Inhaltsverzeichnis

In dieser Ausarbeitung geht es darum Matrizenmultiplikation effizient zu gestalten. Das heißt, dass das Ergebnis einer Matrizenmultiplikation mit möglichst wenigen arithmetischen Operationen errechnet werden soll. Ein Algorithmus, der eine bessere Laufzeit, also weniger Multiplikationen, als der „naive“ Algorithmus besitzt, ist der „Strassen Algorithmus“, welcher in dieser Arbeit genauer betrachtet wird. Zunächst wird jedoch der „naive“ Algorithmus charakterisiert.

1 Einleitung

Die normale Matrizenmultiplikation, eben gerade den „naiven“ Algorithmus, lernt man bereits teilweise in der Schule, spätestens jedoch im Studium einer Naturwissenschaft. Wir betrachten zunächst die Multiplikation zweier 2×2 Matrizen.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix}$$

Um also beispielsweise in unserer Ergebnismatrix den Eintrag, der in der *ersten* Zeile und der *zweiten* Spalte steht, zu errechnen, berechnen wir das paarweise Produkt der Einträge aus der *ersten* Zeile von der linken Matrix mit den Einträgen der *zweiten* Spalte von der rechten Matrix und addieren diese.
Für jeden Eintrag der Ergebnismatrix benötigen wir offensichtlich zwei Multiplikationen und eine Addition. Da in der Ergebnismatrix vier Einträge stehen, sind das also insgesamt 8 Multiplikationen und 4 Additionen.
Betrachten wir im Folgenden den allgemeinen Fall.
Gegeben sei eine $l \times m$ Matrix $A = (a_{ij})_{i=1\ldots l,\ j=1\ldots m}$ und eine $m \times n$ Matrix $B = (b_{ij})_{i=1\ldots m,\ j=1\ldots n}$.

$$A = \begin{pmatrix} a_{11} & \ldots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{l1} & \ldots & a_{lm} \end{pmatrix} \qquad B = \begin{pmatrix} b_{11} & \ldots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \ldots & b_{mn} \end{pmatrix}$$

Das Ergebnis der Multiplikation beider Matrizen (man beachte, dass die Spaltenanzahl der linken Matrix mit der Zeilenanzahl der rechten Matrix übereinstimmen muss) ist dann eine $l \times n$ Matrix $C = A \cdot B = (c_{ij})_{i=1\ldots l,\ j=1\ldots n}$.

$$C = \begin{pmatrix} c_{11} & \ldots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{l1} & \ldots & c_{ln} \end{pmatrix}$$

Der Eintrag, der $i-$ten Zeile und $j-$Spalte dieser Matrix berechnet sich dann nach dem „naiven“ Algorithmus, wie schon im Beispiel zuvor, aus der Addition der paarweisen Produkte der Einträge der

$i-$ten Zeile von der linken Matrix A mit den Einträgen der $j-$ten Spalte von der rechten Matrix B, also

$$c_{ij} = \sum_{k=1}^{m} a_{ik} \cdot b_{kj}.$$

Wie man leicht sieht benötigen wir m Multiplikationen und $m-1$ Additionen je Eintrag. Da in unserer Ergebnismatrix C $l \cdot n$ Einträge vorhanden sind, ergibt sich also insgesamt eine Anzahl von $l \cdot n \cdot m$ Multiplikationen und $l \cdot n \cdot (m-1)$ Additionen damit wir C errechnen können. Für den Fall, dass wir zwei quadratische Matrizen multiplizieren, also $l = m = n$ gilt, werden demzufolge n^3 Multiplikationen und $n^3 - n^2$ Additionen benötigt. Daraus ergibt sich eine Laufzeit von $\mathcal{O}(n^3)$.
Fassen wir am Ende der Einleitung kurz zusammen was wir erreicht haben.
Wir besitzen einen Algorithmus, der fehlerfrei das tut, was vom „Benutzer“ gewünscht ist, und zwar die Multiplikation zweier Matrizen, wir beschränken uns im Folgenden auf den Fall quadratischer Matrizen. Dieser Algorithmus benötigt mit größer werdenden Matrizen immer mehr Zeit und Speicherplatz, da n^3 (Anzahl der benötigten Multiplikationen) sehr schnell wächst.
So führen zum Beispiel Techniken zur Lösung partieller Differentialgleichungen häufig auf riesige lineare Gleichungssysteme. Die Lösung eines solchen Gleichungssystems erfahren wir beispielsweise jeden Abend beim Wetterbericht. Der „naive“ Algorithmus würde für die Multiplikation zweier 723×723 Matrizen ganze 3.778.339.067 Multiplikationen benötigen.
Da stellt sich natürlich die Frage, ob es einen besseren Algorithmus gibt, der auf das gleiche Ergebnis kommt, aber dafür weniger Zeit als der „naive“ Algorithmus in Anspruch nimmt, damit man beispielsweise große Gleichungssysteme viel effizienter lösen kann.
Tatsächlich gibt es bessere Algorithmen, die eine schnellere Laufzeit aufweisen können. Der Strassen Algorithmus benötigt für die Multiplikation zweier quadratischer Matrizen anstatt einer Laufzeit von $\mathcal{O}(n^3)$, nur eine Laufzeit von $\mathcal{O}(n^{2,807})$, daher werden wir uns diesen Algorithmus im nächsten Kapitel genauer zuwenden.

2 Der Algorithmus von Strassen

Dieses Kapitel ist in drei Unterkapitel eingeteilt. Als erstes wird die Grundidee des Algorithmus' vorgestellt und danach eine Möglichkeit, wie Volker Strassen diesen entdeckt haben könnte. Es kann nur eine *Möglichkeit* vorgestellt werden, weil bis heute nicht bekannt ist, wie Strassen diesen Algorithmus fand, allerdings äußerten Thomas H. Cormen, Clifford Stein, Charles E. Leiserson und Robert L. Rivest, die

Autoren des Buches *„Introduction to Algorithms“*, eine Vermutung, welche im zweiten Unterkapitel geschildert wird. Im dritten Unterkapitel geht es dann um die Laufzeit des Algoritmus'. Zunächst folgen allerdings ein paar Informationen zu dem Entdecker.
Volker Strassen wurde im April 1936 in Düsseldorf geboren, ist demzufolge deutscher Mathematiker, der heute in Dresden wohnt. Er studierte Musik, Philosophie, Physik und Mathematik an verschiedenen deutschen Universitäten. 1966 habilitierte er sich in Erlangen. Seitdem hat er sich viel mit informatischen und mathematischen Themen auseinandergesetzt, beispielsweise mit der Komplexitätstheorie oder der Wahrscheinlichkeitstheorie. Seine Abhandlung *„Gaussian Elimination is not optimal“*, die bis heute als Pionierarbeit im Bereich der Algorithmik gilt, veröffentlichte er 1969. In ihr stellte er erstmals den Strassen-Algorithmus zur Multiplikation zweier quadratischer Matrizen vor. Seitdem veröffentlichte er noch weitere Arbeiten, wie beispielsweise den *„Schönhage-Strassen-Algorithmus“*, der bis heute schnellste, praktisch eingesetzte Algorithmus zur Multiplikation großer ganzer Zahlen oder den *„Solovy-Strassen-Test“*, ein Algorithmus, der prüft ob eine Zahl prim ist oder nicht.

2.1 Die Grundidee

Der Strassen Algorithmus ist ein rekursiver Algorithmus, der auf dem *„Divide and Conquer“* Prinzip basiert. Es wird wie folgt vorgegangen

1. Das Problem wird in möglichst gleich große Teilprobleme, derselben Art wie das Originalproblem aufgeteilt. Diese Teilprobleme können mit demselben Algorithmus unabhängig voneinander gelöst werden.

2. Die Lösungen der Teilprobleme werden zu einer Lösung des Originalproblems kombiniert.[1]

Die Aufteilung der Teilprobleme erfolgt so lange, bis eine Lösung dieser trivial ist und sofort erfolgen kann. Der einfachste Fall der „echten“ Matrizenmultiplikation (wir schließen den skalaren Fall $n = 1$ aus) ist die Multiplikation zweier 2×2 Matrizen A und B. Diesen Fall haben wir bereits im Anfangsbeispiel betrachtet:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix} =: \begin{pmatrix} c_1 & c_2 \\ c_3 & c_4 \end{pmatrix}$$

Wenn wir die Multiplikation so ausführen, dann benötigen wir 8 Multiplikationen und 4 Additionen. Wir können aber auch folgendes machen.

[1] „Introduction to Algorithms“, Thomas H. Cormen, Clifford Stein, Charles E. Leiserson Robert L. Rivest

Wir definieren die Hilfsgrößen

$$\begin{aligned}
P_1 &:= a \cdot (f - h) \\
P_2 &:= (a + b) \cdot h \\
P_3 &:= (c + d) \cdot e \\
P_4 &:= d \cdot (g - e) \\
P_5 &:= (a + d) \cdot (e + h) \\
P_6 &:= (b - d) \cdot (g + h) \\
P_7 &:= (a - c) \cdot (e + f).
\end{aligned}$$

Hierfür benötigen wir 7 Multiplikationen und 10 Additionen. Allerdings sind wir noch nicht beim gewünschten Ergebnis. Erst wenn wir

$$c_1 = ae + bg \qquad c_2 = af + bh \qquad c_3 = ce + dg \qquad c_4 = cf + dh$$

haben, sind wir fertig. Durch geschickte Addition der Hilfsgrößen, kommen wir aber ganz einfach auf das Ergebnis.

$$\begin{aligned}
c_1 &= P_5 + P_4 - P_2 + P_6 = (a + d) \cdot (e + h) + d \cdot (g - e) - (a + b) \cdot h + (b - d) \cdot (g + h) \\
&= ae + de + ah + dh + dg - de - ah - bh + bg - dg + bh - dh = ae + bg \\
c_2 &= P_1 + P_2 = a \cdot (f - h) + (a + b) \cdot h = af - ah + ah + bh = af + bh \\
c_3 &= P_3 + P_4 = (c + d) \cdot e + d \cdot (g - e) = ce + de + dg - de = ce + dg \\
c_4 &= P_5 + P_1 - P_3 - P_7 = (a + d) \cdot (e + h) + a \cdot (f - h) - (c + d) \cdot e - (a - c) \cdot (e + f) \\
&= ae + ah + de + dh + af - ah - ce - de - ae + ce - af + cf = dh + cf = cf + dh
\end{aligned}$$

Wir haben also das Produkt der beiden Matrizen gefunden und dabei nur 7 Multiplikationen und 18 Additionen benötigt. Wir sparen immerhin zum „naiven“ Algorithmus eine Multiplikation, aber wir bekommen dafür 14 Additionen mehr. Lohnt sich das? Nun, damals haben Rechner für das Ausführung von Multiplikationen viel länger gebraucht, als für das Ausführen von Additionen, trotzdem lohnt es sich für diesen Fall natürlich noch nicht.

Die besondere Leistung von Strassen war jedoch, dass bei der Berechnung der c_i's und P_i's nur die Distributivität der Elemente und die Kommutativität der Additionen genutzt wurde, *nicht* die Kommutativität der Multiplikation. Strassen erkannte, dass die Berechnungsformeln auch gelten, wenn die Elemente a bis h selbst Matrizen sind, wodurch Rekursion möglich ist. Bei der Berechnung der P_i's, also speziell bei den 7 Multiplikationen, wird der Strassen Algorithmus nochmal angewendet und die

Matrixgröße solange „heruntergebrochen“ bis eine Größe erreicht ist, bei der es sich lohnt den „naiven“ Algorithmus zu verwenden.

2.2 Strassen´s Methode?

Nun kommen wir zur erwähnten Möglichkeit, wie Strassen diesen Algorithmus gefunden haben könnte. Strassen´s Vorgehensweise beruht auf 4 Schritten:

1. Teile die Eingabematrizen A und B in $\frac{n}{2} \times \frac{n}{2}$ Teilmatrizen
2. Berechne 14 Hilfsmatrizen A_1, B_1, A_2, B_2, $\ldots A_7$, B_7 von denen jede eine $\frac{n}{2} \times \frac{n}{2}$ Matrix ist.
3. Berechne die sieben Matrixprodukte $P_i = A_i \cdot B_i$
4. Berechne die benötigten Teilmatrizen c_i und die resultierende Matrix C durch Addition und/oder Subtraktion verschiedener Kombinationen der P_i-Matrizen.[2]

Wir schauen uns im Folgenden etwas genauer an, wie Strassen die Teilmatrixprodukte P_i gefunden haben könnte. Zunächst etwas zur Übersicht.
Die Teilmatrix $c_1 = ae+bg$, lässt sich folgendermaßen umformen (wir verwenden der Einfachheithalber 4×4 Matrizen):

$$\begin{pmatrix} a & b & c & d \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} = \begin{pmatrix} a & 0 & b & 0 \end{pmatrix} \cdot \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} = ae + bg$$

Wir führen nun eine abkürzende Schreibweise ein.

$$\begin{pmatrix} a & b & c & d \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} e \\ f \\ g \\ h \end{pmatrix} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} e & f & g & h \end{array} \\ \begin{pmatrix} + & . & . & . \\ . & . & + & . \\ . & . & . & . \\ . & . & . & . \end{pmatrix} \end{array} = \begin{pmatrix} + & . & . & . \\ . & . & + & . \\ . & . & . & . \\ . & . & . & . \end{pmatrix}$$

[2] „Introduction to Algorithms“, Thomas H. Cormen, Clifford Stein, Charles E. Leiserson Robert L. Rivest

Das + steht für +1, ein − für −1 und ein Punkt bedeutet 0. Für die anderen Teilmatrizen gilt

$$c_2 = af + bh = \begin{pmatrix} . & + & . & . \\ . & . & . & + \\ . & . & . & . \\ . & . & . & . \end{pmatrix}, \qquad c_3 = ce + dg = \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ + & . & . & . \\ . & . & + & . \end{pmatrix}, \qquad c_4 = cf + dh = \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ . & + & . & . \\ . & . & . & + \end{pmatrix}.$$

Jede dieser Teilmatrizen benötigt zwei Multiplikationen. Das liegt daran, dass die Plus immer in unterschiedlichen Zeilen und Spalten sind. Legen wir die 4 Teilmatrizen übereinander, so ergibt sich

$$\begin{pmatrix} + & + & . & . \\ . & . & + & + \\ + & + & . & . \\ . & . & + & + \end{pmatrix}.$$

Jedes der Plus steht für einen Term, die wir für unsere Berechnungen brauchen: $ae, af, bg, bh, ce, cf, dg, dh$. Diese nennen wir im Folgenden *wesentliche* Terme.

Unsere Teilmatrizen benötigen jeweils, wie schon erwähnt, zwei Multiplikationen. Zwei Matrizen, die nur eine Multiplikationen brauchen, wären zum Beispiel

$$(a+b)h = ah + bh = \begin{pmatrix} . & . & . & + \\ . & . & . & + \\ . & . & . & . \\ . & . & . & . \end{pmatrix} \text{ oder } \qquad a(g-h) = ag - ah = \begin{pmatrix} . & . & + & - \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \end{pmatrix}.$$

Hier stehen die Plus, bzw. Minus jeweils in einer Zeile oder Spalte. Jetzt stellt sich die Frage, ob es möglich ist Matrizen zu finden, die alle 8 benötigten *wesentlichen* Terme abdecken, gleichzeitig unsere c_i errechnen und dabei weniger als 8 Multiplikationen benötigen.

Wir beginnen unsere Suche mit der Feststellung, dass die Teilmatrix c_2 aus $P_1 + P_2$ berechnet werden kann, wobei P_1 und P_2 jeweils eine Matrizenmultiplikation benötigen.

$$P_1 = A_1 \cdot B_1 = a(f-h) = af - ah = \begin{pmatrix} . & + & . & - \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \end{pmatrix}, \; P_2 = A_2 \cdot B_2 = (a+b) \cdot h = ah + bh = \begin{pmatrix} . & . & . & + \\ . & . & . & + \\ . & . & . & . \\ . & . & . & . \end{pmatrix}$$

$$c_2 = P_1 + P_2 = af - ah + ah + bh = af + bh = \begin{pmatrix} . & + & . & . \\ . & . & . & + \\ . & . & . & . \\ . & . & . & . \end{pmatrix}$$

Ähnlich dazu, lässt sich c_3 aus P_3 und P_4 berechnen.

$$c_3 = ce + dg = \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ + & . & . & . \\ . & . & + & . \end{pmatrix} = \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ + & . & . & . \\ + & . & . & . \end{pmatrix} + \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ - & . & + & . \end{pmatrix} =$$

$$= ce + de + dg - de = (c+d) \cdot e + d \cdot (g-e) = A_3 \cdot B_3 + A_4 \cdot B_4 = P_3 + P_4$$

Fassen wir kurz zusammen. Wir haben bereits die Teilmatrizen c_2 und c_3 berechnet. Es existiert P_1, das af berechnet; P_2, das bh berechnet; P_3, das ce berechnet und P_4, das dg berechnet. Wir haben bereits 4 Multiplikationen benutzt.

Uns bleibt noch c_1 und c_4 zu berechnen und damit die *wesentlichen* Terme ae, bg, cf und dg. Dabei dürfen allerdings nurnoch höchstens drei Multiplikationen verwendet werden, damit der Algorithmus schneller arbeitet, als der „Naive“.

Wir führen nun P_5 ein, um die beiden *wesentlichen* Terme ae und dh auf einmal zu berechnen.

$$P_5 = A_5 \cdot B_5 = (a+d) \cdot (e+h) = ae + ah + de + dh = \begin{pmatrix} + & . & . & + \\ . & . & . & . \\ . & . & . & . \\ + & . & . & + \end{pmatrix}$$

Es wurde ae und dh berechnet, allerdings müssen ah und de wieder entfernt werden. Dafür können P_4 und P_2 benutzt werden, die schon berechnet sind.

$$P_5 + P_4 - P_2 = \begin{pmatrix} + & . & . & + \\ . & . & . & . \\ . & . & . & . \\ + & . & . & + \end{pmatrix} + \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ - & . & + & . \end{pmatrix} - \begin{pmatrix} . & . & . & + \\ . & . & . & + \\ . & . & . & . \\ . & . & . & . \end{pmatrix} =$$

$$= \begin{pmatrix} + & . & . & . \\ . & . & . & - \\ . & . & . & . \\ . & . & + & + \end{pmatrix} = ae - bh + dg + dh$$

Addieren wir auf dieses Teilergebnis ein zusätzliches Produkt, nämlich

$$P_6 = A_6 \cdot B_6 = (b-d)\cdot(g+h) = bg + bh - dg - dh = \begin{pmatrix} . & . & . & . \\ . & . & + & + \\ . & . & . & . \\ . & . & - & - \end{pmatrix}$$

, so erhalten wir

$$P_5 + P_4 - P_2 + P_6 = \begin{pmatrix} + & . & . & . \\ . & . & . & - \\ . & . & . & . \\ . & . & + & + \end{pmatrix} + \begin{pmatrix} . & . & . & . \\ . & . & + & + \\ . & . & . & . \\ . & . & - & - \end{pmatrix} = \begin{pmatrix} + & . & . & . \\ . & . & + & . \\ . & . & . & . \\ . & . & . & . \end{pmatrix} = ae + bg = c_1.$$

Wir haben c_1 mit zwei weiteren Multiplikationen berechnet. Für c_4 steht uns nur noch eine Multiplikation zur Verfügung.

Um c_4 zu errechnen, werden zunächst P_5, P_1 und P_3 verwendet. Danach subtrahieren wir davon ein zusätzliches Produkt

$$P_7 = A_7 \cdot B_7 = (a-c)\cdot(e+f) = ae + af - ce - cf = \begin{pmatrix} + & + & . & . \\ . & . & . & . \\ - & - & . & . \\ . & . & . & . \end{pmatrix}.$$

Wir erhalten

$$P_5 + P_1 - P_3 - P_7 = \begin{pmatrix} + & . & . & + \\ . & . & . & . \\ . & . & . & . \\ + & . & . & + \end{pmatrix} + \begin{pmatrix} . & + & . & - \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \end{pmatrix} - \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ + & . & . & . \\ + & . & . & . \end{pmatrix} - \begin{pmatrix} + & + & . & . \\ . & . & . & . \\ - & - & . & . \\ . & . & . & . \end{pmatrix}$$

$$= \begin{pmatrix} . & . & . & . \\ . & . & . & . \\ . & + & . & . \\ . & . & . & + \end{pmatrix} = cf + dh = c_4.$$

Damit haben wir alle Teilmatrizen c_i, durch P_i beschrieben und dabei nur 7 Multiplikationen ausgeführt!

3 Laufzeit

Wir beweisen zunächst zwei Lemmata, die wir später benötigen.

Lemma 1:

$$\text{Für } n = 2^k \text{ ist } 7^{\log_2 n} = n^{\log_2 7}.$$

Beweis:

$$7^{\log_2 n} = 7^{\log_2 2^k} = 7^k = 2^{k \cdot \log_2 7} = \left(2^k\right)^{\log_2 7} = n^{\log_2 7} \qquad \square$$

Lemma 2:

$$\text{Für } n = 2^k \text{ ist } \frac{18n^2}{7} \cdot \sum_{i=1}^{\log_2 n} \left(\frac{7}{4}\right)^i = 6n^{\log_2 7} - 6n^2.$$

Beweis: Wir benutzen den Wert der geometrischen Reihe und formen dann um.

$$\begin{aligned}
\frac{18n^2}{7} \cdot \sum_{i=1}^{log_2 n} \left(\frac{7}{4}\right)^i &= \frac{18n^2}{7} \cdot \sum_{i=0}^{log_2 n} \left(\frac{7}{4}\right)^i - \frac{18n^2}{7} = \\
&= \frac{18n^2}{7} \cdot \frac{1 - \left(\frac{7}{4}\right)^{\log_2(n)+1}}{1 - \frac{7}{4}} - \frac{18n^2}{7} \\
&= -\frac{24n^2}{7} \cdot \left(1 - \left(\frac{7}{4}\right)^{\log_2(n)+1}\right) - \frac{18n^2}{7} = \\
&= -\frac{24n^2}{7} + \frac{24n^2 \cdot 7^{\log_2(n)+1}}{4^{\log_2(n)+1}} - \frac{18n^2}{7} = \\
&= -\frac{42}{7}n^2 + 6 \cdot 7^{\log_2 n} \cdot \underbrace{\frac{n^2}{4^{\log_2 n}}}_{=1, \text{ für } n=2^k} = \\
&= 6n^{\log_2 7} - 6n^2 \qquad \square
\end{aligned}$$

Es sei $n = 2^k$ und $T(n)$ die Anzahl der skalaren arithmetischen Operationen, die Strassen´s Algorithmus zur Multiplikation zweier $n \times n$ Matrizen benötigt. Wie wir gesehen haben, gibt es 7 Multiplikationen, die ihrerseits durch Rekursion wieder in einer Laufzeit von $T\left(\frac{n}{2}\right)$ liegen. Des Weiteren gibt es Additionen (einschließlich Subtraktionen), die in der Größenordnung n^2 liegen. Dann gilt

$$\begin{aligned}
T(1) &= 1 \\
T(n) &= 7 \cdot T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2.
\end{aligned}$$

Wir führen nun Iteration durch.

$$
\begin{aligned}
T(n) &= 7 \cdot T\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 = 7\left(7 \cdot T\left(\frac{n}{4}\right) + 18\left(\frac{n}{4}\right)^2\right) + 18\left(\frac{n}{2}\right)^2 = \\
&= 7^2 \cdot T\left(\frac{n}{2^2}\right) + 7 \cdot 18\left(\frac{n}{2^2}\right)^2 + 18\left(\frac{n}{2}\right)^2 = \\
&= 7^3 \cdot T\left(\frac{n}{2^3}\right) + 7^2 \cdot 18\left(\frac{n}{2^3}\right)^2 + 7 \cdot 18\left(\frac{n}{2^2}\right)^2 + 18\left(\frac{n}{2}\right)^2 = \ldots \\
&= 7^{\log_2 n} \cdot T(1) + \sum_{i=1}^{log_2 n} 7^{i-1} \cdot 18\left(\frac{n}{2^i}\right)^2 \\
&= 7^{\log_2 n} \cdot T(1) + 6 \cdot 7^{\log_2 n} - 6n^2 \qquad \text{(Lemma 2)} \\
&< 7^{\log_2 n} \cdot T(1) + 7^{\log_2 n} \cdot 6 \\
&= 7 \cdot n^{\log_2 7} \approx 7 \cdot n^{2{,}807} \qquad \text{(Lemma 1)}
\end{aligned}
$$

Die Anzahl der skalaren arithmetischen Operationen zur Multiplikation zweier $n \times n$ Matrizen nach Strassen´s Methode ist also proportional zu $n^{2,807}$. Daraus ergibt sich eine Laufzeit von $\mathcal{O}\left(n^{2,807}\right)$.

4 Diskussion

Dieser Algorithmus gilt erstmal nur für $n = 2^k$, weil wir in der Rekursion die Dimension der Matrix immer halbieren. Mit diesem Algorithmus kann man aber auch quadratische Matrizen mit $n \neq 2^k$ berechnen, indem man einfach soviele Nullzeilen und -spalten in die Matrix einfügt, sodass eine neue Matrix mit einer Dimension von 2^k entsteht. Im Ergebnis muss man dann entsprechende Zeilen und/oder Spalten löschen.

In der Praxis ist der Algorithmus nur bei sehr großen Matrizen von Vorteil, da er zwar Multiplikationen einspart, aber dafür Additionen hinzukommen, die auch wieder Speicherplatz, Buchhaltung und Rechenzeit benötigen. Der Faktor $\frac{n^3}{n^{2,81}} \approx n^{0,2}$ wächst zu langsam. Zum Beispiel sei $n = 1024$, dann ist $n^{0,2} = 4$, was aber wohl kaum von praktischer Relevanz ist.

Hinzu kommt, dass die numerische Stabilität verringert wird. Das liegt an den vielen Rechneoperationen (wir haben insgesamt, durch die Additionen mehr Rechenoperationen als bei dem „naiven“ Algorithmus) und den Rundungen, die rechnerintern stattfinden, woraus dann mehr Fehler entstehen können. Die bei der Rekursion gebildeten Teilmatrizen können außerdem viel Speicherplatz verbrauchen. Der Strassen Algorithmus ist für dünn besetzte Matrizen (das sind Matrizen mit vielen Nullen) nicht optimal. Es gibt für diese bessere Algorithmen.[3] Des Weiteren verlangt der Algorithmus einen erhöhten Programmieraufwand, was in diesem Sinne auch Zeit kostet.

[3] „Introduction to Algorithms“, Thomas H. Cormen, Clifford Stein, Charles E. Leiserson Robert L. Rivest

Einige Argumente konnten allerdings im Laufe der Zeit relativiert werden, so hat *Higham* 1990 gezeigt, dass der Unterschied zur numerischen Stabilität des „naiven“ Algorithmus nicht so gravierend ist, wie vermutet und gleichzeitig fand *Bailey* Methoden zur Verringerung des Speicherbedarfs von Strassen´s Algorithmus.

Der Algorithmus von Strassen galt lange nur als theoretisch interessantes Beispiel zur Komplexitätsreduktion bei der Matrizenmultiplikation, weil es keine einfache Aufgabe ist, aus dem Grundkonzept des Algorithmus effiziente Programme zu entwickeln. *C. C. Douglas*, *M. Heroux*, *G. Slishman* und *R. M. Smith* ist es schließlich doch gelungen den Algorithmus effizient zu implementieren. Sie konnten effiziente Programme für Einprozessor-Computer und Parallelrechner entwickeln. Sie haben auch herausgefunden, dass die „Grenzdimension“, ab der eine Strassenimplementierung weniger Rechenzeit erfordert als die naive Methode, sehr systemabhängig ist. Sie liegt zwischen 32 und 256. Bei Matrizen, die diese Grenzdimension überschreiten, tritt dann eine signifikante Verringerung der erforderlichen Multiplikationen und eine entsprechende Verkürzung der Rechenzeit ein.

Es existieren heute allerdings noch schnellere Algorithmen für die Matrixmultiplikation. Der theoretisch schnellste bekannte Algorithmus ist von *Coppersmith* und *Winograd* (1987) mit einer Laufzeit von $\mathcal{O}\left(n^{2,3755}\right)$. Dieser wurde über die Zeit minimal verbessert, unter anderem von *Andrew Stothers*, der die Laufzeit auf $\mathcal{O}\left(n^{2,3736}\right)$ reduzierte. Im Jahr 2011 optimierte *Virginia Williams* die Idee von *Stothers* auf eine Laufzeit von $\mathcal{O}\left(n^{2,3727}\right)$. *Henry Cohn, Robert Kleinberg, Balázs Szegedy* und *Christopher Umans* zeigten, dass unter der Annahme einer von zwei verschiedenen Vermutungen der optimale Exponent zwei ist (was auch vermutet wurde), konnten dies aber noch nicht beweisen.

5 Quellen

- „Introduction to Algorithms“, Thomas H. Cormen, Clifford Stein, Charles E. Leiserson Robert L. Rivest
- `http://www.michaelbiebl.de/files/12_matrizen.pdf`
- `http://cvpr.uni-muenster.de/teaching/ss06/info2/script/InfoII-Kapitel_3.pdf`