Institut für Parallele und Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3267

Implementierung und Anwendung von Laplacian-Eigenmap Verfahren

Ramzy Saleh

Studiengang:	Informatik
Prüfer:	Prof. Dr. Marc Alexander Schweitzer
Betreuer:	Dr. Stefan Zimmer
begonnen am:	02. November 2011
beendet am:	03. Mai 2012
CR-Klassifikation:	G.1.3, G.2.2

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Matrix-Matrix-Multiplikation für Matrizen $A, B, C \in \mathbb{C}^{N \times N}$ sollte für jeden, der in der Schule Mathematik gelernt hat, keine große Herausforderung darstellen. Man erhält einen Eintrag in C z.B C_{ij}, indem man die i-te Zeile aus A mit der j-ten Spalte aus B multipliziert. Dieses Vorgehen ist leicht nachvollziehbar, wenn man die Matrizen von Hand multiplizieren muss. Mit dem Rechner stellt sich aber nicht nur die Frage, wie einfach die Matrix-Matrix-Multiplikation zu implementieren ist, sondern es kommen noch viele weitere Fragen hinzu. Zum Beispiel wie groß ist der Cache, in dem die Elemente der Matrizen während des Multiplikationsverlaufs gespeichert werden? Wie oft werden die Matrixeinträge aus dem Hauptspeicher in den Cache geholt? Können solche Zugriffe auf den Hauptspeicher während des Multiplikationsverlaufs reduziert werden?

Solche Überlegungen und Fragestellungen werde ich in dieser Arbeit tiefer untersuchen und aufbauend auf den Laplacian Eigenmap Verfahren [BN03] eine Multiplikationsvariante $\pi_{Laplace}$ erstellen, die eine gute Cache-Ausnutzung besitzt.

Zum Vergleich werde ich zwei andere bekannte Multiplikationsvarianten, $\pi_{Standard}$ und π_{Peano} [BZ06], mit einbeziehen. Die erste $\pi_{Standard}$ ist die ganz normale Multiplikation, die A zeilenweise und B spaltenweise durchläuft. Die zweite π_{Peano} ist eine Variante mit sehr guter Cache-Ausnutzung[1] und beruht auf der Peano-Kurve[2].

Als erstes wollen wir aber eine Methode des Laplacian-Eigenmap-Verfahrens [BN03] vorstellen, welche im Folgenden als LE bezeichnet wird. Durch LE erhält man als Ausgabe einen Pfad der Länge k. Dieser Pfad verläuft durch die Punktmenge $M := \{x_1, x_2 \dots x_k : x_i \in \mathbb{R}^l\}$ (Eingabe von LE) so, dass die Elemente von M, die in $\mathbb{R}^l$ nah zueinander liegen, miteinander verbunden werden.

Weiterhin betrachten wir die Operationen der Multiplikation $C = A * B$ für $A, B, C \in \mathbb{C}^{N \times N}$ als Koordinaten in $\mathbb{R}^3$. So steht z.B. für die Operation $C_{ij} += A_{ik} * B_{kj}$ die Koordinate $(i, j, k) \in \mathbb{R}^3$. Unser Ziel ist es, LE auf diese Koordinaten anzuwenden, so dass in dem entstehenden Pfad Operationen nebeneinander stehen, die die gleichen Operanden besitzen.

[1] Mit Cache-Ausnutzung ist der Durchschnitt für verschiedene Cache-Größen gemeint.
[2] http://de.wikipedia.org/wiki/Peano-Kurve

Dieses Vorgehen soll dafür sorgen, dass die ausgeführten Operationen ihre Operanden meist im Cache finden. Dadurch wird der wiederholte Zugriff auf den Hauptspeicher während des Multiplikationsverlaufs reduziert.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Einführung: Hier wird das LE-Verfahren, erklärt und implementiert und durch verschiedene Beispiele von Punktmengen $\in \mathbb{R}^2$ erläutert.

Kapitel 3 – Matrix-Matrix-Multiplikation: In diesem Teil geht es allgemein um die Matrix-Matrix-Multiplikation für Matrizen $\in \mathbb{C}^{N \times N}$. Es wird auf die Multiplikationsvariante eingegangen, wobei zwei verschiedene Varianten vorgestellt werden.

Kapitel 4 – Analysewerkzeug: Vorstellung des Analysetools, das ich für Darstellung und Analyse der Ergebnisse implementiert habe.

Kapitel 5 – Laplace-Multiplikation: Dies ist der Hauptteil dieser Arbeit. Hier wird eine neue Multiplikationsvariante vorgestellt, die durch LE erzeugt wird. Diese Variante wird dann am Ende auf Cache-Ausnutzung untersucht.

Kapitel 6 – Zusammenfassung und Ausblick: Dieses Kapitel fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2 Einführung

In diesem Kapitel wird eine diskrete Methode des Laplacian-Eigenmap-Verfahrens erklärt. [BN03]. Diese Methode, die wir im Folgenden als LE benennen, liefert einen Eigenvektor zum kleinsten nichttrivialen Eigenwert eines diskreten Laplaceoperator, der räumliche Entfernungen zwischen Punkten aus $\mathbb{R}^l$ beschreibt. Mittels des Eigenvektors wird dann eine Ordnung auf diesen Punkten eingeführt.

Für die Implementierung des Verfahrens wird die Programmiersprache Python[1] sowie die Bibliothek NumPy[2] [JOP$^+$] verwendet. Alle in dieser Arbeit vorkommenden Quellcode-Beispiele sind in Python geschrieben und werden mit Hilfe von minted[3][Rud11] dargestellt. Am Ende wird LE anhand einiger Beispiele mit verschiedenen Punktmengen aus $\mathbb{R}^2$ verdeutlicht.

In diesem Kapitel handelt es sich bei allen Quellcode-Beispielen nicht unbedingt um eine effiziente Implementierung des LE-Verfahrens. Es werden nur bestimmte Beispielprobleme gerechnet, die zum Schluss dieses Kapitels zu sehen sind.

2.1 Laplacian-Eigenmap-Verfahren

Gegeben sind k Punkte $x_1, x_2, ..., x_k$ in $\mathbb{R}^l$. Diese Punkte werden als Knoten eines Graphen G betrachtet. Den Punkt x_i bezeichnen wir manchmal auch als v_i, als einen Knoten aus dem Graphen G. Für die Punktmenge $M := \{x_1, x_2, .., x_k : x_i \in \mathbb{R}^l\}$ erhalten wir den gewichteten Graph $G := (V, E)$ mit $V := \{v_1, v_2, ..., v_k\}$ und $E = V \times V$ als die Menge der Kanten und deren Gewichtsfunktion $\delta : E \to \mathbb{R}_0$.

Für zwei Knoten $v, w \in V$ gilt $e_{vw} \in E$ und $e_{wv} \in E$. Nach dieser Definition besitzt G stets k^2 Kanten. Es soll an dieser Stelle jedoch darauf hingewiesen werden, dass viele $e \in E$ das Gewicht 0 haben dürfen. Z.B. $\forall v \in V$ gilt $\delta(e_{vv}) = 0$ und es gilt auch $\delta(e_{vw}) = 0$, falls die Knoten v und w nicht benachbart sind.

[1] http://www.python.org/
[2] http://numpy.scipy.org/
[3] http://code.google.com/p/minted/

Um auf die Punkte $x_1, x_2, .., x_k$ und ihre euklidischen Abstände schnell zugreifen zu können, definieren wir die Differenz Matrix $Diff \in \mathbb{R}^{k \times k}$ mit $Diff_{ij} = Diff_{ji} = \|x_i - x_j\|^2 : x_i, x_j \in \{x_1, x_2, .., x_k\}$.

2.1.1 Die Gewichtsfunktion δ

Sind v_i und v_j zwei verschiedene Knoten in G, dann sind v_i und v_j benachbart bzw. deren Gewicht $\delta(e_{ij}) = \delta_{ij} > 0$, falls:

1. Variante 1 verbindet[4] Knoten miteinander, deren Abstand kleiner als ϵ ist.

$$Diff_{ij} < \epsilon : [Parameter\ \epsilon \in \mathbb{R}]. \tag{2.1}$$

```
def Is_Epsilon_neighborhoods(Epsilon,i,j):
    # Die Matrix Diff ist global definiert.
    global Diff
    # return (Diff_ij < ε) ∧ (Diff_ij > 0.0)
    return Diff[i,j]<Epsilon and Diff[i,j]>0.0
```

Und die Gewichtsfunktion δ:

```
def delta_Epsilon(Epsilon,i,j,t):
    if Is_Epsilon_neighborhoods(Epsilon,i,j):
                    -Diff_ij/t
        # return e          : t ∈ ℝ
        return power(math.e, -Diff[i,j]/t))
    else:
        return 0.0
```

2. Die zweite Variante verbindet x_i nur mit den ersten $n \in \mathbb{N}$ Punkten, die den kleinsten Abstand zu x_i haben. Ist x_j unter diesen n Punkten, dann werden v_i und v_j durch eine Kante verbunden.

[4]Mit verbunden ist gemeint, dass die Kanten ein Gewicht $\delta_{ij} > 0$ haben.

```
def Is_n_nearst_neighbors(n,i,j):
    k=0
    Zaehler=n
    global Diff
    # X= die i-te Zeile von Diff
    X=Diff[:,i]
    # Mit len bekommt man die Laenge von der Liste X
    while k<len(X) and Zaehler>0:
        if k!=i and k!=j and X[k]<Diff[i,j]:
            Zaehler=Zaehler-1
            k=k+1
    return Zaehler>0
```

Durch die Zeile[5] $Diff[:,i]$ erhält man die Abstände zu x_i. So können alle Abstände gezählt werden, die größer als $Diff[i,j]$ sind. Am Ende wird nur geprüft, ob diese Anzahl kleiner als n ist oder nicht.

Und die Gewichtsfunktion δ:

```
def delta_n(n,i,j,t):
    if Is_n_nearst_neighbors(n,i,j):
                    -Diff_ij/t
        # return e          : t ∈ ℝ
        return power(math.e, -Diff[i,j]\t))
    else:
        return 0.0
```

Kleinere Abstände sollen mehr Gewicht als größere Abstände erhalten. Das können wir durch den Parameter $t \in \mathbb{R}$ beeinflussen. Siehe Gleichung 2.2.

$$\delta_{ij} = \begin{cases} e^{\left(-\frac{Diff_{ij}}{t}\right)} : t \in \mathbb{R}, & \text{falls } v_i \text{ und } v_j \text{ benachbart sind.} \\ 0, & sonst. \end{cases} \tag{2.2}$$

Sind v_i und v_j zwei verschiedene Knoten in G, dann gilt laut Gleichung 2.2:

1. Für $t \to \infty$ gilt $e^{\left(-\frac{Diff_{ij}}{t}\right)} \to 1$.
2. Für $t \to 0$ gilt $\delta_{ij} = 0 \; \forall e_{ij} \in E$.
3. $(Diff_{ij} > Diff_{mn}) \Rightarrow (\delta_{ij} \leq \delta_{mn}) \; \forall e_{ij}, e_{mn} \in E$. Nah zueinander liegende Punkte erhalten mehr Gewicht als die Punkte, die weit auseinander liegen.

[5] Python-Code: $Diff[:,i]$ ist die i-te Zeile von der Matrix $Diff$.

4. δ ist symmetrisch d.h. $\forall e_{ij} \in E$ gilt $\delta_{ij} = \delta_{ji}$.
5. $\forall e_{ij} \in E$ gilt $0 \leq \delta_{ij} \leq 1$.

2.1.2 Laplacian-Matrix

Aus dem Graphen G kann durch die Gewichtsfunktion δ die Adjazenzmatrix W leicht erzeugt werden.

$$W_{ij} = \delta_{ij} \tag{2.3}$$

ϵ-Variante:

```
def get_W_Epsilon(Epsilon):
   global t
   global Diff
   k=len(Diff)
   W=zeros((k, k))
   for i in range(k):
      for j in range(k):
         W[i,j]=delta_Epsilon(Epsilon,i,j,t)
   return W
```

n-Variante:

```
def get_W_n(n):
   global t
   global Diff
   k=len(Diff)
   W=zeros((k, k))
   for i in range(k):
      for j in range(k):
         W[i,j]=delta_n(n,i,j,t)
   return W
```

Zur Berechnung der Laplacian-Matrix L, siehe Gleichung 2.4, benötigen wir zusätzlich die Diagonal-Matrix $D : D_{ii} = \sum_j W_{ji}$. D erhält in ihrer Diagonale die Spaltensummen[6] von der Matrix W.

$$L = D - W \tag{2.4}$$

Mit der Bibliothek NumPy sieht es wie folgt aus:

[6] Oder Zeilensummen, da W symmetrisch ist.

```
# D_ii = sum_j W_ji
D = diag(W.sum(axis=0))
L = D - W
```

2.1.3 Eigenwerte und Eigenvektoren

Das Ziel des LE-Verfahrens ist, aus der Punktmenge $M := \{x_1, x_2, \ldots, x_k\}$ einen Pfad der Länge k zu finden, der alle Punkte $x_1, x_2, \ldots, x_k$ besucht, und sie so hintereinander einordnet, dass Punkte, die nah zueinander liegen, nebeneinander auf dem Pfad stehen. Das liefert uns das LE-Verfahren, indem es die Eigenvektoren $\mathbf{f}_0, \mathbf{f}_1, \ldots, \mathbf{f}_{k-1}$ von $D^{-1}L$ berechnet, die Folge $\pi = (x_1, x_2 \ldots x_k)$ nach $\mathbf{f}_l$ sortiert und ausgibt.

Der Eigenvektor $\mathbf{f}_l$ gehört dem ersten Eigenwert $\lambda_l \in \{\lambda_0, \lambda_1 \ldots \lambda_{k-1}\}$ an, der größer 0 ist. D.h. $(\forall \lambda_m : \lambda_m > 0) \Rightarrow (\lambda_m \geq \lambda_l)$

Sei $V = \begin{pmatrix} V_0 & V_1 & \ldots & V_{k-1} \end{pmatrix}^T$ ein Vektor in $\mathbb{R}^k$, dann liefert uns die Funktion $\alpha_i(V)$, die Stelle mit dem i-kleinsten Element in V. Das bedeutet $\alpha_0(V)$ entspricht dem Minimum von V und $\alpha_{k-1}(V)$ dem Maximum.
Durch die Funktion α kann man die Reihenfolge $\pi = (x_0, x_1, .., x_{k-1})$ nach einem gegebenen Vektor V sortieren

$$\pi_V = (x_{\alpha_0(V)}, x_{\alpha_1(V)}, .., x_{\alpha_{k-1}(V)}) \tag{2.5}$$

Eine Folge π_V sortiert die Folge π nach dem Vektor V. Ist V_i der kleinste Wert in V, dann entspricht das nullte Element von π_V dem i-ten Element von π d.h. $\pi_V(0) = \pi(i)$ usw.

Die Eigenwerte $\lambda_0, \lambda_1, .., \lambda_{k-1}$ und die zugehörigen Eigenvektoren $\mathbf{f}_0, \mathbf{f}_1, .., \mathbf{f}_{k-1}$ werden durch Lösung des verallgemeinerten Eigenwertproblems

$$L\mathbf{f} = \lambda D\mathbf{f} \tag{2.6}$$

ermittelt.

Man kann an dieser Stelle schnell feststellen, dass die Matrix $L = D - W$ symmetrisch ist, denn wir erhalten die Matrix W aus der symmetrischen Funktion δ (siehe die Gleichung 2.3). Matrix D ist auch symmetrisch, da sie diagonal ist, und die Differenz von zwei symmetrischen Matrizen wiederum symmetrisch ist. Daraus folgt, dass $L = D - W$ symmetrisch ist.

Diese Symmetrie gilt aber nicht für die Matrix $D^{-1}L$. Aufgrund einiger besonderer Eigenschaften symmetrischer Matrizen ist die numerische Lösung des symmetrischen Eigenwertproblems viel schneller und günstiger als bei beliebigen Matrizen. Deswegen

werden wir das Gleichungssystem 2.6 umformen, so dass wir an Stelle von $D^{-1}L$ die symmetrische Matrix $A = C^{-T}LC^{-1}$ erhalten, siehe Gleichung 2.8.

$$C = C^T = \begin{pmatrix} \sqrt{D_{11}} & & 0 \\ & \ddots & \\ 0 & & \sqrt{D_{kk}} \end{pmatrix}$$

Die Umformung und Ermittlung der Eigenvektoren in drei Schritten:

1. Ersetze die Diagonalmatrix D durch C^TC.

$$L\mathbf{f} = \lambda C^T C\mathbf{f} \tag{2.7}$$

2. Setze $\tilde{\mathbf{f}}$ für $C\mathbf{f}$ ein, d.h. $\mathbf{f} = C^{-1}\tilde{\mathbf{f}}$, so erhalten wir aus der Gleichung 2.7.

$$C^{-T}LC^{-1}\tilde{\mathbf{f}} = \lambda\tilde{\mathbf{f}} \tag{2.8}$$

$$A = C^{-T}LC^{-1} = C^{-1}LC^{-1} \tag{2.9}$$

$$C^{-1} = C^{-T} = \begin{pmatrix} \frac{1}{C_{11}} & & 0 \\ & \ddots & \\ 0 & & \frac{1}{C_{kk}} \end{pmatrix}$$

3. Berechne Eigenwerte und Eigenvektoren von A und durch $\mathbf{f}C = \tilde{\mathbf{f}}$ die Eigenvektoren von $D^{-1}L$.

$$\begin{aligned} &A\tilde{\mathbf{f}}_0 = \lambda_0\tilde{\mathbf{f}}_0 && \mathbf{f}_0 = \tilde{\mathbf{f}}_0C^{-1} \\ &A\tilde{\mathbf{f}}_1 = \lambda_1\tilde{\mathbf{f}}_1 && \mathbf{f}_1 = \tilde{\mathbf{f}}_1C^{-1} \\ &A\tilde{\mathbf{f}}_2 = \lambda_2\tilde{\mathbf{f}}_2 && \mathbf{f}_2 = \tilde{\mathbf{f}}_2C^{-1} \\ &... \\ &A\tilde{\mathbf{f}}_{k-1} = \lambda_{k-1}\tilde{\mathbf{f}}_{k-1} && \mathbf{f}_{k-1} = \tilde{\mathbf{f}}_{k-1}C^{-1} \end{aligned}$$

$$0 = \lambda_0 \leq \lambda_1 \leq \lambda_2 \leq .... \leq \lambda_{k-1}$$

Wir interessieren uns nur für den ersten Eigenwert λ_1, der größer Null ist, und seinen Eigenvektor $\mathbf{f}_1 = (v_1, v_2, ..., v_k)$.

Sortiere $\begin{pmatrix} v_1 & x_1 \\ v_2 & x_2 \\ \vdots & \vdots \\ v_k & x_k \end{pmatrix}$ aufsteigend nach $\mathbf{f}_1$ und verbinde dann $x_1, x_2, ..., x_k$ in ihrer neuen Reihenfolge miteinander. Die dadurch entstehende Kurve stellt den Laplacian-Eigenmap der Punkte $x_1, x_2, ..., x_k$ dar.

2.2 Entwurf von LE

Wir betrachten den Punkt $x \in \mathbb{R}^2$ als ein Array der Größe Zwei. Beispielsweise für $x = (1.0, 1.0)$ schreiben wir in Python:

```
x=array([1.0,1.0])
```

Als nächstes erstellen wir aus einer gegebenen Punkt-Liste $M = [x_1, x_2, .., x_k]$ die Differenz-Matrix *Diff*. Dadurch können wir die Matrizen W, D und L berechnen.

```
# Matrix Diff
# M = [x_1, x_2 ... x_k]
k=len(M)
Diff=zeros((k,k))
for i in range(k):
   for j in range(k):
      # Diff_ij = || x_i - x_j ||^2
      Diff[i,j]=norm(M[i]-M[j])
```

Die Erstellung der Matrix W ist abhängig von der Strategie, die man wählt.

```
# W nach zwei Strategien mit ϵ oder n
W=get_W_Epsilon(Epsilon) #oder
# W=get_W_n(n)
```

```
# Diagonal-Matrix D und D^-1
# D_ii = sum_j W_ji
D = diag(W.sum(axis=0))
inverseD = diag(1.0/diagonal(D))
```

```
# Diagonal-Matrix C und C^-1
C = diag(sqrt(diagonal(D)))
inverseC = diag(1.0/diagonal(C))
```

```
# Laplacian Matrix L
L = D - W
```

NumPy hat eine Funktion *dot* für die Matrix-Matrix-Multiplikation. Wir könnten $A = C^{-1}LC^{-1}$ durch $dot(inverseC, dot(A, inverseC))$ berechnen. Da C^{-1} eine Diagonal-Matrix ist, können wir den Rechenaufwand von $2n^3$ auf $2n^2$ reduzieren. In dem folgenden Code entspricht $L[i,:]$ der i-ten Spalte von L und $A[:,i]$ die i-te Zeile von A.

```
# A = C^{-1}LC^{-1}
A = identity(k)
for i in range(k):
   A[i,:] = inverseC[i][i]*L[i,:]
for i in range(k):
   A[:,i] = inverseC[i][i]*A[:,i]
```

Die Funktion *eigh* liefert die Eigenwerte in v und ihre Eigenvektoren in w. Die Eigenwerte sind in v aufsteigend sortiert und zu Eigenwert $v[i]$ gehört der Eigenvektor $w[:,i]$.

```
#Eigenwerte v = [λ_0 ... λ_{k-1}]
#Eigenvektoren w = [f̃_0 ... f̃_{k-1}]
v,w = eigh(A)
# den ersten Eigenvektor  EV1 = f̃_1
EV1 = w[:,1]
```

Durch $\mathbf{f}_1 C = \tilde{\mathbf{f}}_1$ wird aus $\tilde{\mathbf{f}}_1$ der endgültige Eigenvektor $\mathbf{f}_1$ ermittelt.

```
# f_1 = f̃_1 C^{-1}
for i in range(len(w)):
   EV1 = EV1*inverseC[i][i]
```

Um die Punkt-Folge M nach dem Eigenvektor $EV1$ zu sortieren, definieren wir als erstes einen neuen Datentypen, der gleichzeitig einen Wert von $EV1$ und einen Wert von M erhalten kann. Dadurch können jetzt die Elemente von M und die Elemente von $EV1$ als Paare in der Liste *values* gespeichert werden, d.h. $values[i] = (EV1[i], M[i])$. Durch den Befehl $order =' ev'$ wird *values* nach dem Eigenvektor $EV1$ sortiert und das Endergebnis liefert uns *Emap*.

```
dtype = [('ev', float), ('point', list)]
values = []
for i in range(len(M)):
   values.append(EV1[i],M[i])
values = array(values,dtype=dtype)
values = sort(values,order='ev')
Emap = values['point']
```

2.3 Beispiele

Hier sind verschiedene Beispiele zu sehen: Ergebnisse für unterschiedliche $t \in \mathbb{R}_0$ mit ϵ- oder N-Variante.

2.3.1 Koch-Kurve

Das erste Beispiel soll die Koch-Kurve darstellen. Die erzeugten Koordinaten liegen zwischen 10.0 und 11.0 auf der x-Achse und zwischen 0.1 und 0.4 auf der y-Achse. Siehe Abbildung 2.1. Abbildung 2.2 zeigt uns den erstellten Graph für $\epsilon = 0.05$. In diesem Graph werden Knoten miteinander verbunden, deren Abstand kleiner als ϵ ist. Abbildung 2.3 zeigt den zugehörigen Eigenmap[7] für den ersten Eigenvektor mit $t = 0.003$. In Abbildung 2.4 sieht man die Ergebnisse bei der Anwendung der Eigenvektoren 0, 2 und 3.

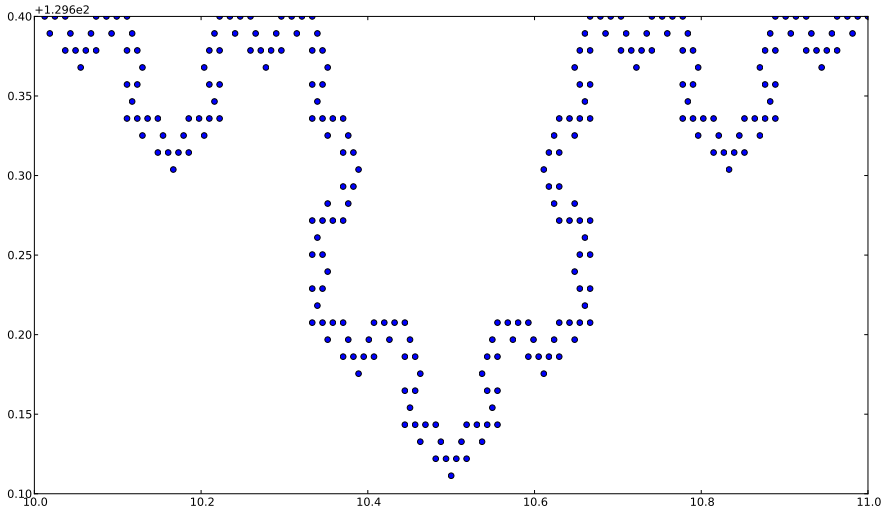


Abbildung 2.1: Punkte der Koch-Kurve.

2.3.2 Spirale

Beispiel 2 ist eine Spirale mit 84 $\mathbb{R}^2$-Koordinaten $(x, y) : (x = t_i * 0.15 * cos(t), y = t_i * 0.15 * sin(t))$ und $t_i = 0.25 * i + 0.5 : i \in \{0, 1, 2, ..., 83\}$. Siehe Abbildungen 2.5 und 2.6.

[7]Das Resultat von LE.

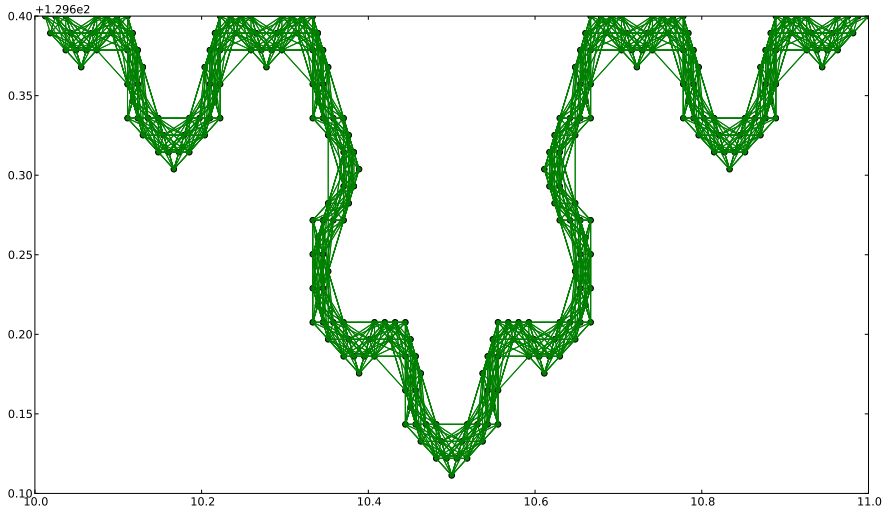


Abbildung 2.2: Graph der Koch-Kurve für $\epsilon = 0.05$.

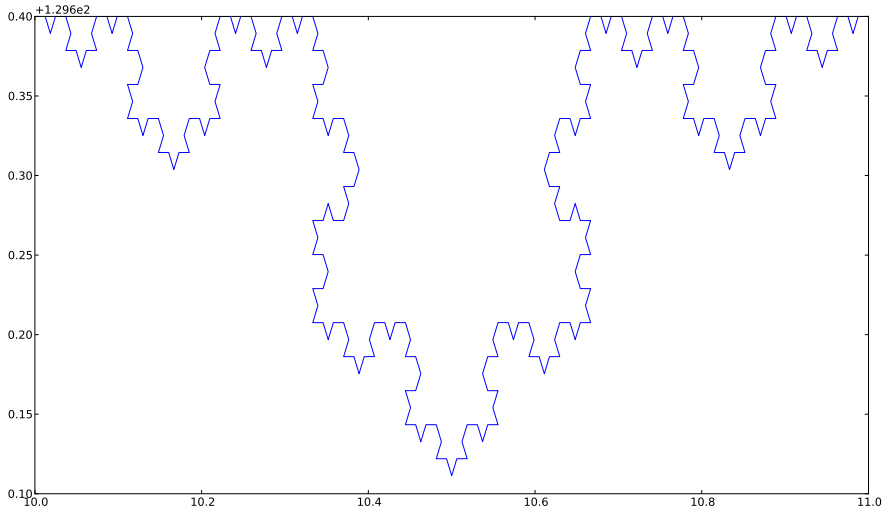
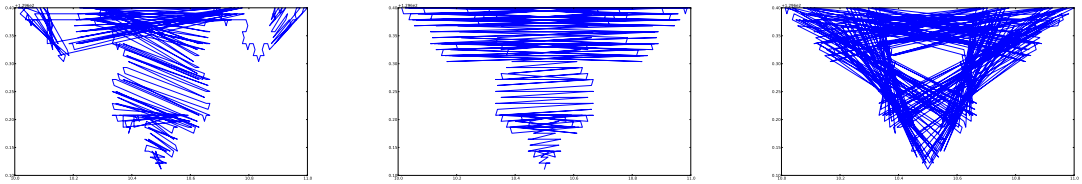
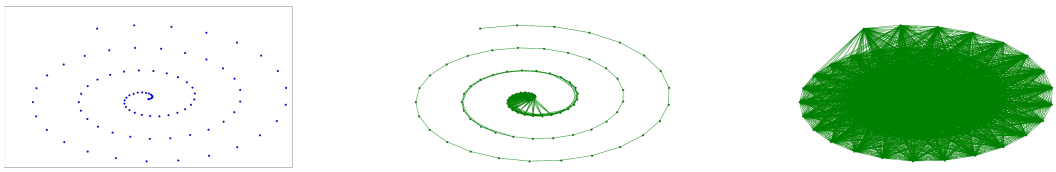


Abbildung 2.3: Laplacian-Eigenmap der Koch-Kurve für $\epsilon = 0.05$ und $t = 0.003$.



(a) Eigenvektor Nummer 0 (b) Eigenvektor Nummer 2 (c) Eigenvektor Nummer 3

Abbildung 2.4: Weitere Eigenvektoren.

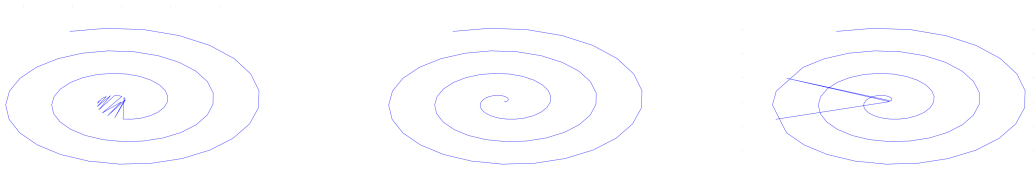


(a) Punkte (b) Graph $\epsilon = 0.8$ (c) Graph $N = 82$

Abbildung 2.5: Spirale mit 84 Punkten, ein Graph mit $\epsilon = 0.8$ und Graph mit $N = 82$.

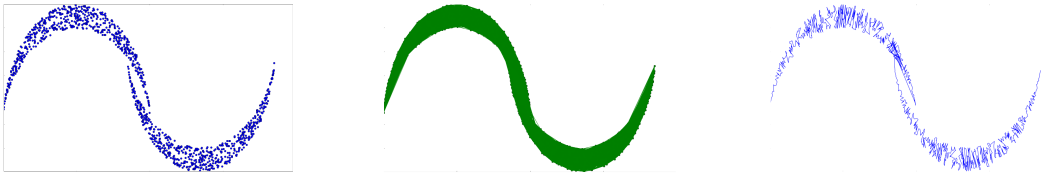
2.3.3 Zwei Halbmonde

Im nächsten Beispiel sieht man zwei Halbmonde ineinander. Die Punkte sind zufällig generiert und liegen zwischen -10.0 und 10.0 auf der x-Achse und zwischen -10.0 und 5.0 auf der y-Achse. Abbildungen 2.7 und 2.8.



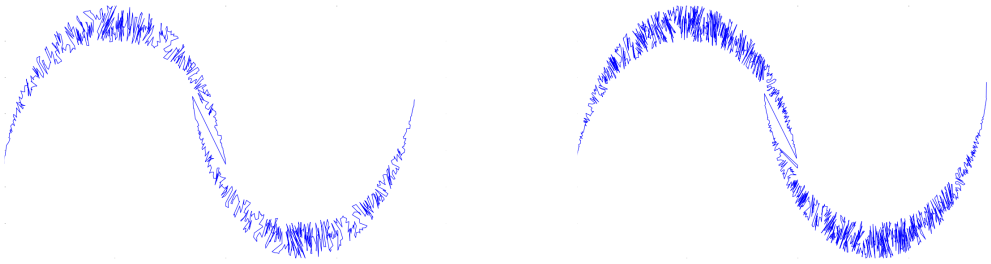
(a) Eigenmap $t = 0.3$ (b) Eigenmap $t = 0.03$ für $\epsilon = 0.8$ und $83 > N > 7$ (c) Eigenmap $t = 0.003$

Abbildung 2.6: Spirale mit 84 Punkten $\epsilon = 0.8$ oder $N > 7$, falls $t = 0.03$ ist.



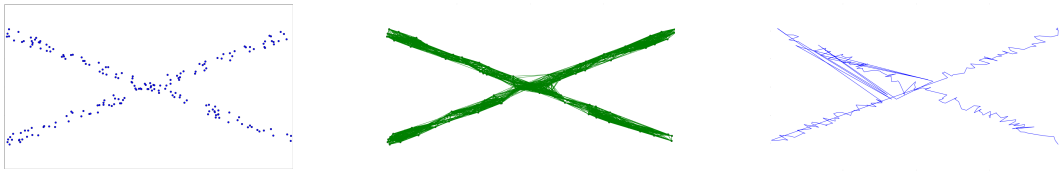
(a) 1200 Punkte (b) Graph $N = 100$ (c) Eigenmap $t = 0.03$

Abbildung 2.7: Zwei Halbmonde mit 1200 Punkten.



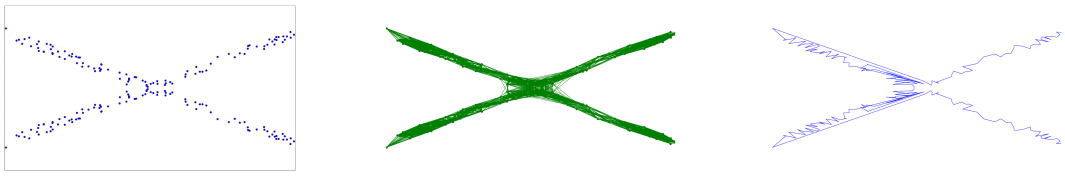
(a) Eigenmap mit $\epsilon = 0.7$, $t = 0.03$ und 1200 Punkten (b) Eigenmap mit $N = 100$, $t = 0.03$ und 2000 Punkten

Abbildung 2.8: Weitere Eigenmaps.



(a) 200 Punkte (b) Graph $N = 20$ (c) Eigenmap $t = 0.3$.

Abbildung 2.9: Kreuz mit 200 Punkten.



(a) 200 Punkte (b) Graph $N = 20$ (c) Eigenmap $t = 0.3$

Abbildung 2.10: Kreuz. Die Punkte unter der x-Achse sind ein Spiegelbild der Oberen.

2.3.4 Kreuz

Als letztes zeigen wir noch einen Fall, in dem das Verfahren scheitern kann und die Kurve falsch darstellt. Hier sieht man eine Menge von Punkten, die ein Kreuz darstellen. Die Punkte liegen zwischen -0.5 und 0.5 auf der x-Achse und zwischen -0.6 und 0.6 auf der y-Achse. Siehe Abbildungen 2.9 und 2.10. LE kann sich hier und speziell in diesem Beispiel nicht für einen bestimmten Weg entscheiden, denn in dem Mittelpunkt des Kreuzes kann LE entweder geradeaus links oder rechts weiterlaufen. Hier kommt der erste Eigenwert, der größer Null ist, mehrfach vor und dadurch erhält man anstelle von einem einzigen Eigenvektor einen Eigenraum, der unendlich viele Eigenvektoren hat.

3 Matrix-Matrix-Multiplikation

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{pmatrix} \tag{3.1}$$

Eine Matrix-Matrix-Multiplikation für quadratische Matrizen $A, B \in \mathbb{C}^{N \times N}$, (Gleichung 3.1 für $N = 3$), ist nur eine Ausführung von N^3 verschiedenen Operationen der Form $C_{ij} += A_{ik} * B_{kj}$ hintereinander. In welcher Reihenfolge diese Operationen bearbeitet werden, spielt bei der Korrektheit der Berechnung von $C = AB$ keine Rolle. Wichtig ist für das Endergebnis nur, dass alle diese N^3-Operationen bearbeitet werden.

Eine Multiplikationsreihenfolge hat N^3 verschiedene Operationen und jede ihre Operationen hat drei Operanden. Mit Operanden sind Matrix-Elemente gemeint, z.B. $A_{ik} : i, k \in \{1, 2 \dots N\}$. Das heißt eine Multiplikationsfolge π der Länge N^3 enthält insgesamt $3 * |\pi| = 3N^3$ Operanden. Die Matrizen A, B, C bestehen aber jeweils nur aus N^2 Einträgen und zusammen $3 * N^2$ Einträgen.

Gemeint ist somit, dass ein einziger Operand in π N-mal vorkommt und zwar in N verschiedenen Operationen. Sollte es bei der Ausführung von π ein Cache der Größe $3 * N^2$ vorhanden sein, dann muß jeder Operand nur ein einziges Mal vom Hauptspeicher in den Cache geholt werden. Ist der Cache kleiner als $3 * N^2$, dann kann es vorkommen, dass zu einem Zeitpunkt $t \in \{1, 2 \dots N^3\}$ Operanden, die im Cache liegen, ausgelagert werden um neue Operanden einfügen zu können, die für die Operation $\pi(t)$ benötigt werden. In solchen Fällen sind mehr als $3 * N^2$ Zugriffe auf den Hauptspeicher nötig.

Abbildung 3.1 zeigt uns das vorläufige Speicher-Modell. Der Hauptspeicher ist ein langsamer Puffer im Vergleich zum Cache (schneller Puffer). D.h., die Kommunikation zwischen CPU und Cache ist viel schneller als die Kommunikation zwischen Cache und Hauptspeicher.

Einen Operanden aus dem Hauptspeicher in den Cache zu holen sollte möglichst vermieden werden. Komplett vermeiden lässt es sich nicht, denn es sind mindestens $3 * N^2$ Zugriffe nötig. Die Anzahl der Zugriffe auf den Hauptspeicher ist von der Cache-Größe und der Operationsfolge π abhängig. Auf Cache und Cache-Varianten kommen wir später nochmal zurück.

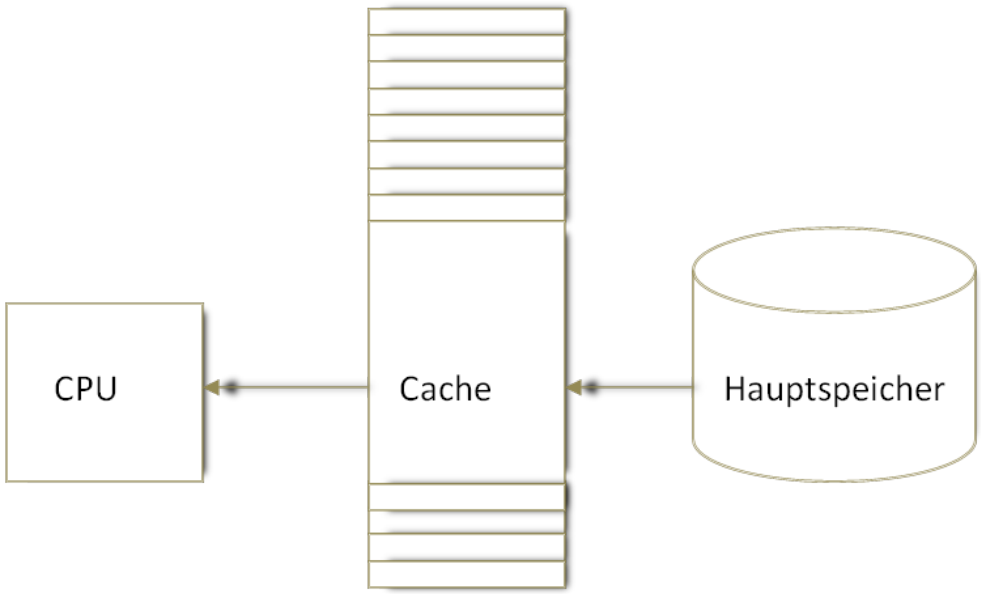


Abbildung 3.1: Speicher-Modell. [BZ06]

3.1 Operation-Folgen

Eine Operation der Form $C_{ij} += A_{ik} * B_{kj}$ werde ich im Folgenden als *ijk* bezeichnen. Insgesamt erhalten wir N^3 verschiedene Operationen *ijk*. Z.B. erhalten wir für $N = 2$:

$1 \Rightarrow 111 \Rightarrow C_{11} += A_{11} * B_{11}$

$2 \Rightarrow 112 \Rightarrow C_{11} += A_{12} * B_{21}$

$3 \Rightarrow 121 \Rightarrow C_{12} += A_{11} * B_{12}$

$4 \Rightarrow 122 \Rightarrow C_{12} += A_{12} * B_{22}$

$5 \Rightarrow 211 \Rightarrow C_{21} += A_{21} * B_{11}$

$6 \Rightarrow 212 \Rightarrow C_{21} += A_{22} * B_{21}$

$7 \Rightarrow 221 \Rightarrow C_{22} += A_{21} * B_{12}$

$8 \Rightarrow 222 \Rightarrow C_{22} += A_{22} * B_{22}$

Nach der Formel $z = (i-1)N^2 + (j-1)N + k$ können wir jede Operation *ijk* eine eindeutige Zahl z aus der Menge $\{1, 2 \ldots N^3\}$ zuordnen. Mit z erhalten wir die Nummer der verschiedenen Operationen *ijk*.

```
def number(i,j,k,N):
   #return (i-1)N^2 + (j-1)N + k
   return (i-1)*N**2 +(j-1)*N +k
```

Man kann auch umgekehrt aus einer Nummer z die Operation ijk bestimmen.

$$\begin{aligned} i &= 1 + (z_0 \div N^2) && : z_0 = z - 1 \\ j &= 1 + (z_1 \div N) && : z_1 = z_0 \mod N^2 \\ k &= 1 + (z_1 \mod N) \end{aligned}$$

Diese Funktion, die das Tripel (i,j,k) liefert, bezeichne ich hier als $ijk_N(z)$. Die zugehörige Python-Funktion $ijk(number, N)$ liefert die Liste $[i,j,k]$ aus.

```
def ijk(number,N):
   z0=number-1
   z1=z0%(N**2)
   #return [i,j,k]
   return [1+(z0/N**2),1+(z1/N),1+(z1%N)]
```

Sei π eine Permutation der Zahlen $1, 2 \ldots N^3$ (beispielsweise $\pi = (2,1,4,3,6,5,8,7)$ für $N = 2$), dann kann man durch die Anwendung von ijk_N auf alle Elemente von π alle Tripeln erhalten, die wir für die Matrix-Matrix-Multiplikation brauchen.

```
# MatrixMult bekommt als Eingabe die Permutation π.
def MatrixMult(pi):
   global A
   global B
   C=zeros((len(A),len(A))
   for z in range(1,1+len(pi)):
      [i,j,k]=ijk(pi[z],N)
      # C_ij+ = A_ik * B_kj
      C[i,j]=C[i,j]+A[i,k]*B[k,j]
   return C
```

Ist $\pi = id$, wobei id diejenige Permutation ist, die ihre Elemente aufsteigend sortiert hat, dann entspricht $MatrixMult(id)$ der Standard-Multiplikation, die wir alle kennen.

```
def StandardMult():
   global A
   global B
   C=zeros((len(A),len(A))
   for i in range(1,N+1):
      for j in range(1,N+1):
         for k in range(1,N+1):
            C[i,j]=C[i,j]+A[i,k]*B[k,j]
   return C
```

Die Permutation *id* entspricht nur eine von $N^3!$ verschiedenen Permutationen $\pi_1, \pi_2 \dots \pi_{N^3!}$ mit $\pi_1 = id$. Als Beispiel für $N = 3$ hat man $3^3! = 27! \approx 10^{28}$ verschiedene Permutationen. Alle $N^3!$-Folgen auf Cache-Effizienz zu testen ist unmöglich.

Später werden wir eine spezielle Folge, genannt Laplace-Folge (kurz $\pi_{Laplace}$), definieren und sie auf Cache-Effizienz überprüfen und dann mit *id* und einer anderen bekannten Multiplikationsfolge, nämlich die Peano-Folge (kurz π_{Peano}), vergleichen.

3.2 Cache-Effizienz-Messung

Eine Operation *ijk* enthält drei Operanden A_{ik}, B_{kj} und C_{ij}, die für die Ausführung von *ijk* in dem Cache vorhanden sein müssen. Bei der Implementierung haben Operationen und Operanden die gleiche Datenstruktur *OP*. *OP* besteht aus einer Liste von drei Zahlen aus der Menge $\{0, 1 \dots N\}$.

```
OP = [('i', int), ('j', int), ('k', int)]
# [i,j,k] ⇒ C_ij += A_ik * B_kj .
# [i,0,k] ⇒ A_ik .
# [0,j,k] ⇒ B_kj .
# [i,j,0] ⇒ C_ij .
```

Ist eine der drei Zahlen eine Null, dann repräsentiert *OP* einen Operanden. Sind alle drei Zahlen größer als Null, dann ist *OP* eine Operation. Der Cache P und die Multiplikation-Folge π sind im Folgenden nur Listen von *OP*-Elementen.

Hat der Cache P die Größe m, dann liefert uns die Funktion $\tau_m(\pi, t)$ den Cache-Inhalt nach der Ausführung von der Operation $\pi(t)$ als eine Liste. Es gilt $|\tau_m(\pi, t)| \leq m$ $\forall t \in \{0, 1 \dots N^3 - 1\}$. Seien weiterhin die Funktionen $\psi_m(\pi, t)$ und $\sigma_m(\pi, t)$ mit:

$$\psi_m(\pi, t) = \begin{cases} 0 & , t = 0 \\ \sum |e : e \in \tau_m(\pi, t) \wedge e \notin \tau_m(\pi, t-1)| & , 1 \leq t < N^3 \end{cases} \tag{3.2}$$

$$\sigma_m(\pi, t) = \sum_{n=0}^{t} \psi_m(\pi, n) \tag{3.3}$$

Mit der Funktion ψ wird gezählt, wie viele Operanden zum Zeitpunkt t neu in den Cache gekommen sind. Es ist im Allgemeinen $\psi_m(\pi, t) \neq |\tau_m(\pi, t)| - |\tau_m(\pi, t-1)|$, das gilt zwar für manche t aber nicht für alle, denn wir entfernen Operanden entweder wenn der Cache voll ist oder wenn sie schon N-Mal bei der Berechnung vorkamen. Letzteres besagt, falls wir zum Zeitpunkt $t - 1$ feststellen können, dass e schon N-Mal in der Berechnung vorkam, dann wird er erst aus $\tau_m(\pi, t)$ entfernt, aber nicht aus $\tau_m(\pi, t-1)$. Von daher

stellt die Differenz in dieser Situation zwischen $\tau_m(\pi, t)$ und $\tau_m(\pi, t-1)$ ein falsches Ergebnis dar.

Die Funktion σ zählt alle neu geladenen Operanden von Zeitpunkt 0 bis Zeitpunkt t. Durch $\sigma_m(\pi, N^3)$ bekommt man die Anzahl aller neu geladenen Operanden, während des Multiplikationsverlaufs abhängig von der Cache-Größe m und der Permutation π.

Ist $\sigma_m(\pi_a, N^3) \leq \sigma_m(\pi_b, N^3)$, so wissen wir, dass π_a eine bessere Cache-Effizienz als π_b hat.

Die verschiedenen Werte von τ stehen später in der Liste *cacheVerlauf*, d.h. $cacheVerlauf[t] = \tau_m(\pi, t+1)$. Die Funktionen τ, ψ und σ:

```
#tau(t) ⟹  cacheVerlauf[t − 1]
#psi(t) ⟹  Psi[t − 1]
#sigma(t) ⟹  Sigma[t − 1]
def tau(t):
   return cacheVerlauf[t-1]

def psi(t):
   return Psi[t-1]

def sigma(t):
   return Sigma[t-1]
```

Das nächste Beispiel verdeutlicht die Funktionen τ, ψ und σ. Ist $\pi = id$, dann gilt für $N = 3$ und $m = 7$:

$$\tau_7(id, 0) = []$$
$$\tau_7(id, 1) = [[1,0,1],[0,1,1],[1,1,0]]$$
$$\tau_7(id, 2) = [[1,0,1],[0,1,1],[1,1,0],[1,0,2],[0,1,2]]$$
$$\tau_7(id, 3) = [[1,0,1],[0,1,1],[1,1,0],[1,0,2],[0,1,2],[1,0,3],[0,1,3]]$$

und

$$\psi_7(id, 0) = 0 \quad \sigma_7(id, 0) = 0$$
$$\psi_7(id, 1) = 3 \quad \sigma_7(id, 1) = 3$$
$$\psi_7(id, 2) = 2 \quad \sigma_7(id, 2) = 5$$
$$\psi_7(id, 3) = 2 \quad \sigma_7(id, 3) = 7$$

Hier sieht man die Ausgaben nach der Ausführung von $id(1), id(2)$ und $id(3)$. Die nächste Operation $id(4)$ hat keinen Platz mehr im Cache für alle ihre Operanden, da $m = 7$ ist. D.h. $id(4)$ besteht aus den Operanden $[1,0,1], [0,2,1]$ und $[1,2,0]$ und einer davon $[1,0,1]$ befindet sich schon im Cache an der ersten Stelle. In dieser Situation müssen mindestens 2 Operanden aus dem Cache entfernt werden. Welche Elemente man aus dem Cache entfernt, bestimmt die gewählte Cache-Strategie, siehe weiter unten.

Ein Operand e der in den restlichen Operationen von π nicht mehr vorkommen wird, wird aus dem Cache entfernt. D.h. e würde schon N-Mal aufgerufen und wird für die Berechnung nicht mehr benötigt. Er wird aus dem Cache entfernt und zwar bei allen Cache-Strategien, die man verwendet. Dabei ist es gleichgültig, ob der Cache voll ist oder nicht. Jeder Operand darf in dem Cache nur einmal vorkommen.

Für jeden Operanden, der sich im Cache befindet, kommen die folgenden drei Fälle in Frage:

1. Der Operand wird zum ersten Mal gebraucht und muss vom Hauptspeicher in dem Cache neu geladen werden.
2. Der Operand befindet sich schon im Cache, da er bei den Ausführungen früherer Operationen benötigt würde.
3. Der Operand war schon im Cache, aber er wurde zwischenzeitlich ausgelagert, da der Cache voll war und Platz für andere Operanden nötig war. Hier muss der Operand neu geladen werden.

3.2.1 Cache-Arten

Ein Cache ist für uns nur ein Puffer, der Operanden wie A_{ij} speichern kann. Zur Effizienz-Untersuchung verwenden wir zwei verschiedene Caches-Implementierungsstrategien[1]:

Erste Strategie

Die erste Strategie soll in die Zukunft schauen können und diejenigen Elemente aus dem Cache entfernen, die als letztes in π vorkommen. Dazu wird eine Liste *nextUse* erstellt, die für jeden Operand aus dem Cache sein nächstes Vorkommen in π enthält. Ein Operand, der in π nicht mehr vorkommen wird, enthält in *nextUse* eine Null, so kann er bei der Ausführung der nächsten Operation aus dem Cache entfernt werden. Zum Beispiel bevor wir $\pi(t)$ bearbeiten, gilt $|L| = |nextUse|$ mit $L = \tau_m(\pi, t-1)$ und $nextUse[i]$ enthält das nächste Vorkommen von $L[i]$ in π. Im nächsten Schritt muss überprüft werden, ob die Liste *nextUse* eine Null enthält. Gilt z.B. $nextUse[i] = 0$, dann müssen $nextUse[i]$ und $L[i]$ aus *nextUse* bzw. aus L entfernt werden. Das wird durch die Funktion $cleanL()$ erledigt.

[1] Die Implementierung bezieht sich nur auf Matrix-Matrix-Multiplikation.

```
def cleanL(L,nextUse):
   i=len(L)-1
   while i>=0:
      if nextUse[i]==0:
         nextUse.pop(i)
         L.pop(i)
      i=i-1
```

```
def Operanden(t):
   global N
   global pi
   operation=ijk(pi(t),N)
   [e1,e2,e3]=[[operation[0],0,operation[2]],
               [0,operation[1],operation[2]],
               [operation[0],operation[1],0]]
   return [e1,e2,e3]
```

Als Nächstes werden die Operanden $e_1 = A_{ik}, e_2 = B_{kj}$ und $e_3 = C_{ij}$ von $\pi(t)$ nacheinander in L eingefügt. Hier muss man drei Fälle beachten:

1. Existiert ein Element aus L, zum Beispiel $L[i] = e_1$ (für e_2 und e_3 gilt das Gleiche), so muss nur $nextUse[i]$ aktualisiert werden. Durch die Funktion $next\_use(e, pi, t)$ wird bestimmt, wann der Operand $e = L[i]$ in π als nächstes vorkommt.

   ```
   def next_use(e,pi,t):
      k=t+1
      while k<len(pi):
         operanden=Operanden(pi[k])
         if operanden.count(e)>0:
            return k
         else:
            k=k+1
      return 0
   ```

2. e_1 ist nicht in L vorhanden und es gilt $|L| < m$, dann muss e_1 und sein nächstes Vorkommen in π am Ende von L bzw. $nextUse$ eingefügt werden. Eingefügt wird stets am Ende des Caches.

```
def cases(t,L,nextUse):
    global pi,m
    operanden=Operanden(t-1)
    cleanL(L,nextUse)
    for i in range(len(operanden)):
        e=operanden[i]
        if L.count(e)==1:#Fall 1
            index=L.index(e)
            nextUse[index]=next_use(e,pi,t-1)
        else:#Fall 2 und 3
            if len(L)==m:#Fall 3
                index=nextUse.index(max(nextUse))
                L.pop(index)
                nextUse.pop(index)
            l=len(L)
            L.append(e)
            nextUse.append(next_use(e,pi,t-1))
    return [L,nextUse]
```

3. Ansonsten gilt der Fall, dass $|L| = m$ und e_1 in L nicht enthalten ist. Hier muss für die Stelle i das Maximum (entspricht dem Element, das in π als letztes vorkommt) von *nextUse* bestimmt werden, damit wir $L[i]$ und $nextUse[i]$ löschen können. Dann wird e_1 am Ende von L eingefügt und $next\_use(len(L), pi, t)$ am Ende von der Liste *nextUse*.

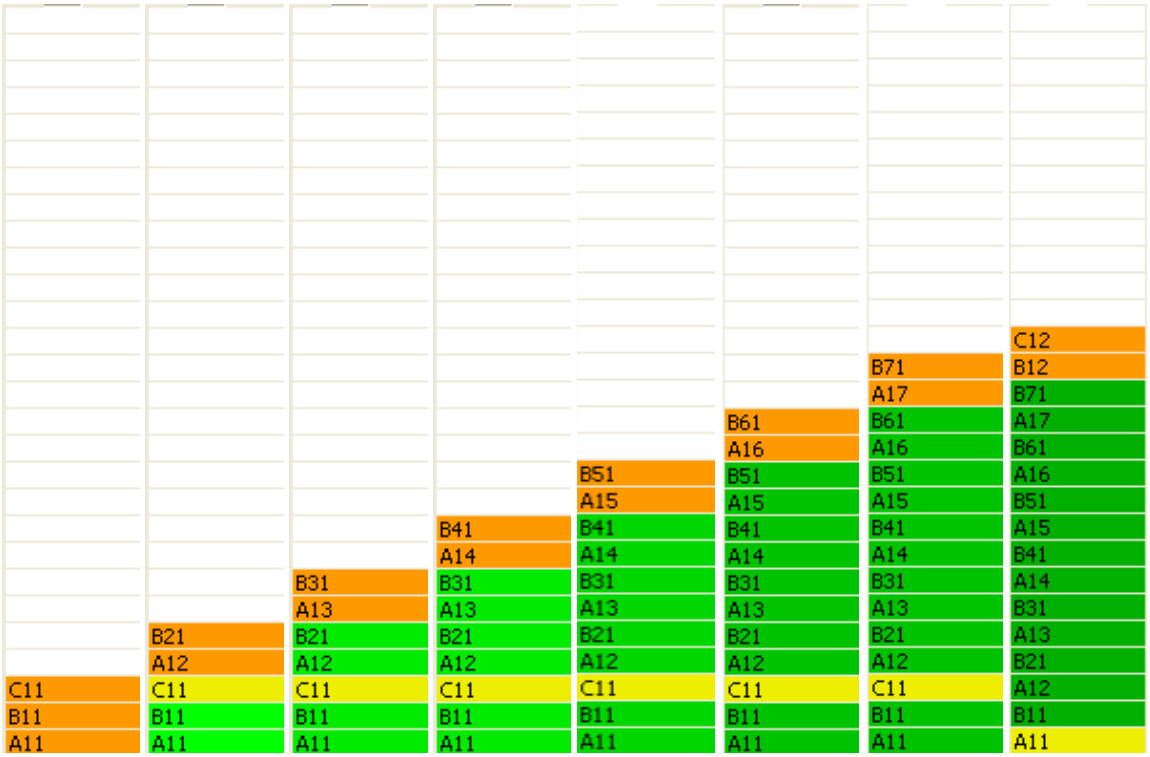
```
# Es gilt m ≥ 3
cacheVerlauf=[]
Psi=[]
Sigma=[]
for i in range(len(pi)):
    x=[]
    if i==0:
        x.append(Operanden(0))
        x.append([next_use(x[0][0],pi,0),
                  next_use(x[0][1],pi,0),
                  next_use(x[0][2],pi,0)])
        cache.append(x)# x=[L,nextUse]
        Psi.append(3)
        Sigma.append(3)
    else:
        x=cases(i+1,cache[i-1][0],cache[i-1][1])
        cache.append(x)# x=[L,nextUse]
        l1=cache[i][0]
        l2=cache[i-1][0]
        count=0
        # ψ = ∑ |e| : e ∈ τ_m(π,t) ∧ e ∉ τ_m(π,t−1)
        for j in range(len(l1)):
            if l2.count(l1[j])==0:
                count=count+1
        Psi.append(count)
        # σ = ∑_{n=0}^{t} ψ_m(π,n)
        Sigma.append(Psi[i]+Sigma[i-1])
    cacheVerlauf.append(array(cache[i]))
```

Zweite Strategie

Die zweite Strategie soll diejenigen Elemente aus dem Cache entfernen, die am längsten nicht verwendet wurden. Dafür muss nur die Liste *nextUse* durch *lastUse* ersetzt werden. Die neue Liste merkt sich nicht das nächste Vorkommen vom Operand e in π, sondern sie merkt sich die letzte Verwendung von e im Cache. Entfernt werden die Elemente $L[i]$ und $lastUse[i]$ aus L bzw. *lastUse*, wobei $lastUse[i]$ dem Minimum in *lastUse* entspricht.

3.2.2 Beispiel

Als Beispiel betrachten wir den Cache-Verlauf für die Standard-Multiplikation mit $N = 7$ und ein Cache der Größe 28. Wir verwenden hier die erste Strategie, die in die Zukunft schaut. In diesem Fall hat man $7^3 = 343$ verschiedene Operationen. Die Abbildung 3.2 zeigt uns den Cache-Verlauf für $t = 1$ bis $t = 7$ und berechnet den Endwert von C_{11}.



(a) $t = 1$ (b) $t = 2$ (c) $t = 3$ (d) $t = 4$ (e) $t = 5$ (f) $t = 6$ (g) $t = 7$ (h) $t = 8$

Abbildung 3.2: Cache Ablauf für Standard-Multiplikation $t = 1 \ldots 8$.

$$C_{11} = \begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} & A_{15} & A_{16} & A_{17} \end{pmatrix} * \begin{pmatrix} B_{11} \\ B_{21} \\ B_{31} \\ B_{41} \\ B_{51} \\ B_{61} \\ B_{71} \end{pmatrix}$$

C_{11} wurde nur einmal aus dem Speicher in den Cache geladen. Sie wurde in allen Operationen $id(1), id(2), id(3), id(4), id(5), id(6)$ und $id(7)$ gebraucht. Bei der Ausführung von $id(8)$ wird C_{11} aus dem Cache entfernt.

Die Farben gelb, orange und rot zeigen auf die Operanden, die von der Operation $id(t)$ benötigt werden. Operanden, die von $id(t)$ nicht benötigt werden, sind in grün gefärbt. Die grüne Farbe verdeutlicht in unserem Beispiel auch die Belastung des Caches. Hellgrün für niedrige Belastung und dunkelgrün für hohe Belastung.

1. Gelb für Operanden, die zum Zeitpunkt t im Cache vorhanden sind.

2. Orange für Operanden, die zum ersten Mal im Cache landen.
3. Rot für Operanden, die zum wiederholten Mal im Cache geladen werden müssen.

Abbildung 3.3 zeigt uns den Cache-Ablauf für $t = 80$ bis $t = 86$. Hier sieht man, dass der Cache voll belastet ist. Die Operanden von der Matrix B werden bei Ausführung der Operationen $id(80)$, $id(81)$, $id(82)$, $id(83)$ und $id(84)$ neu geladen und werden direkt danach aus dem Cache entfernt. Operanden der Matrix B tauchen bei der Standard-Multiplikation in id nach N^2-Schritten wieder auf. Deshalb werden sie in unserem Fall sofort aus dem Cache entfernt, aber nicht diejenigen Operanden, die schon länger im Cache sind.

In der Abbildung 3.4 sehen wir die Standard-Matrix-Multiplikation $C = A * B$ für $N = 7$. Im Bild sieht man den Zustand nach der Ausführung von $id(86)$. In gelb sind die Operanden zu sehen, die bei der Ausführung von $id(86) = [2, 6, 2]$ benötigt werden ($C_{26} += A_{22} * B_{26}$).

Die Zahlen, die in den Operanden-Felder zu sehen sind, zeigen uns, wie oft die jeweiligen Operanden bis Zeitpunkt $t = 86$ aufgerufen wurden. Dort stehen Zahlen zwischen 0 und 7. Am Anfang überall 0 und nach der Ausführung von $id(342) = [7, 7, 7]$ überall 7.

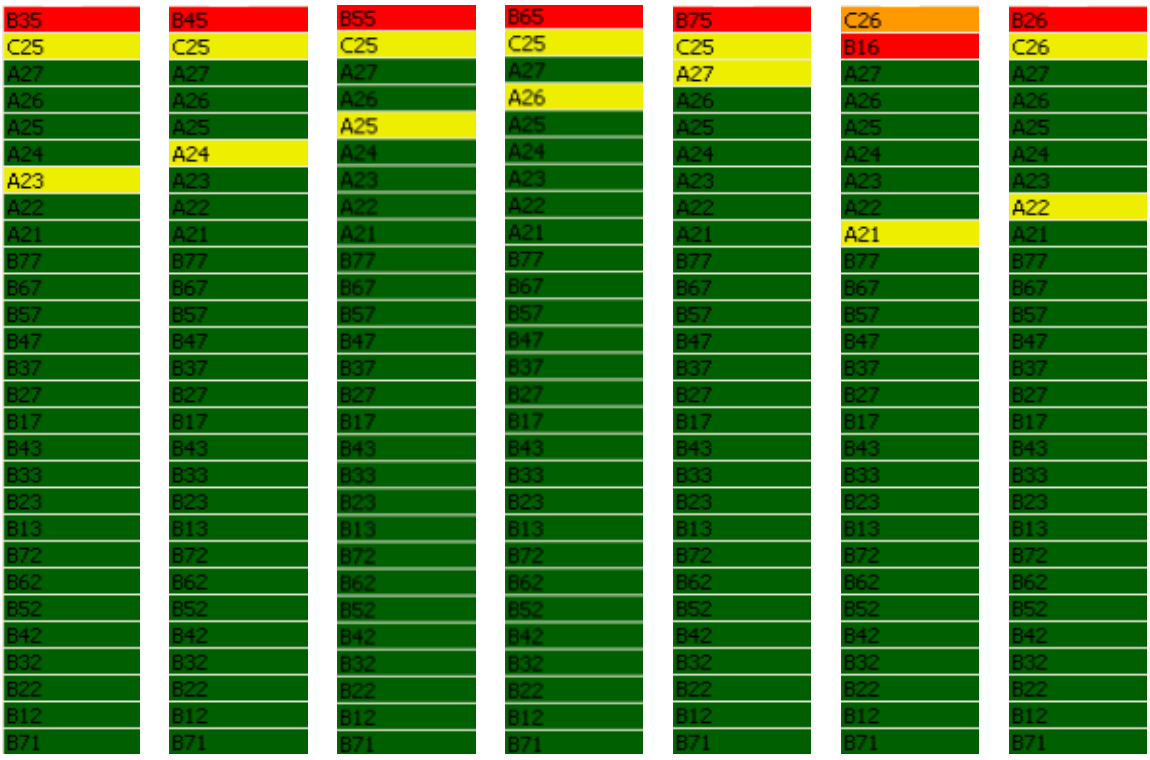
Die Felder in Abbildung 3.5 enthalten die Anzahl der Neuladungen der verschiedenen Operanden, in denen angezeigt wird, wie oft jeder Operand vom Speicher ins Cache geladen wurde. Abbildung 3.5 zeigt auch den Zustand für $t = 86$.

3.3 Standard-Multiplikation

Unter der Standard-Multiplikation meinen wir die aufsteigende Reihenfolge $\pi_{Standard}$, d.h. die Nummerierung der Operationen ist aufsteigend sortiert. Der folgende Quellcode liefert die Nummer der Operationen für die bekannte Standard-Multiplikation.

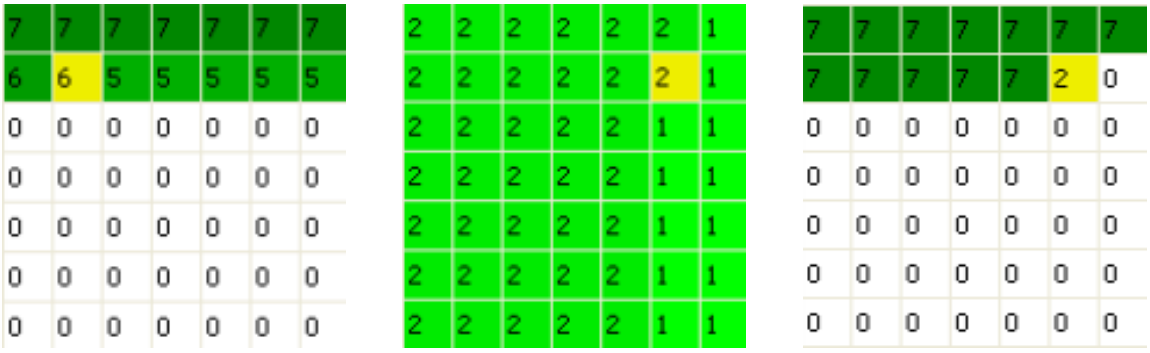
```
N=3
for i in range(1,N+1):
   for j in range(1,N+1):
      for k in range(1,N+1):
         C[i,j]=C[i,j]+A[i,k]*B[k,j]
            print number(i,j,k,N)
# Ausgabe 1 2 3 .. 26 27
```

Betrachten wir die Operation ijk als 3D-Punkt (i, j, k), dann zeigt die Abbildung 3.6 die Reihenfolge für $N = 9$. In dieser Reihenfolge kommen die gleichen Operanden von C stets hintereinander. D.h, C_{11} taucht in jeder der ersten N-Operationen von $\pi_{Standard}$ auf und wird nach $\pi_{Standard}(N)$ nicht mehr bei der Berechnung auftauchen. Operanden von A kommen wiederholt erst nach N Schritten in $\pi_{Standard}$ wieder und Operanden von B



(a) $t = 80$ (b) $t = 81$ (c) $t = 82$ (d) $t = 83$ (e) $t = 84$ (f) $t = 85$ (g) $t = 86$

Abbildung 3.3: Cache Ablauf für Standard-Multiplikation $t = 80 \ldots 86$.



(a) Matrix A (b) Matrix B (c) Matrix C

Abbildung 3.4: Matrix-Multiplikation nach der Operation $t = 86$.

1	1	1	1	1	1	1
1	1	1	1	1	1	1
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

(a) Matrix A

1	1	1	2	2	2	1
1	1	1	2	2	2	1
1	1	1	2	2	1	1
1	1	2	2	2	1	1
1	1	2	2	2	1	1
1	1	2	2	2	1	1
1	1	2	2	2	1	1

(b) Matrix B

1	1	1	1	1	1	1
1	1	1	1	1	1	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

(c) Matrix C

Abbildung 3.5: Anzahl der Ladungen von allen Operanden nach Ausführung der Operation $t = 86$.

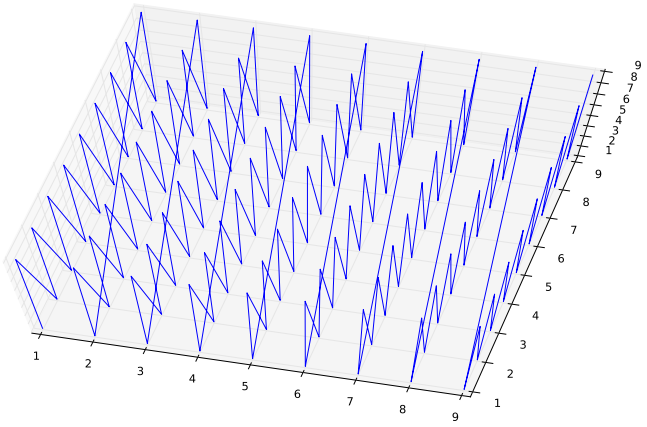
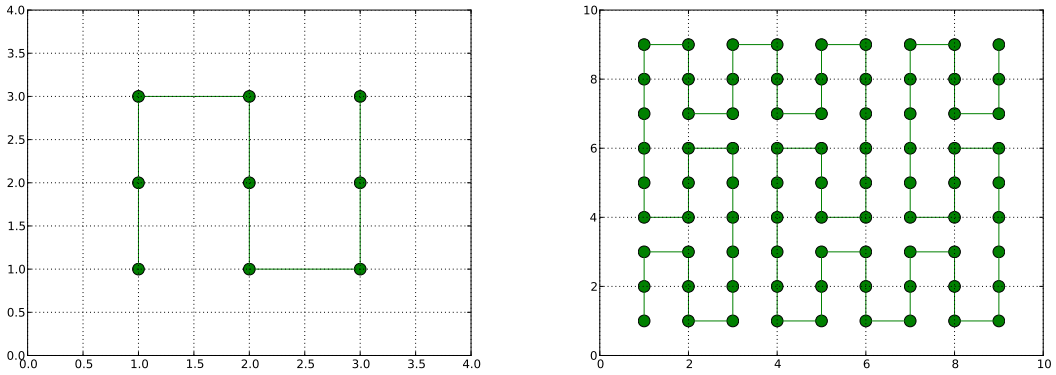


Abbildung 3.6: Standard-Multiplikation $N = 9$.



(a) Peano-Kurve $N = 3$

(b) Peano-Kurve $N = 9$

Abbildung 3.7: Die ersten beiden Iterationen der Peano-Kurve.

stets nach N^2 Schritten. Falls die Cache-Größe $>= N^2 + N + 1$ ist, dann reicht es, jeden Operanden nur einmal in den Cache zu laden. D.h., in diesem Fall enthält der Cache die Matrix B, eine Zeile von A und nur einen Operanden von C. Insgesamt sind $N^2 + N + 1$ Plätze nötig.

3.4 Peano-Multiplikation

Unter der Peano-Multiplikation [BZ06] ist eine Permutation π_{Peano} gemeint, die durch die Operationen (i, j, k) der Matrix-Matrix-Multiplikation anhand der Peano-Kurve wie in Abbildung 3.7 läuft. D.h., wenn man diese Operationen als 3D-Koordinaten betrachtet, dann werden diese Koordinaten so nacheinander bearbeitet, wie die Peano-Kurve diese vorgibt. In [BZ06] wurde das Vorgehen beschrieben. Es werden erst Blöcke der Größe $(3 \times 3 \times 3)$ bearbeitet. Ist die Matrix mit $|N|$ größer als drei, so wird sie in mehreren Blöcken geteilt, die dann rekursiv anhand der Peano-Kurve hintereinander bearbeitet werden. Allgemein für größere N wird erst nach Blöcke der Größe $3^k : 3^k < N \leq 3^{k+1}$ geteilt und dann rekursiv weiter für $k-1, k-2 \ldots 1, 0$.

Für $N = 3$ und $N = 9$ ist der Peano-Verlauf in Abbildung 3.8 bzw. Abbildung 3.9 deutlich zu sehen.

In der Gleichung 3.4 sieht man einen möglichen Verlauf der Peano-Multiplikation für Blöcke der Größe 3^k ($k = N/3$). Z.B. entspricht der erste Block A_{11} in A einer Matrix der Größe $3^k \times 3^k$:($k = 0, 1, 2, \ldots$). Hat man π_{Peano} für Matrizen der Größe 3^n berechnet, so erhält man aus π_{Peano} alle Peano-Folgen für Matrizen, die kleiner als 3^n sind. Unnötige Operationen werden in diesem Fall nicht durchgeführt.

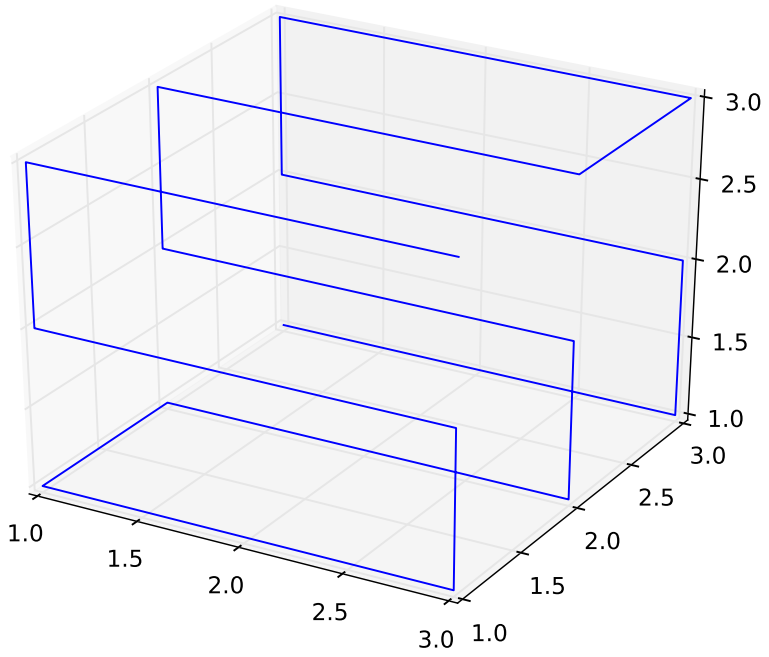


Abbildung 3.8: Peano-Multiplikation $N = 3$.

$$\begin{pmatrix} C_{13} += A_{11} * B_{13} \\ C_{23} += A_{21} * B_{13} \\ C_{33} += A_{31} * B_{13} \\ C_{33} += A_{32} * B_{23} \\ C_{23} += A_{22} * B_{23} \\ C_{13} += A_{12} * B_{23} \\ C_{13} += A_{13} * B_{33} \\ C_{23} += A_{23} * B_{33} \\ C_{33} += A_{33} * B_{33} \end{pmatrix} \Longrightarrow \begin{pmatrix} C_{32} += A_{33} * B_{32} \\ C_{22} += A_{23} * B_{32} \\ C_{12} += A_{13} * B_{32} \\ C_{12} += A_{12} * B_{22} \\ C_{22} += A_{22} * B_{22} \\ C_{32} += A_{32} * B_{22} \\ C_{32} += A_{31} * B_{12} \\ C_{22} += A_{21} * B_{12} \\ C_{12} += A_{11} * B_{12} \end{pmatrix} \Longrightarrow \begin{pmatrix} C_{11} += A_{11} * B_{11} \\ C_{21} += A_{21} * B_{11} \\ C_{31} += A_{31} * B_{11} \\ C_{31} += A_{32} * B_{21} \\ C_{21} += A_{22} * B_{21} \\ C_{11} += A_{12} * B_{21} \\ C_{11} += A_{13} * B_{31} \\ C_{21} += A_{23} * B_{31} \\ C_{31} += A_{33} * B_{31} \end{pmatrix} \tag{3.4}$$

Peano-Multiplikation π_{Peano} hat einen deutlich komplizierteren Quellcode im Vergleich zur Standard-Multiplikation $\pi_{Standard}$, aber eine viel bessere Cache-Ausnutzung und ermöglicht die Multiplikation für beliebige Matrix-Größen.

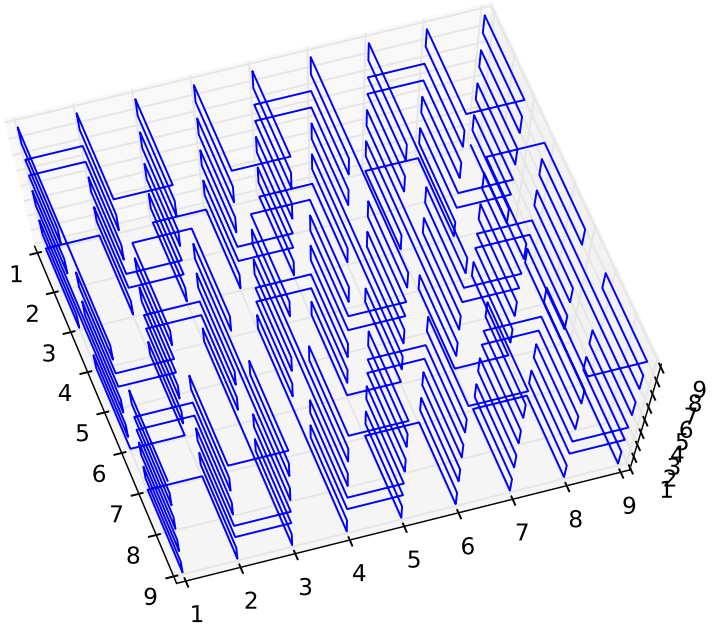


Abbildung 3.9: Peano-Multiplikation $N = 9$.

Die nächsten vier Abbildungen 3.10, Abbildung 3.11, Abbildung 3.12 und Abbildung 3.13 demonstrieren die Peano-Multiplikation für Matrizen $A, B, C \in \mathbb{C}^{N \times N}$. In gelb sind Operanden markiert, die zum Zeitpunkt t die Operation $\pi_{Peano}(t)$ enthält. Grün für Operanden, die bei der Berechnung schon benutzt wurden. In jedem Feld der drei Matrizen steht , wie oft jeder Operand bis Zeitpunk t verwendet wurde.

In diesem Teil der Arbeit wurden zwei Multiplikationsvarianten Standard- und Peano-Multiplikation erläutert und hauptsächlich ein Werkzeug für Cache-Effizienz-Messung geschaffen, welches wir im nächsten Teil benötigen. Im nächsten Kapitel wird eine neue Multiplikationsvariante Laplace-Multiplikation $\pi_{Laplace}$, die auf das LE-Verfahren beruht, entworfen. Die Laplace-Multiplikation wird dann auf Cache-Effizienz mit den anderen zwei Varianten verglichen. Das heißt, wir werden die Werte $\sigma_m(\pi_{Laplace}, N^3)$, $\sigma_m(\pi_{Standard}, N^3)$ und $\sigma_m(\pi_{Peano}, N^3)$ genauer analysieren.

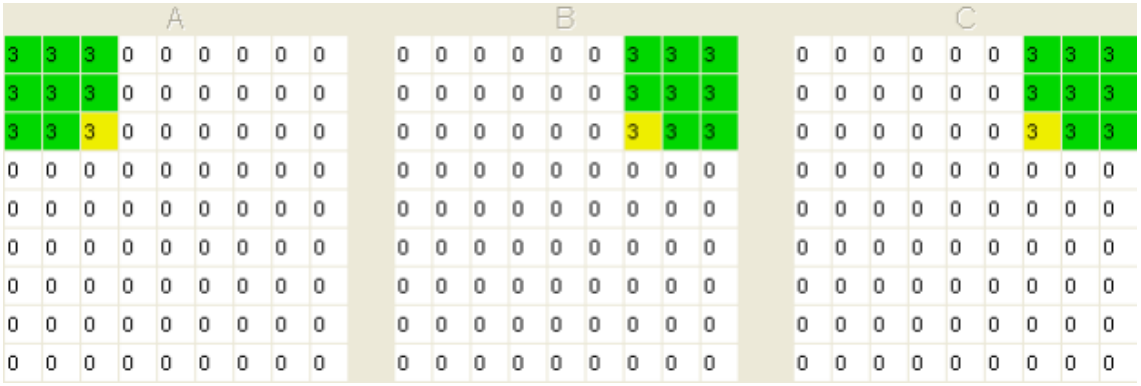


Abbildung 3.10: Peano-Multiplikation $N = 9$. Die Operationen $0 \ldots 26$.

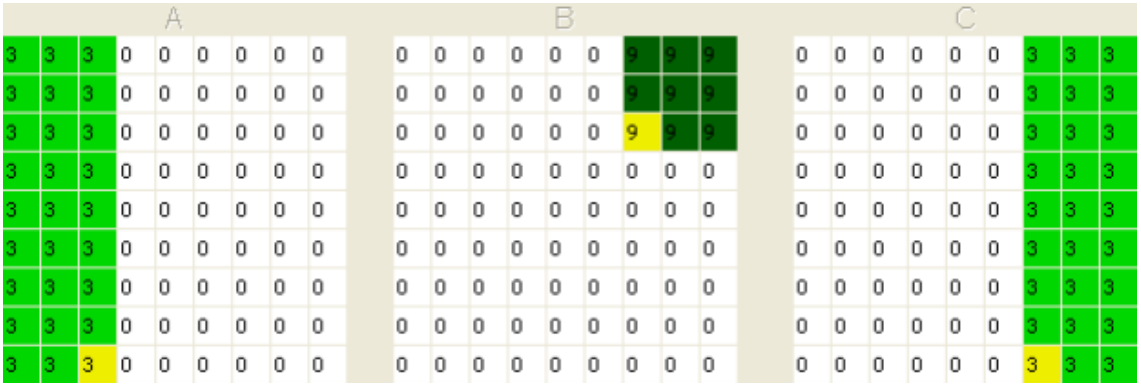


Abbildung 3.11: Peano-Multiplikation $N = 9$. Die Operationen $0 \ldots 80$.

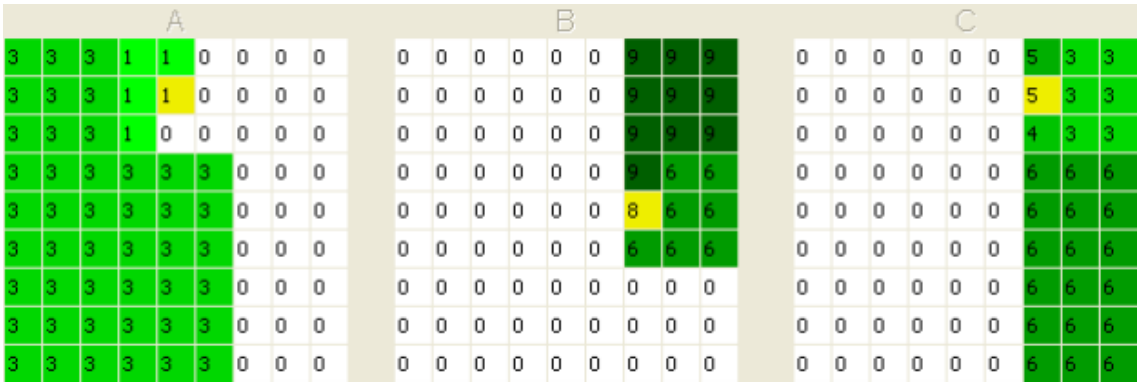


Abbildung 3.12: Peano-Multiplikation $N = 9$. Die Operationen $0 \ldots 139$.

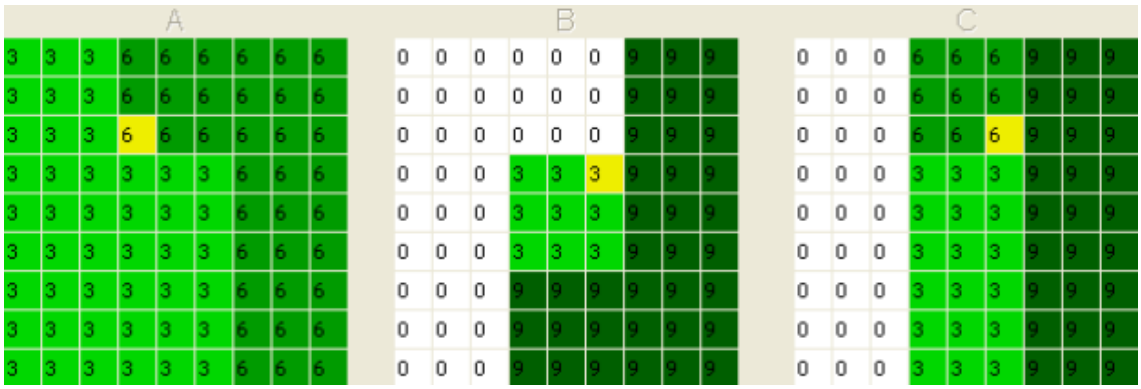


Abbildung 3.13: Peano-Multiplikation $N = 9$. Die Operationen $0 \ldots 350$.

4 Analysewerkzeug

Im Rahmen dieser Arbeit habe ich ein kleines Programm entworfen, das mir bei der Ergebnisdarstellung und Ergebnisanalyse hilft. Im Kapitel Matrix-Matrix-Multiplikation wurden schon mehrere Abbildungen aus diesem Programm vorgestellt.

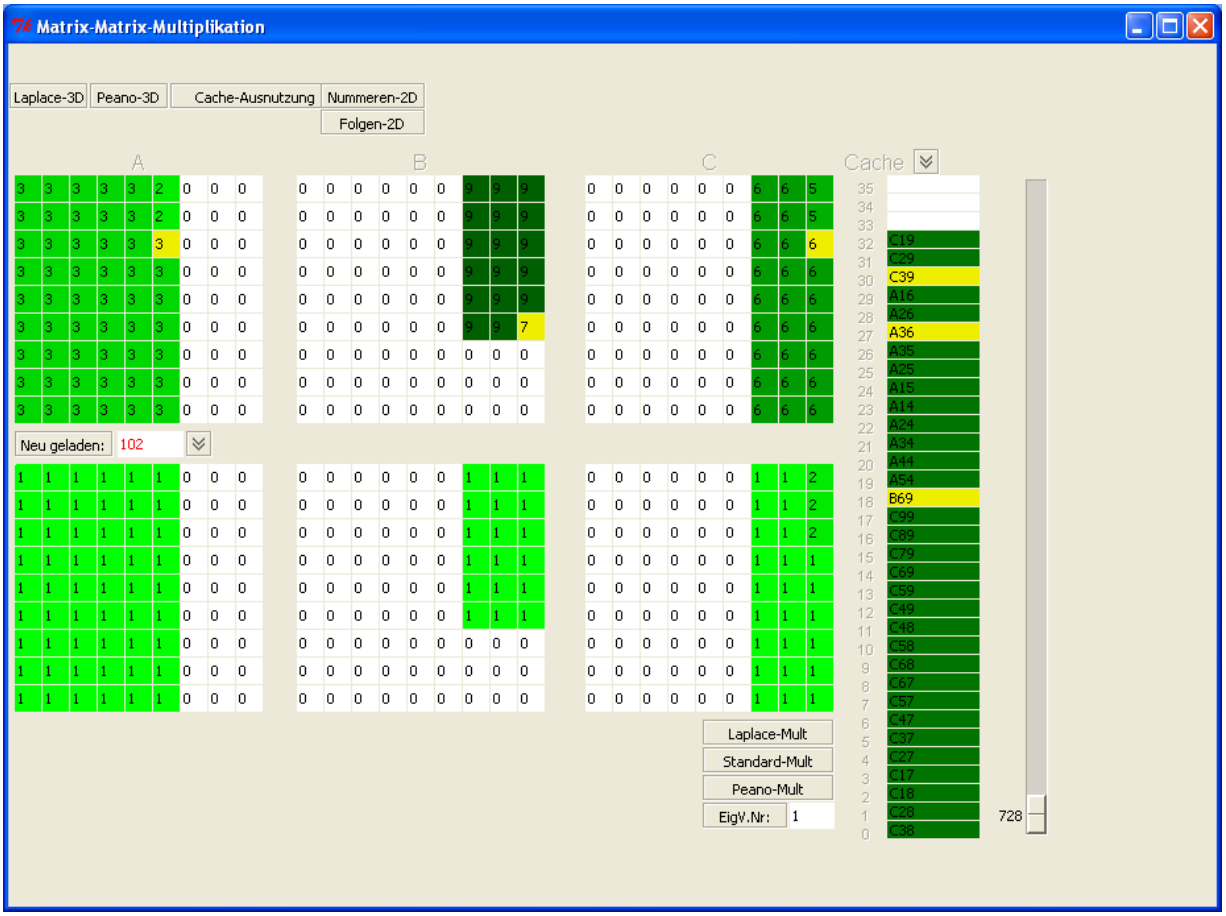


Abbildung 4.1: Analysetool.

In Abbildung 4.1 sieht man drei Matrizen, die wir durch eine bestimmte Multiplikationsvariante miteinander multiplizieren wollen. Durch die drei Buttons »Laplace-Mult«,

»Standard-Mult« und »Peano-Mult« kann man die Multiplikation starten. Der Multiplikationsverlauf kann visuell Schritt für Schritt beobachtet werden. Unter den Feldern, auf welchen die Buchstaben A, B und C stehen, sieht man welche Matrixeinträge bis jetzt besucht und bearbeitet wurden. Gelb zeigt auf Elemente, die gerade bearbeitet werden, und grün steht für Operanden, die schon zuvor benötigt wurden.

Rechts können wir sehen, ob diese Operanden im Cache vorhanden sind oder nicht. Durch die Farbe gelb sieht man im Cache die Elemente, die gerade benötigt werden. Unter dem Label »Cache« kann die Cache-Auslastung nach jeder Operation beobachtet werden, sowie welche Elemente in dem Cache an welcher Stelle stehen.

Mit dem »Slider« rechts vom Cache kann ich bestimmen, ob die Multiplikationsfolge komplett bis $N^3 - 1$ durchgelaufen wird oder nur bis zu einem Zeitpunkt $t < N^3 - 1$.

Neben dem Button »Neu geladen« steht die Anzahl der Neuladungen vom Hauptspeicher in den Cache. Diese Anzahl wird bei jeder Operation aktualisiert. In den Feldern darunter steht wie oft jeder Operand insgesamt neu geladen wurde.

Kurze Beschreibung der restlichen Buttons in Abbildung 4.1:

1. »Laplace-3D« – Darstellung der erzeugten Laplace-Folge $\pi_{Laplace}$ in 3D, siehe Abbildung 5.13.
2. »Peano-3D« – Darstellung der Peano-Folge π_{Peano} in 3D, siehe Abbildung 3.9.
3. Der Button »Cache-Ausnutzung« stellt die Cache-Ausnutzung im Abhängigkeit von der Cache-Größe m dar. Blau für π_{Peano}, rot für $\pi_{Standard}$ und grün für $\pi_{Laplace}$, siehe Abbildung 5.17.
4. »Nummern-2D« und »Folgen-2D« – Nummerierung der Operationen in den drei Multiplikationsfolgen, siehe Abbildungen 5.14 und 5.15.

5 Laplace-Multiplikation

Durch das LE-Verfahren wollen wir in diesem Abschnitt eine neue Multiplikationsfolge $\pi_{Laplace}$ erzeugen. Dabei werden die Operationen der Matrix-Matrix-Multiplikation als 3D-Koordinaten betrachtet und somit wie in der Einführung als Knoten eines Graphen $G := (V, E)$ dargestellt:

$$Operation : ijk \Rightarrow Koordinate : (i, j, k) \in \mathbb{N}^3 \Rightarrow Knoten : v_{ijk} \in V \tag{5.1}$$

Man erhält insgesamt N^3 verschiedene Knoten aus N^3 verschiedenen Operationen. Zwei Knoten sind benachbart, falls ihre zugehörigen Operationen einen gemeinsamen Operanden besitzen. D.h. $v_{i_0 j_0 k_0}$ und $v_{i_1 j_1 k_1}$ sind durch eine Kante verbunden, falls:

$((i_0 \neq i_1) \wedge (j_0 = j_1) \wedge (k_0 = k_1)) \vee$
$((i_0 = i_1) \wedge (j_0 \neq j_1) \wedge (k_0 = k_1)) \vee$
$((i_0 = i_1) \wedge (j_0 = j_1) \wedge (k_0 \neq k_1))$

Jede Operation ijk hat eine eindeutige Nummer $n \in \{1, 2, \ldots, N^3\}$ (Die aufsteigende Numerierung entsprecht die Permutation $id = \pi_{Standard} = (1, 2, \ldots, N^3)$). Dies ermöglicht uns ganz einfach die Adjazenzmatrix W zu erstellen, denn $W[i, j]$ soll das Gewicht zwischen der Operationen mit den Nummern $n_0 = i + 1$ und $n_1 = j + 1$ enthalten.

$$W[i, j] = |i - j|^{var}, var \in \mathbb{R} \tag{5.2}$$

In der Gleichung 5.2 haben wir die euklidische Distanz zwischen n_0 und n_1, abhängig von der Variable *var*[1]. Durch *var* werden wir später das Gewicht zwischen allen benachbarten Knoten gleichmäßig beeinflussen. Man erhält z.B. für $var = 0$ überall Gewichte der Größe 1. Für $var > 0$ erhalten größere Distanzen mehr Gewicht als kleinere Distanzen und genau umgekehrt für $var < 0$.

Die folgende Funktion *adjacent* liefert *True*, falls n_0 und n_1 benachbart sind, ansonsten *False*.

```
def adjacent(n0,n1):
   [i0,j0,k0]=ijk(n0,N)
   [i1,j1,k1]=ijk(n1,N)
   return ((i0 !=i1)and(j0 ==j1)and(k0 ==k1))or
          ((i0 ==i1)and(j0 !=j1)and(k0 ==k1))or
          ((i0 ==i1)and(j0 ==j1)and(k0 !=k1))
```

[1] *var* steht für Variante.

Welche Gewichte benachbarte Operationen bekommen, bestimmt man durch die Funktion *weight*.

```
def weight(n0,n1,var):
   if adjacent(n0,n1):
      return power(float(abs(n0-n1)), var)
   else:
      return 0.0
```

5.1 Erste Gewichtungsvariante

Zuerst wollen wir uns den Fall anschauen mit $var = 0$. Dabei sind alle Einträge von der Matrix W entweder Null oder Eins, also Eins für benachbarte Knoten, sonst Null.

Für $N = 2$ sehen die Matrizen W, D und L wie im Folgenden aus.

```
# Die Adjazenzmatrix W mit N = 2 und var = 0.
                        [[ 0.,  1.,  1.,  0.,  1.,  0.,  0.,  0.],
                         [ 1.,  0.,  0.,  1.,  0.,  1.,  0.,  0.],
                         [ 1.,  0.,  0.,  1.,  0.,  0.,  1.,  0.],
                         [ 0.,  1.,  1.,  0.,  0.,  0.,  0.,  1.],
                         [ 1.,  0.,  0.,  0.,  0.,  1.,  1.,  0.],
                         [ 0.,  1.,  0.,  0.,  1.,  0.,  0.,  1.],
                         [ 0.,  0.,  1.,  0.,  1.,  0.,  0.,  1.],
                         [ 0.,  0.,  0.,  1.,  0.,  1.,  1.,  0.]]
```

```
# Die Diagonalmatrix D: D_ii = 3 * N - 3 mit N = 2 und var = 0.
                        [[ 3.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
                         [ 0.,  3.,  0.,  0.,  0.,  0.,  0.,  0.],
                         [ 0.,  0.,  3.,  0.,  0.,  0.,  0.,  0.],
                         [ 0.,  0.,  0.,  3.,  0.,  0.,  0.,  0.],
                         [ 0.,  0.,  0.,  0.,  3.,  0.,  0.,  0.],
                         [ 0.,  0.,  0.,  0.,  0.,  3.,  0.,  0.],
                         [ 0.,  0.,  0.,  0.,  0.,  0.,  3.,  0.],
                         [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  3.]]
```

```
# Die Matrix L = D − W mit N = 2 und var = 0.
                        [[ 3., -1., -1.,  0., -1.,  0.,  0.,  0.],
                         [-1.,  3.,  0., -1.,  0., -1.,  0.,  0.],
                         [-1.,  0.,  3., -1.,  0.,  0., -1.,  0.],
                         [ 0., -1., -1.,  3.,  0.,  0.,  0., -1.],
                         [-1.,  0.,  0.,  0.,  3., -1., -1.,  0.],
                         [ 0., -1.,  0.,  0., -1.,  3.,  0., -1.],
                         [ 0.,  0., -1.,  0., -1.,  0.,  3., -1.],
                         [ 0.,  0.,  0., -1.,  0., -1., -1.,  3.]]
```

Allgemein gilt bei dieser Variante $D_{ii} = 3N - 3$ und $D_{ii}^{-1} = \frac{1}{3N-3}$. Das heißt, die Diagonal-Matrix D enthält an der Diagonale nur die Zahl $3N - 3$. Daraus folgt, dass $A = D^{-1}L = \frac{1}{3N-3}L$ eine symmetrische Matrix ist.

```
# A = 1/(3N−3) L mit N = 2.
[[ 1.       ,-0.333333,-0.333333, 0.       ,-0.333333, 0.       ,0.       , 0.       ],
 [-0.333333, 1.       , 0.       ,-0.333333, 0.       ,-0.333333, 0.       , 0.       ],
 [-0.333333, 0.       , 1.       ,-0.333333, 0.       , 0.       ,-0.333333, 0.       ],
 [ 0.       , 0.333333,-0.333333, 1.       , 0.       , 0.       , 0.       ,-0.333333],
 [-0.333333, 0.       , 0.       , 0.       , 1.       ,-0.333333,-0.333333, 0.       ],
 [ 0.       ,-0.333333, 0.       , 0.       ,-0.333333, 1.       , 0.       ,-0.333333],
 [ 0.       , 0.       ,-0.333333, 0.       ,-0.333333, 0.       , 1.       ,-0.333333],
 [ 0.       , 0.       , 0.       ,-0.333333, 0.       ,-0.333333,-0.333333, 1.       ]]
```

Wenn man jetzt die Eigenwerte und Eigenvektoren von A berechnet, sieht man sofort, dass der erste Eigenwert, der größer Null ist, $3(N-1)$-Mal vorkommt. Eine Erklärung dafür können wir aus dem erstellten Graph G lesen, denn in G hat jeder Knoten genau $3(N-1)$ Kanten, die alle mit 1 gewichtet sind (siehe Abbildung 5.1).

Ein Knoten entspricht einer Operation und jede Operation hat 3 Operanden. Jeder dieser Operanden kommt in allen Operationen N-Mal vor, d.h. der Knoten v_{ijk} entspricht der Operation $C_{ij} += A_{ik} * B_{kj}$. Wegen des Operanden C_{ij} wird v_{ijk} mit allen $N-1$ anderen Operationen, in denen C_{ij} vorkommt, verbunden. Das Gleiche gilt für A_{ik} und B_{kj}. Somit erhalten wir insgesamt für jeden Knoten $3(N-1)$ verschiedene Nachbarn.

```
# Eigenwerte von A mit N = 2.
[  1.06794662e-16   6.66666667e-01   6.66666667e-01   6.66666667e-01
   1.33333333e+00   1.33333333e+00   1.33333333e+00   2.00000000e+00]
```

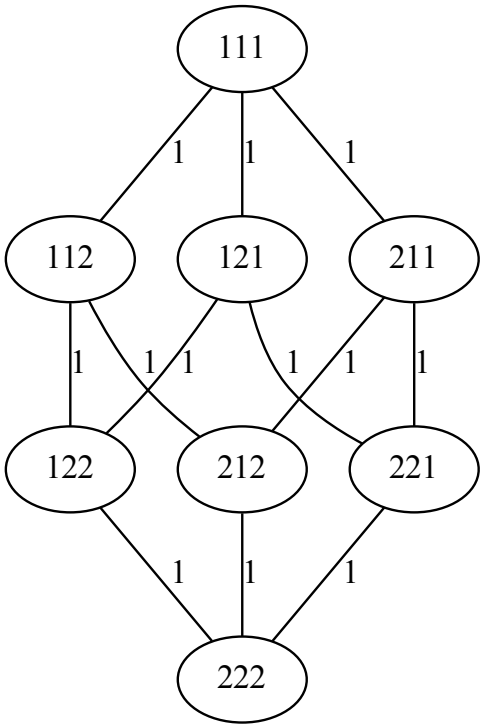
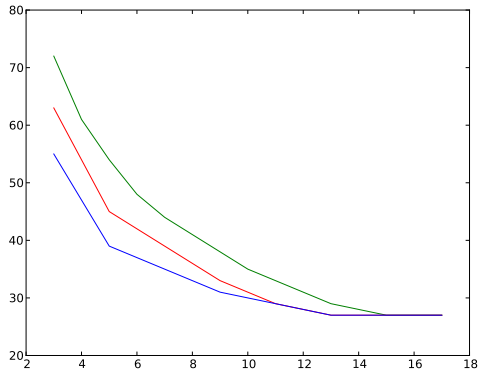


Abbildung 5.1: Graph der Matrix-Matrix-Multiplikation für $N = 2$ und $var = 0.0$.

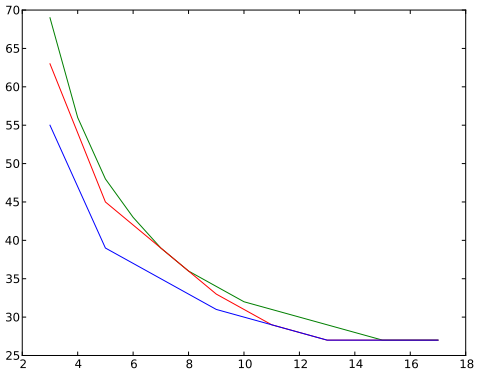
```
# Eigenwerte von A mit N = 3.
[  3.32707242e-16   5.00000000e-01   5.00000000e-01   5.00000000e-01
   5.00000000e-01   5.00000000e-01   5.00000000e-01   1.00000000e+00
   1.00000000e+00   1.00000000e+00   1.00000000e+00   1.00000000e+00
   1.00000000e+00   1.00000000e+00   1.00000000e+00   1.00000000e+00
   1.00000000e+00   1.00000000e+00   1.00000000e+00   1.50000000e+00
   1.50000000e+00   1.50000000e+00   1.50000000e+00   1.50000000e+00
   1.50000000e+00   1.50000000e+00   1.50000000e+00]
```

Logischerweise hat hier das LE-Verfahren an jedem Knoten $3(N-1)$ gleichwertige Wege zu verfolgen. Daher bekommt man anstelle von einem eindeutigen Eigenvektor einen Eigenraum, der die Dimension $3(N-1)$ hat.

Abbildung 5.2 zeigt uns die Cache-Ausnutzung für zwei zufällig gewählte Eigenvektoren aus dem ermittelten Eigenraum für $N = 3$. Grün steht für $\pi_{Laplace}$, rot für die Standard-



(a) möglicher Eigenvektor



(b) möglicher Eigenvektor

Abbildung 5.2: $\sigma_m(\pi_{Laplace}, 27)$ in grün, $\sigma_m(\pi_{Standard}, 27)$ in rot, $\sigma_m(\pi_{Peano}, 27)$ in blau für verschiedenen Cache-Größen m.

Folge und blau für die Peano-Folge. Auf der X-Achse stehen die verschiedenen Cache-Größen m. Die Y-Achse stellt die Cache-Ausnutzung $\sigma_m(\pi, N^3)$ dar.

Wie man in Abbildung 5.2 sieht, liefert diese Variante keine gute Folge im Vergleich zu den anderen Standard- und Peano-Folgen. Wir wissen aber, dass ein sechsdimensionaler Eigenraum unendlich viele Eigenvektoren hat, und wir nur zwei zufällig gewählte Eigenvektoren aus diesem Eigenraum untersucht haben.

5.2 Eigenraumoptimierung

Um zu besseren Eigenvektoren zu gelangen, müssen wir mehrere Eigenvektoren miteinander kombinieren. Wir kombinieren zwei Eigenvektoren EV_0 und EV_1 durch den Winkel α miteinander und erhalten so wiederum einen Eigenvektor EV, der in dem gleichen Eigenraum liegt.

$$EV = \sin(\alpha)EV_0 + \cos(\alpha)EV_1 \tag{5.3}$$

Das Ganze soll für mehrere Winkel Alpha berechnet und jedes Mal auf Cache-Ausnutzung hin getestet werden. Aus diesem Verlauf erhalten wir einen Eigenvektor EV_{opt}, der mindestens so gut wie EV_0 oder EV_1 ist . Dieses Ergebnis EV_{opt} können wir dann mit weiteren zufälligen Eigenvektoren kombinieren usw.

Durch dieses Vorgehen bekommen wir auf jeden Fall ein besseres Ergebnis als in der Abbildung 5.2. Einen möglichen Eigenvektor EV_{opt} stellt die Abbildung 5.3 dar.

Die Suche nach einen besseren EV_{opt} kann man selbst beeinflussen, nämlich durch die Anzahl der Eigenvektoren, die man miteinander kombiniert, sowie durch die Auswahl von verschiedenen Winkel α. Obwohl ich den Fall für kleine $N = 3$ betrachte, finde ich keine eindeutige Strategie, welche am Ende das wirkliche Optimum in diesem Eigenraum liefert.

Für größere N oder sogar für $N = 3$ dauert diese Suche unendlich lange. Von daher werden wir an dieser Stelle diesen Weg nicht weiter verfolgen und uns eine andere Variante anschauen, die uns nur einen einzigen Eigenvektor liefert.

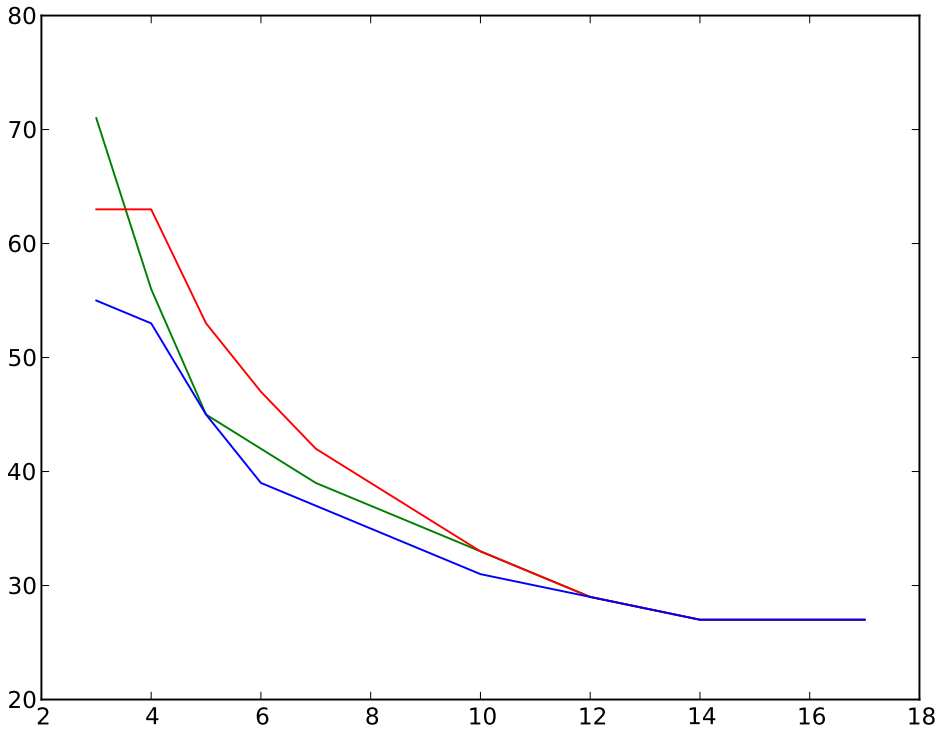


Abbildung 5.3: Eigenraumoptimierung für $N = 3$.

5.3 Zweite Gewichtungsvariante $var \neq 0.0$

In diesem Abschnitt wollen wir eine Strategie verfolgen, die uns nur einen einzigen Eigenvektor liefert. Dadurch wird die Suche nach einem Optimum in einem $3(N-1)$-dimensionalen Eigenraum vermieden. Die Idee ist ganz einfach. Kanten, die aus einem

Knoten v_{ijk} gehen, sollen unterschiedliche Gewichte bekommen. Eine Möglichkeit wäre die Gewichtung nach der Gleichung 5.4 zu belegen.

$$W[i,j] = |i-j| \tag{5.4}$$

Wir erhalten die Gleichung 5.4, indem wir in der Gleichung 5.2 für var den Wert 1 setzen.

```
# Die Adjazenzmatrix W mit N = 2 und var = 1.0.
                        [[ 0.,  1.,  2.,  0.,  4.,  0.,  0.,  0.],
                         [ 1.,  0.,  0.,  2.,  0.,  4.,  0.,  0.],
                         [ 2.,  0.,  0.,  1.,  0.,  0.,  4.,  0.],
                         [ 0.,  2.,  1.,  0.,  0.,  0.,  0.,  4.],
                         [ 4.,  0.,  0.,  0.,  0.,  1.,  2.,  0.],
                         [ 0.,  4.,  0.,  0.,  1.,  0.,  0.,  2.],
                         [ 0.,  0.,  4.,  0.,  2.,  0.,  0.,  1.],
                         [ 0.,  0.,  0.,  4.,  0.,  2.,  1.,  0.]]
```

```
# Die Diagonalmatrix D mit N = 2 und var = 1.0.
                        [[ 7.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
                         [ 0.,  7.,  0.,  0.,  0.,  0.,  0.,  0.],
                         [ 0.,  0.,  7.,  0.,  0.,  0.,  0.,  0.],
                         [ 0.,  0.,  0.,  7.,  0.,  0.,  0.,  0.],
                         [ 0.,  0.,  0.,  0.,  7.,  0.,  0.,  0.],
                         [ 0.,  0.,  0.,  0.,  0.,  7.,  0.,  0.],
                         [ 0.,  0.,  0.,  0.,  0.,  0.,  7.,  0.],
                         [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  7.]]
```

```
# Die Matrix L = D - W mit N = 2 und var = 1.0.
                        [[ 7., -1., -2.,  0., -4.,  0.,  0.,  0.],
                         [-1.,  7.,  0., -2.,  0., -4.,  0.,  0.],
                         [-2.,  0.,  7., -1.,  0.,  0., -4.,  0.],
                         [ 0., -2., -1.,  7.,  0.,  0.,  0., -4.],
                         [-4.,  0.,  0.,  0.,  7., -1., -2.,  0.],
                         [ 0., -4.,  0.,  0., -1.,  7.,  0., -2.],
                         [ 0.,  0., -4.,  0., -2.,  0.,  7., -1.],
                         [ 0.,  0.,  0., -4.,  0., -2., -1.,  7.]]
```

```
# Die Symmetrische Matrix A mit N = 2 und var = 1.0.
[[ 1.       ,-0.142857,-0.285714, 0.       ,-0.571428, 0.       , 0.       , 0.       ],
 [-0.142857, 1.       , 0.       ,-0.285714, 0.       ,-0.571428, 0.       , 0.       ],
 [-0.285714, 0.       , 1.       ,-0.142857, 0.       , 0.       ,-0.571428, 0.       ],
 [ 0.       ,-0.285714,-0.142857, 1.       , 0.       , 0.       , 0.       ,-0.571428],
 [-0.571428, 0.       , 0.       , 0.       , 1.       ,-0.142857,-0.285714, 0.       ],
 [ 0.       ,-0.571428, 0.       , 0.       ,-0.142857, 1.       , 0.       ,-0.285714],
 [ 0.       , 0.       ,-0.571428, 0.       ,-0.285714, 0.       , 1.       ,-0.142857],
 [ 0.       , 0.       , 0.       ,-0.571428, 0.       ,-0.285714,-0.142857, 1.       ]]
```

```
# Eigenwerte von A fuer  N = 2 und var = 1.0.
[  2.63677968e-16   2.85714286e-01   5.71428571e-01   8.57142857e-01
   1.14285714e+00   1.42857143e+00   1.71428571e+00   2.00000000e+00]
```

Zwei benachbarte Operationen unterscheiden sich letztendlich nur um den Index. Deswegen können wir bei dieser Variante ihren euklidischen Abstand ohne die zugehörigen Nummern ermitteln:

1. Falls die Operationen sich nur durch den ersten Index unterscheiden, dann gilt für ihre Distanz: $\|ijk - i_0jk\| = |i - i_0| * N^2$.
2. Durch den zweiten Index: $\|ijk - ij_0k\| = |j - j_0| * N$
3. Durch den dritten Index: $\|ijk - ijk_0\| = |k - k_0|$

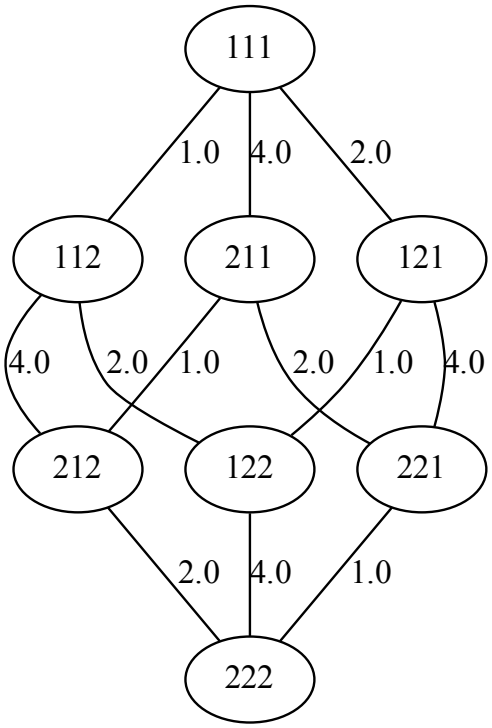


Abbildung 5.4: Graph der Matrix-Matrix-Multiplikation für $N = 2$ und $var = 1.0$.

Aus dieser Betrachtung wissen wir, dass Operationen, die sich bei dem ersten Index unterscheiden (das sind Operationen, deren gemeinsamer Operand B_{kj} ist), die großen Gewichte im Graph G bekommen. An zweiter Stelle kommen die Operationen, die A_{ik} als gemeinsamen Operand haben, und die ganz kleinen Gewichte erhalten die Operationen, die C_{ij} als gemeinsamen Operand haben. Dieses Verhältnis zwischen den Gewichten gilt nicht nur für $var = 1$, sondern allgemein für $var > 0$. Von daher wirkt sich der Wert von var, falls $var > 0$ ist, kaum auf das Resultat $\pi_{Laplace}$ von LE aus.

Falls $var < 0$ ist, so erhalten wir eine andere Multiplikation-Folge $\pi_{Laplace}$ wie bei $var > 0$, aber in beiden Fällen die gleiche Cache-Ausnutzung $\sigma_m(\pi_{Laplace}, N^3)$. Wie diese zwei Folgen für $var > 0$ und $var < 0$ aussehen, wollen wir anhand eines Beispiel für $N = 9$ anschauen.

In Abbildung 5.5 werden folgende Operationen hintereinander ausgeführt:

$C_{19} += A_{16} * B_{69} \rightarrow 196$

$C_{99} += A_{96} * B_{69} \rightarrow 996$

$C_{11} += A_{14} * B_{41} \rightarrow 114$

$C_{11} += A_{16} * B_{61} \rightarrow 116$

$C_{91} += A_{96} * B_{61} \rightarrow 916$

$C_{91} += A_{94} * B_{41} \rightarrow 914$

$C_{19} += A_{14} * B_{49} \rightarrow 194$

$C_{99} += A_{94} * B_{49} \rightarrow 994$

Hier sieht man, dass das Verfahren LE mit den benachbarten Operationen, die am weitesten auseinander liegen, anfängt.

$\|196 - 996\| = \|114 - 994\| = \|116 - 916\| = \|194 - 994\| = 8 * 9^2 = 648.$

In den nächsten drei Abbildungen 5.6, 5.7 und 5.8 ist das LE-Prinzip für $var > 0$ deutlich zu sehen, nämlich an der Matrix C. Dort sieht man, wie die Felder von außen nach Innen nacheinander bearbeitet werden. Dasselbe passiert auch in A und B, aber nur spalten- bzw. zeilenweise. Ist die Matrix C von außen nach innen komplett durchgelaufen, dann wiederholt sich dieser Vorgang wieder von innen nach außen. Für $N = 9$ und allgemein für ungerade N kommt es immer nur einmal vor, dass die Matrix C von außen nach innen durchlaufen wird.

Ist $var < 0$ (siehe Abbildungen 5.9, 5.10, 5.11 und 5.12), so werden die Matrizen A, B und C wiederholt von außen nach innen durchlaufen.

Dieser Durchlauf von innen nach außen oder umgekehrt spielt bei der Cache-Ausnutzung beider Folgen nur eine minimale Rolle.

Man kann sich natürlich an dieser Stelle viele Gedanken darüber machen, wie man das Ergebnis verbessern kann. Etwa wenn wir beispielsweise wiederholt einmal von innen und einmal von außen die Matrix durchlaufen. Oder vielleicht, wenn wir LE rekursiv auf gleichmäßig geteilte Blöcke laufen lassen. Dadurch können wir auf jeden Fall das Resultat $\pi_{Laplace}$ verbessern und somit bessere Cache-Ausnutzung erzielen. Aber die Optimierung und die Ergebnisverbesserung sind nicht das Ziel dieser Arbeit. Wir interessieren uns viel mehr für das Analysieren der resultierenden $\pi_{Laplace}$ Folge. D.h. wie gut ist diese Folge $\pi_{Laplace}$ im Vergleich zu $\pi_{Standard}$ und π_{Peano}.

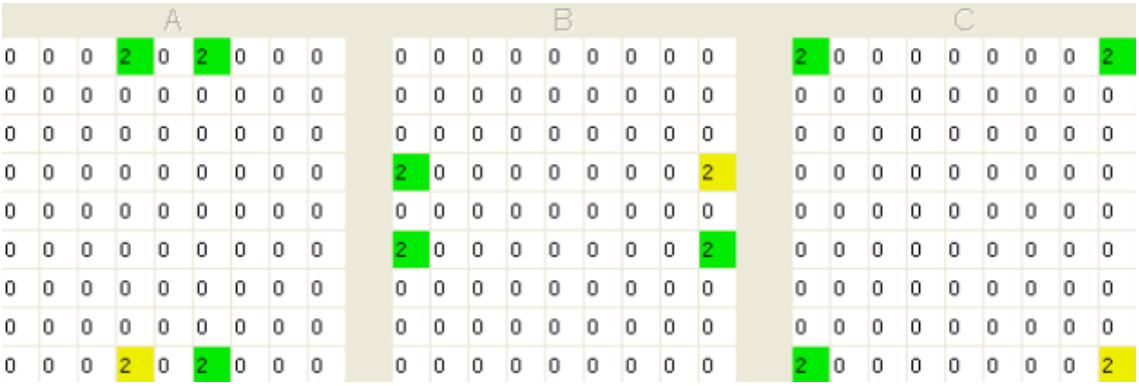


Abbildung 5.5: Laplace-Multiplikation $N = 9$ und $var = 1.0$. Die Operationen $0 \ldots 7$.

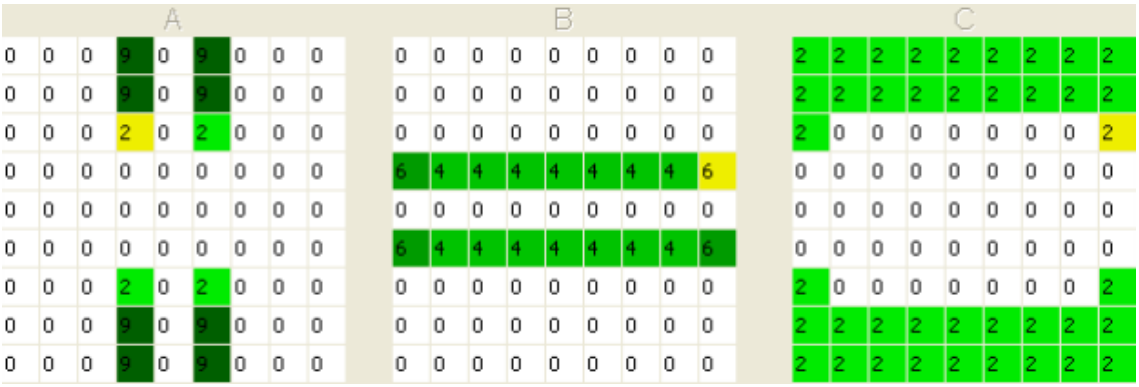


Abbildung 5.6: Laplace-Multiplikation $N = 9$ und $var = 1.0$. Die Operationen $0 \ldots 79$.

5.3.1 Cache-Ausnutzung von $\pi_{Laplace}$

Wir würden eine Folge erwarten, die ähnlich wie π_{Peano} verläuft. D.h., einen Pfad, der nur Kanten aus dem erstellten Graph G hat. Aber wie wir in Abbildung 5.13 sehen, liefert LE zwar einen Pfad, der im ersten Augenblick sehr komplex aussieht, aber trotzdem über

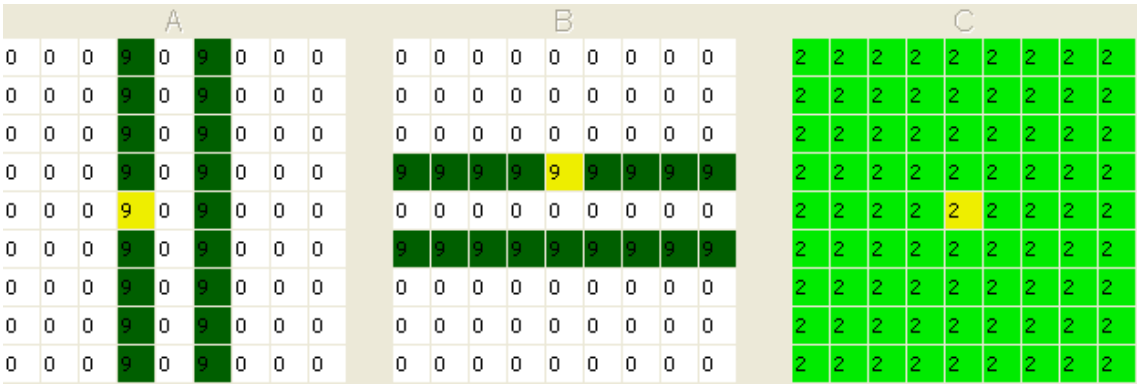


Abbildung 5.7: Laplace-Multiplikation $N = 9$ und $var = 1.0$. Die Operationen $0 \ldots 181$.

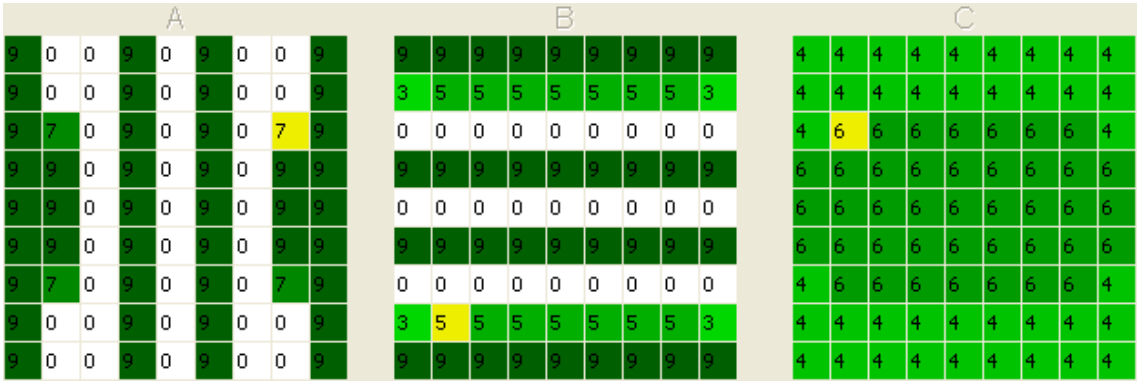


Abbildung 5.8: Laplace-Multiplikation $N = 9$ und $var = 1.0$. Die Operationen $0 \ldots 405$.

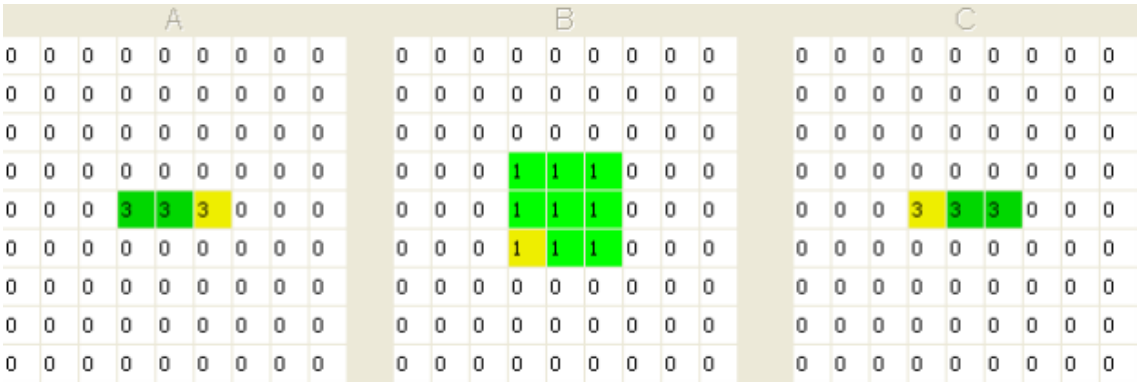


Abbildung 5.9: Laplace-Multiplikation $N = 9$ und $var = -1.0$. Die Operationen $0 \ldots 8$.

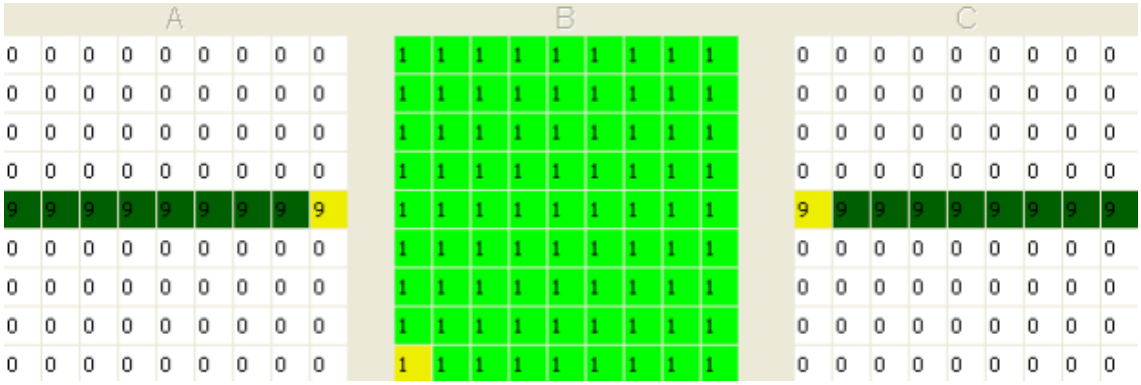


Abbildung 5.10: Laplace-Multiplikation $N = 9$ und $var = -1.0$. Die Operationen $0 \ldots 80$.

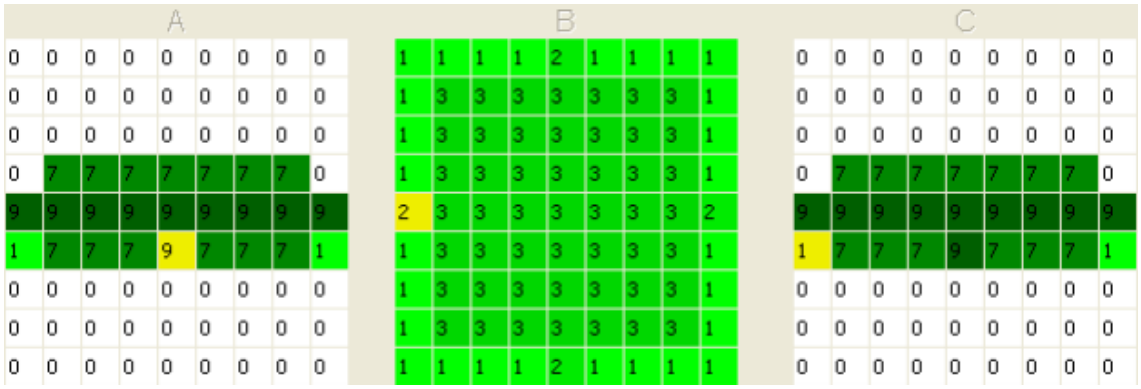


Abbildung 5.11: Laplace-Multiplikation $N = 9$ und $var = -1.0$. Die Operationen $0 \ldots 182$.

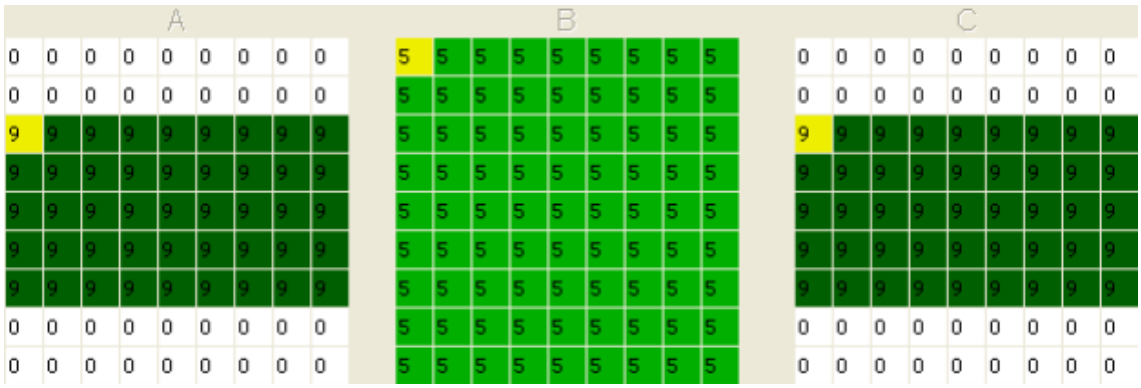


Abbildung 5.12: Laplace-Multiplikation $N = 9$ und $var = -1.0$. Die Operationen $0 \ldots 404$.

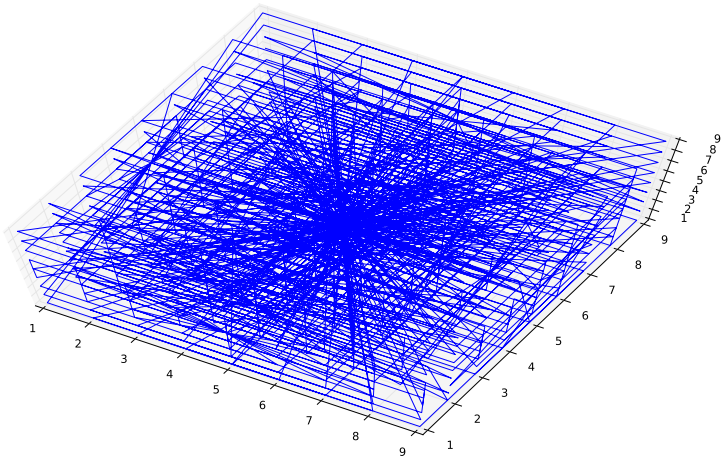


Abbildung 5.13: Laplace-Pfad für $N = 9$ und $var = 1.0$.

70% Kanten aus G enthält. Eine bessere Sicht zeigen uns die Abbildungen 5.14 und 5.15. Hier sieht man auf der y-Achse die Nummern der Operationen und auf der x-Achse die Stellen der Operationen in π.

Die Cache-Ausnutzung von $\pi_{Laplace}$ wird auch grafisch dargestellt. Wir werden den Fall für Matrizen der Größe $N = 9$ zeigen. Abbildung 5.16 beschränkt sich nur auf einen Cache der Größe 36 (also viermal größer als N) und zeigt uns den Vergleich zwischen den Peano- (blau), Standard- (rot) und Laplace-Folge (grün). Hier wird die Cache-Ausnutzung in Abhängigkeit von t (x-Achse) dargestellt, d.h wie groß die Cache-Ausnutzung nach Ausführung der Operation $\pi(t)$ ist.

In Abbildung 5.17 wird die Cache-Ausnutzung für verschiedene Cache-Größen (x-Achse), und zwar nach Ausführung aller N^3-Operationen, dargestellt.

Das LE-Verfahren liefert sozusagen eine Folge, die deutlich besser als die Standard-Folge ist und eine ähnliche Cache-Ausnutzung-Kurve zu π_{Peano} hat. Man sieht auch in Abbildung 5.17, dass die π_{Peano} eine bessere Cache-Ausnutzung-Kurve als $\pi_{Laplace}$ hat.

5.3.2 Vorteile und Nachteile

Vorteile:

1. Die Multiplikation-Folge $\pi_{Laplace}$ hat eine sehr gute Cache-Ausnutzung im Vergleich zu $\pi_{Standard}$.

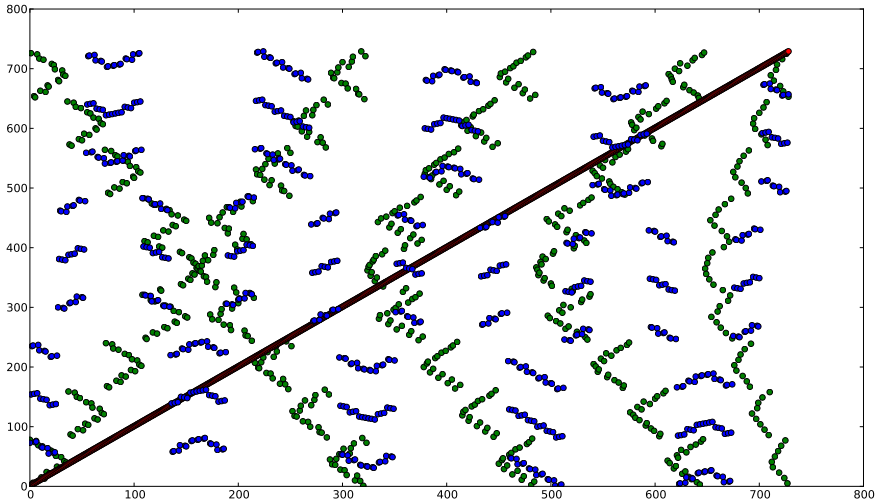


Abbildung 5.14: Nummerierung der Operationen. $N = 9$ mit grün für Laplace, blau für Peano und rot für Standard.

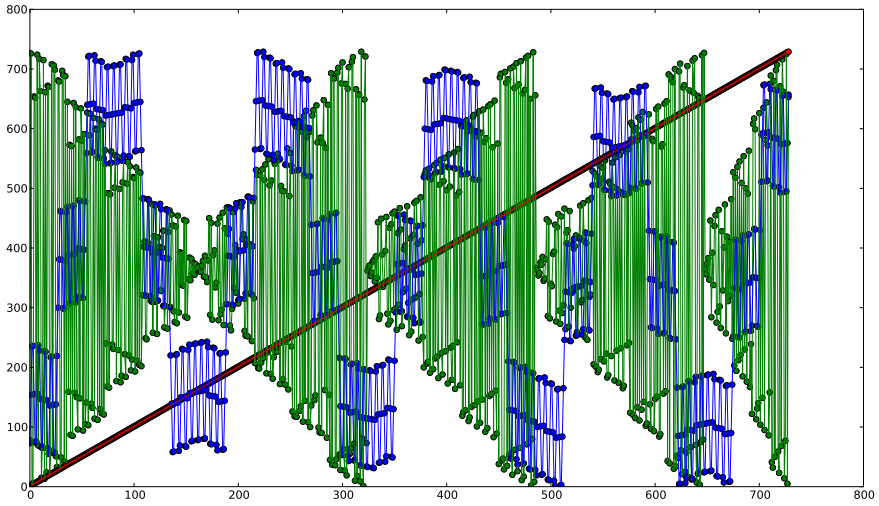


Abbildung 5.15: Folge der Operationen. $N = 9$ mit grün für Laplace, blau für Peano und rot für Standard.

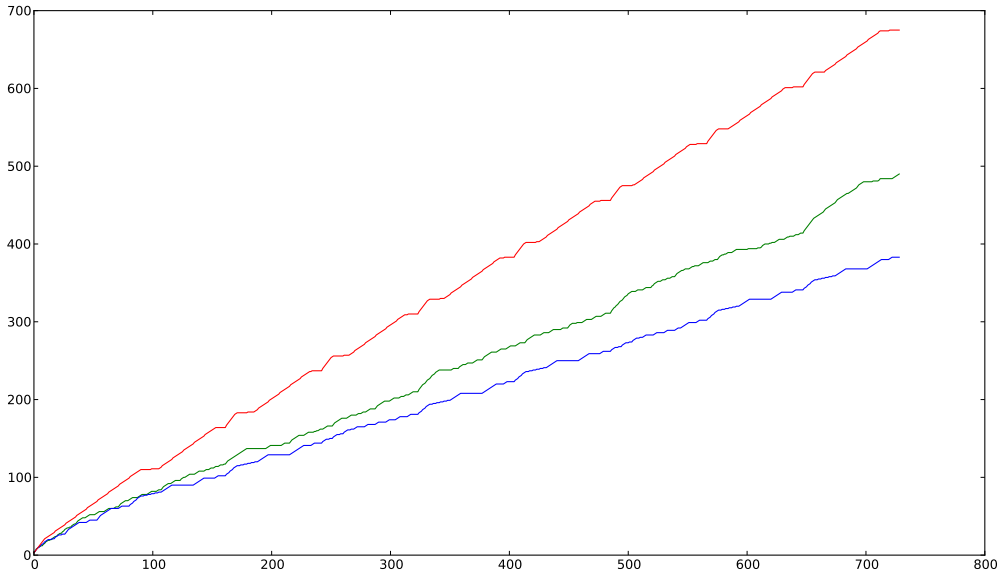


Abbildung 5.16: Cache-Ausnutzung für Matrizen der Größe $N = 9$, Cache-Größe $m = 4 * N$ und $t = 0 \ldots N^3 - 1$.

2. Der Verlauf von $\pi_{Laplace}$ hat eine Struktur, die sehr leicht ohne LE implementiert werden kann. Im Vergleich zu π_{Peano} ist diese Implementierung sehr einfach und für alle Matrixgrößen möglich.
3. Man kann $\pi_{Laplace}$ weiter verbessern, indem man z.B. LE rekursiv erst auf kleine Blöcke und anschließend auf größere Blöcke anwendet.

Nachteile:

1. Die Multiplikation-Folge $\pi_{Laplace}$ hat zwar eine ähnliche Cache-Ausnutzung-Kurve (siehe Abbildung 5.17) wie π_{Peano}, aber mit etwas schlechterer Cache-Ausnutzung.
2. Dadurch, dass der Graph G symmetrisch ist, hat LE an vielen Stellen mehrere Operationen zur Auswahl. Daher enthält der resultierende Pfad viele Kanten, die nicht aus G sind. Dies verursacht viele Sprünge während des Multiplikationsverlaufs. Mit einem Sprung meine ich, dass zwei Operationen, die in $\pi_{Laplace}$ nebeneinander stehen, Operanden enthalten, die in den jeweiligen Matrizen weit auseinander liegen. Dieses Verhalten kommt in π_{Peano} nicht vor.

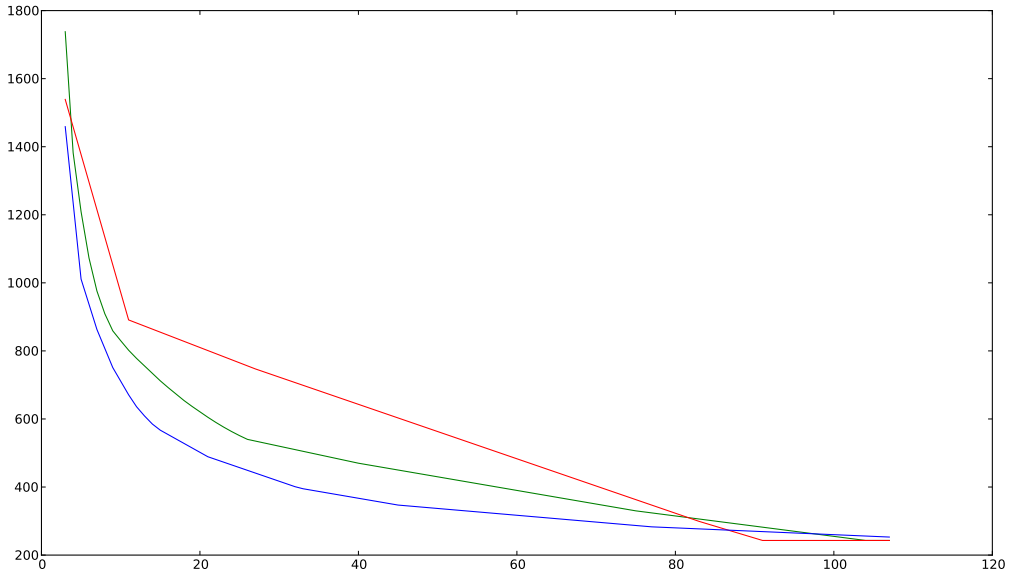


Abbildung 5.17: Cache-Ausnutzung für verschiedenen Cache-Größen und Matrizen der Größe $N = 9$.

6 Zusammenfassung und Ausblick

Zusammenfassung

Mit dem Laplacian-Eigenmap-Verfahren kann man auf eine bestimmte Punktmenge im Raum eine Struktur oder eine Ordnung einführen. In dieser Arbeit wurde eine spezielle Ordnung verfolgt, die eine Punktmenge so miteinander ordnet, dass Punkte, die nah beieinander liegen, zusammen gehören. Dieses Vorgehen wurde detailliert und anhand vieler Beispiele vorgestellt.

Weiterhin wurden Operationen der Matrix-Matrix-Multiplikation als 3D-Koordinaten betrachtet und eine neue Struktur darauf definiert, die Operationen miteinander ordnet, die den gleichen Operanden besitzen. Die Anwendung des Laplacian-Eigenmap-Verfahren auf diesen 3D-Koordinaten und ihre neue Struktur liefert als Ausgabe die Operationen der Matrix-Matrix-Multiplikation in einer für den Cache sehr effizienten Reihenfolge.

Diese erstellte Multiplikationsfolge, die ich Laplace-Folge genannt habe, wurde in dieser Arbeit einer vergleichenden Analyse mit zwei anderen, etablierten Verfahren unterzogen. Im Vergleich zu der konservativen Matrix-Matrix-Multiplikation [1] führt die erwähnte Multiplikationsfolge zu einer deutlich besseren Cache-Ausnutzung, die einer Cache-Ausnutzung-Kurve wie bei der Peano-Multiplikation-Folge [2] ähnelt und nahekommt, jedoch etwas schlechtere Werte aufweist.

Ausblick

Eine sehr gute Cache-Ausnutzung liefert uns bekannterweise die Peano-Multiplikation. Sie ist nur eine von $N^3!$ verschiedenen Multiplikationsfolgen, unter denen manche sehr schlecht sind, manche genau so gut wie die Peano-Multiplikation sind und es kann auch vorkommen, dass es bessere Folgen gibt! Für Matrizen der Größe 2 habe ich alle Folgen (insgesamt $2^3! = 8! = 40320$) auf Cache-Ausnutzung untersucht. Es gab keine, die besser war als die Peano-Multiplikation, aber es gab viele, die genau so gut waren. Für $N = 3$ ist

[1] Zeile aus Matrix A multipliziert mit Spalte aus Matrix B.
[2] Beruht auf die Peano-Kurve und bekannt für ihre effiziente Cache-Ausnutzung.

dieser Weg unmöglich, da das Ganze ungefähr 10^{20} Jahren dauern würde, unter der Voraussetzung, dass eine einzige Folge für die Cache-Ausnutzung-Analyse nur eine Sekunde Rechenzeit braucht. Von daher war der Versuch mit dem Laplacian-Eigenmap-Verfahren so wertvoll und hilfreich, sodass man einschätzen kann, wie Multiplikationsfolgen mit guter Cache-Ausnutzung aussehen können.

Man kann auf jeden Fall die resultierende Folge weiter verbessern, indem man überlegte Umformungen und Änderungen an $\pi_{Laplace}$ direkt durchführt. Zum Beispiel könnte ich durch Umdrehen[3] von Teilfolgen in $\pi_{Laplace}$ dafür sorgen, dass die Matrizen-Operanden abwechselnd von außen herein und von innen heraus durchlaufen werden und dadurch die Cache-Ausnutzung verbessern.

Weitere mögliche Überlegungen, die Verbesserungen erzielen können, wären zum einen, so wie bei dem Peano-Prinzip, LE rekursiv erst auf kleine Blöcke und dann auf größere Blöcke anzuwenden, oder zum anderen LE erst auf Matrix-Vektor-Multiplikation zu testen und dann für Matrix-Matrix-Multiplikation zu verallgemeinern.

[3] Die Plätze der Operationen in einer Teilfolge von $\pi_{Laplace}$ absteigend sortieren.

Literaturverzeichnis

[BN03] M. Belkin, P. Niyogi. Laplacian Eigenmaps for Dimensionality Reduction and Data Representation. *Neural Computation*, 15:1373–1396, 2003. (Zitiert auf den Seiten 7 und 9)

[BZ06] M. Bader, C. Zenger. Cache oblivious matrix multiplication using an element ordering based on a Peano curve. *Linear Algebra and Its Applications*, 417(2–3):301—-313, 2006. (Zitiert auf den Seiten 4, 7, 24 und 36)

[JOP+] E. Jones, T. Oliphant, P. Peterson, et al. NumPy: Open source scientific tools for Python, 2001–. URL http://numpy.scipy.org/. (Zitiert auf Seite 9)

[Rud11] K. Rudolph. The minted package: Highlighted source code in LaTeX. 2011. (Zitiert auf Seite 9)

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Ramzy Saleh)