. Moderne Bibliotheken und Algorithmen bieten hier deutlich besser alternativen. Diese sind jedoch meist deutlich komplexer und daraus hergehend schwieriger als Anwender zu verstehen.

Fazit

In dieser Arbeit haben wir eine eigene Implementierung der Cramer'schen Regel in Python für beliebig große Matrizen entwickelt und deren numerische Stabilität sowie den Rechenaufwand untersucht. Insgesamt liefert diese Arbeit wertvolle Erkenntnisse über die Anwendung der Cramer'schen Regel, jedoch ist zu beachten, dass es in unserem Algorithmus zu numerischer Instabilität kommen kann. Das aufgetretene Problem bietet eine Grundlage für weitere Forschung und Entwicklung hinsichtlich numerischer Stabilität von Algorithmen.

Git-Zugriff

Unsere Ergebnisse sind in einem git veröffentlicht und können über diese link eingesehen werden. https://github.com/FelixWoRe/mathe-cramersche-regel/

[7] Python Software Foundation. (2024, Juni 05). Floating Point Arithmetic: Issues and Limitations. Python-Docs. https://docs.python.org/3/tutorial/floatingpoint.html (Zugriff am 2024, Juni 06)

Literaturverzeichnis

Literaturquellen

[1] Barth P., Bohli J. (2022). Algorithmen und Datenstrukturen (Skript). Hochschule Mannheim [Seite 30]

Internetquellen

[2] Python Software Foundation, (2024, Juni 05). Math - Mathematical functions. https://docs.python.org/3/library/math.html (Zugriff am 2024, Juni 06)

[3] Python Software Foundation, (2024, Juni 05). timeit – Measure execution time of small code. https://docs.python.org/3/library/timeit.html (Zugriff am 2024, Juni 06)

[4] The Matplotlib development team. (o. D.). matplotlib. https://matplotlib.org/stable/api/pyplot_summary.html (Zugriff am 2024, Juni 06)

[5] Python Software Foundation, (2024, Juni 05). copy – Shallow and deep copy operations. https://docs.python.org/3/library/copy.html (Zugriff am 2024, Juni 06)

[6] Python Software Foundation, (2024, Juni 05). random – Generate pseudo-random numbers. https://docs.python.org/3/library/random.html (Zugriff am 2024, Juni 06)

[7] Python Software Foundation. (2024, Juni 05). Floating Point Arithmetic: Issues and Limitations. Python-Docs. https://docs.python.org/3/tutorial/floatingpoint.html (Zugriff am 2024, Juni 06)