Abschlussprojekt

Ida Hönigmann Fabian Dopf

January 21, 2022

Aufgabe 1: Aufwandsordnung numerischer Verfahren

Wir betrachten ein abstraktes numerisches Verfahren, das für $N \in \mathbb{N}$ Eingabedaten eine Laufzeit von $y_N \in \mathbb{R}_+$ hat. Man sagt, das Verfahren habe Aufwandsordnung $p > 0$, falls eine Konstante $C > 0$ existiert, sodass $y_N \leq CN^p$ für alle $N \in \mathbb{N}$.

Teilaufgabe 1a:

Die Aufwandsordnung lässt sich über die Folge $\{p_N\}_{N\in\mathbb{N}}$ mit

$$p_N = \frac{\log(y_{2N}) - \log(y_n)}{\log(2)} \text{ für } N \in \mathbb{N} \tag{1}$$

quantifizieren. Beachten Sie, dass die Bestimmung von p_N die Verfügbarkeit von zwei aufeinanderfolgenden Folgengliedern y_N und y_{2N} erfordert. Verwenden Sie den Ansatz $y_N = CN^p$ und leiten Sie die Formel in 1 her!

Beweis. Sei $(y_N)_{N\in\mathbb{N}}$ die Laufzeit eines numerischen Verfahrens mit

$$\exists C > 0 \exists p > 0 \forall N \in \mathbb{N} : y_N = CN^p.$$

Durch Umformen ergibt sich

$$\log(y_{2N}) - \log(y_N) = \log\left(\frac{y_{2N}}{y_N}\right) = \log\left(\frac{C(2N)^p}{C \cdot N^p}\right) = \log(2^p) = p\log(2)$$
$$\implies p = \frac{\log(y_{2N}) - \log(y_N)}{\log(2)}.$$

Wenn sich y_N asymptotisch wie CN^p verhält konvergiert p_N gegen p.

□

Teilaufgabe 1b:

Sei $\{\delta_N\}_{N\in\mathbb{N}} \subseteq \mathbb{R}$ eine Nullfolge, d.h. es gilt $\delta_N \to 0$ für $N \to \infty$. Weiters verhalte sich die Laufzeit wie $y_N = (C + \delta_N)N^p$ mit $C > 0$. Zeigen Sie, dass die Folge $\{p_N\}_{N\in\mathbb{N}}$ gegen p konvergiert, d.h. es gilt $p_N \to p$ für $N \to \infty$.

Beweis. Zuerst berechnen wir einen Grenzwert, den wir in späterer Folge verwenden werden.

$$\lim_{n\to\infty} \log\left(\frac{C+\delta_{2N}}{C+\delta_N}\right) = \log\left(\lim_{n\to\infty} \frac{C+\delta_{2N}}{C+\delta_N}\right) = \log\left(\frac{\lim\limits_{n\to\infty} C+\delta_{2N}}{\lim\limits_{n\to\infty} C+\delta_N}\right) = \log\left(\frac{C}{C}\right) = \log(1) = 0$$

Die Gleichungen stimmen, da lim stetig ist und da laut Voraussetzung δ_N und somit auch δ_{2N} als Teilfolge, gegen 0 konvergieren.

Wir berechnen $\lim\limits_{n\to\infty} p_n$ indem wir die Gleichung 1 verwenden. Durch Einsetzen der Voraussetzung

$$y_N = (C+\delta_N)N^p$$

und den Rechenregeln von Limiten und dem Logarithmus erhalten wir folgendes:

$$\begin{aligned}\implies p_N &= \frac{\log(y_{2N}) - \log(y_N)}{\log(2)} = \frac{\log((C+\delta_{2N})(2N)^p) - \log((C+\delta_N)N^p)}{\log(2)} = \frac{\log\left(\frac{(C+\delta_{2N})(2N)^p}{(C+\delta_N)N^p}\right)}{\log(2)} \\ &= \frac{\log\left(\frac{(C+\delta_{2N})2^p}{(C+\delta_N)}\right)}{\log(2)} = \frac{p\log(2) + \log\left(\frac{C+\delta_{2N}}{C+\delta_N}\right)}{\log(2)} = p + \frac{\log\left(\frac{C+\delta_{2N}}{C+\delta_N}\right)}{\log(2)} \xrightarrow{n\to\infty} p + 0 = p\end{aligned}$$

Zusammenfassend gilt nun $\lim\limits_{n\to\infty} p_n = p$, was zu zeigen war.

□

Teilaufgabe 1c:

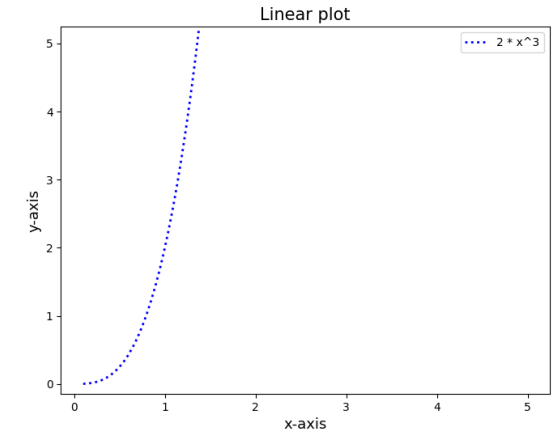
In sogenannter doppelt logarithmischer Darstellung (log-log Plots) wird für beide Koordinatenachsen eine logarithmische Skalierung verwendet, d.h. sowohl die waagrechte als auch die senkrechte Koordinatenachse wird logarithmisch unterteilt. Wie werden Potenzfunktionen der Form $y = cx^p$ in einem log-log Plot dargestellt? Wie können Sie die Ordnung p und die Konstante $c > 0$ aus einem log-log Plot von $y = cx^p$ direkt auslesen?

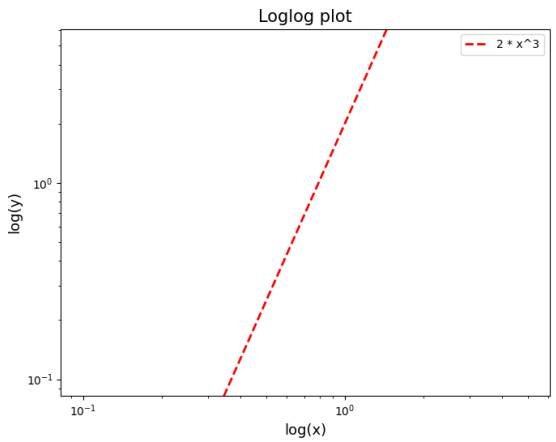
Ausarbeitung. Die Darstellung hat die Form einer Geraden, da die neuen Achsen $X = \log(x)$ und $Y = \log(y)$ sind und gilt

$$\begin{aligned} y &= c \cdot x^p \\ \iff \log(y) &= \log(c \cdot x^p) = \log(c) + p\log(x) \\ \iff Y &= \log(c) + p \cdot X \end{aligned} \tag{2}$$

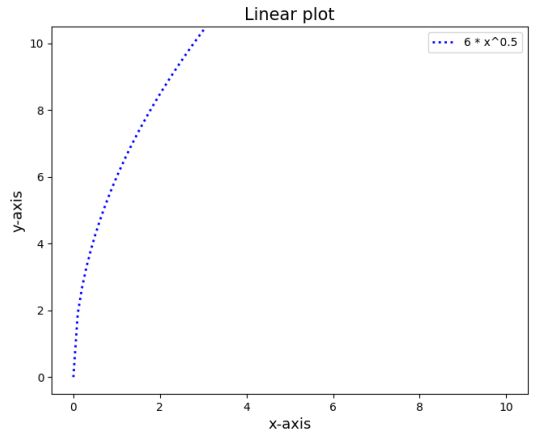
was der Gleichung einer Geraden entspricht.

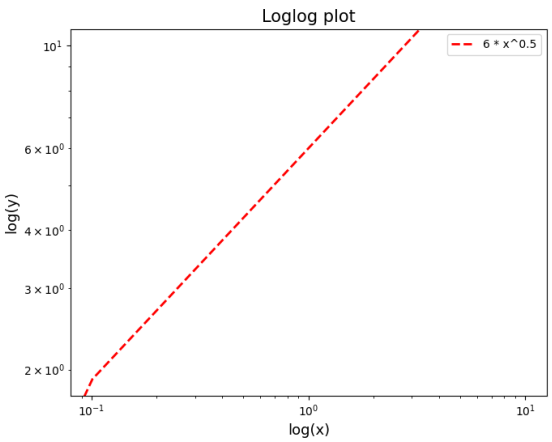
Um den Wert von c abzulesen, kann man den Funktionswert bei 1 ablesen, da $f(1) = c \cdot 1^p = c$. Um p abzulesen, kann man die Steigung der Geraden ansehen, wenn beide Achsen "gleich" skaliert, da p die Steigung der Geraden in Gleichung 2 ist.





(a) Vergleich der beiden Graphen der Gleichung $y = 2x^3$. Im Loglog plot ist zu erkennen, dass die Steigung der Geraden 3 bei gleichen Achsenskalierungen ist, also gilt $p = 3$.





(b) Vergleich der beiden Graphen der Gleichung $y = 6x^{0.5}$. Im Loglog plot ist zu erkennen, dass bei die Funktion bei $x = 1$ den Funktionswert 6 annimmt, also gilt $c = 6$.

Figure 1: Zwei unterschiedliche Funktionen der Form $y = cx^p$, jeweils in einem linear skalierten und einem Loglog Plot.

Aufgabe 2: Cholesky-Verfahren und Skyline-Matrizen

Eine Matrix $A \in \mathbb{R}^{n\times n}$ heißt Skyline-Matrix, falls es für $l = 1, ..., n$ Zahlen $p_l, q_l \in \mathbb{N}_0$ gibt, sodass für die i-te Zeile und j-te Spalte von A gilt:

- $A_{i,k} = 0$ für $k < i - p_i$,
- $A_{k,j} = 0$ für $k < j - q_j$,

Folgendes Beispiel illustriert diese Aussage:

$$A = \begin{pmatrix} 1 & & & & 1 \\ & 1 & & 2 & 2 \\ & & 1 & 3 & 3 \\ 1 & 2 & 3 & 14 & 18 \\ & 4 & 5 & 29 & 48 \end{pmatrix}.$$

Teilaufgabe 2a:

Beweisen Sie, dass das Cholesky-Verfahren genau dann wohldefiniert ist (d.h. es wird nicht durch Null dividiert oder die Wurzel aus einer negativen Zahl gezogen), wenn die Matrix $A \in \mathbb{R}^{n\times n}$ symmetrisch und positiv definit ist.

Beweis. Wir wiederholen zuerst die relevanten Definitionen.

- Eine Matrix $A \in \mathbb{R}^{n\times n}$ heißt symmetrisch, falls $A = A^T$.
- Eine Matrix $A \in \mathbb{R}^{n\times n}$ heißt positiv definit, falls $\forall u \in \mathbb{R}^n \setminus \{0\} : u^T Au > 0$.

Wir zeigen

$$\forall A \in \mathbb{K}^{n\times n} \text{ symmetrisch und positiv definit } \exists L \in \mathbb{R}^{n\times n} \text{ untere Dreiecksmatrix } : LL^T = A$$

durch vollständige Induktion nach n:

- **Induktionsanfang:** $n = 1$

  Wenn $A := \begin{pmatrix} a_{11} \end{pmatrix} \in \mathbb{R}^{1\times 1}$ eine beliebige symmetrische und positiv definite Matrix ist, folgt aus positiv definit, dass für

  $$u := \begin{pmatrix} 1 \end{pmatrix} \in \mathbb{R}^1 \setminus \{0\} \qquad 0 < u^T Au = \begin{pmatrix} 1 \end{pmatrix} \begin{pmatrix} a_{11} \end{pmatrix} \begin{pmatrix} 1 \end{pmatrix} = a_{11}$$

  Da also $a_{11} > 0$ gilt, ist $L := \begin{pmatrix} \sqrt{a_{11}} \end{pmatrix} \in \mathbb{R}^1$ wohldefiniert. Dann gilt

  $$LL^T = \begin{pmatrix} \sqrt{a_{11}} \end{pmatrix} \begin{pmatrix} \sqrt{a_{11}} \end{pmatrix} = \begin{pmatrix} a_{11} \end{pmatrix} = A$$

- **Induktionsvoraussetzung:** $\forall A \in \mathbb{R}^{n-1\times n-1}$ symmetrisch und positiv definit $\exists L \in \mathbb{R}^{n-1\times n-1}$ untere Dreiecksmatrix : $LL^T = A$.

- **Induktionsschritt:** $n-1 \implies n$

  Sei $A \in \mathbb{R}^{n\times n}$ eine symmetrische und positiv definite Matrix. Wir definieren eine Matrix $B \in \mathbb{R}^{(n-1)\times(n-1)}$, einen Vektor $a \in \mathbb{R}^{n-1}$ und eine Zahl $\alpha \in \mathbb{R}$ durch

  $$\forall i,j \in \{1,...,n-1\} : B_{i,j} := A_{i,j} \qquad \forall i \in \{1,...,n-1\} : a_i := A_{i,n} \qquad \alpha := A_{n,n}.$$

  Zusammengefasst gilt nun

  $$A = \begin{pmatrix} B & a \\ a^T & \alpha \end{pmatrix} \qquad \text{wobei sich das } a^T \text{ aus der Symmetrie von } A \text{ ergibt.}$$

  B ist eine symmetrische, positiv definite Matrix aus $\mathbb{R}^{(n-1)\times(n-1)}$, daher existiert laut Induktionsvoraussetzung eine untere Dreiecksmatrix $P \in \mathbb{R}^{(n-1)\times(n-1)}$ mit $PP^T = B$.

  Da B positiv definit ist und somit regulär ist, folgt die eindeutige Existenz eines Vektors $l \in \mathbb{R}^{n-1}$ der die Gleichung $Pl = a$ erfüllt.

  Wir wollen nun $\beta \in \mathbb{R}$ so definieren, dass $\beta = \sqrt{\alpha - l^T l}$. Dazu müssen wir sicherstellen, dass $\alpha - l^T l > 0$.

  Wenn wir die Definition von l verwenden und Umformen erhalten wir

  $$\begin{aligned} \alpha - l^T l &= \alpha - (P^{-1}a)^T(P^{-1}a) = \alpha - a^T(P^{-1})^T P^{-1} a \\ &= \alpha - a^T(PP^T)^{-1}a = \alpha - a^T B^{-1} a \end{aligned}$$

  Da A positiv definit ist ergibt sich

  $$0 < \begin{pmatrix} -B^{-1}a \\ 1 \end{pmatrix}^T \underbrace{\begin{pmatrix} B & a \\ a^T & \alpha \end{pmatrix}}_{=A} \begin{pmatrix} -B^{-1}a \\ 1 \end{pmatrix} = \alpha - a^T B^{-1} a$$

  Also ist $\beta := \sqrt{\alpha - l^T l}$ wohldefiniert. Umgeformt gilt nun $l^T l + \beta^2 = \alpha$.

  Definieren wir nun $L \in \mathbb{R}^{n\times n}$ durch

  $$L = \begin{pmatrix} P & 0 \\ l^T & \beta \end{pmatrix}$$

  Dann gilt

  $$LL^T = \begin{pmatrix} P & 0 \\ l^T & \beta \end{pmatrix} \begin{pmatrix} P^T & l \\ 0 & \beta \end{pmatrix} = \begin{pmatrix} PP^T & Pl \\ l^T P^T & l^T l + \beta^2 \end{pmatrix} = \begin{pmatrix} PP^T & Pl \\ (Pl)^T & l^T l + \beta^2 \end{pmatrix} = \begin{pmatrix} B & a \\ a^T & \alpha \end{pmatrix} = A$$

  □

Teilaufgabe 2b:

Beweisen Sie, dass die Besetzungsstruktur der Cholesky-Zerlegung der Skyline-Matrix A erhalten bleibt, d.h. dass auch die untere Dreiecksmatrix L eine geeignete Bandstruktur aufweist.

Beweis. TODO Beweis 2b □

Aufgabe 3: Pseudocode für Cholesky-Zerlegung von Skyline-Matrizen

Verwenden Sie den Cholesky-Algorithmus aus der Vorlesung. Entwerfen Sie jeweils einen Pseudocode, der für eine Skyline-Matrix:

Teilaufgabe 3a:

möglichst effizient die Struktur erkennt.

Ausarbeitung.

```
def to_skyline(matrix):
  values = list()

  for i in range(dim(matrix)):
    up_branch = matrix[:, i][:i + 1]  # i-th column from top to diagonal
    up_branch.reverse()

    # remove all entries outside of skyline-branch
    while not up_branch.empty and up_branch[-1] == 0:
      up_branch.pop(-1)

    left_branch = matrix[i, :][:i]    # i-th row from left to diagonal
    left_branch.reverse()

    # remove all entries outside of skyline-branch
    while not left_branch.empty and left_branch[-1] == 0:
      left_branch.pop(-1)

    values.append([up_branch, left_branch])

  return values

```

Listing 1: Strukturerkennung einer nicht notwendigerweise symmetrischen Skyline-Matrix

```
def to_spd_skyline(matrix):
  values = list()

  for i in range(dim(matrix)):
    branch = matrix[:, i][:i + 1] # i-th column (= row) from top to diagonal
    branch.reverse()

    # remove all entries outside of skyline-branch
    while not branch.empty and branch[-1] == 0:
      branch.pop(-1)

    values.append(branch)

  return values

```

Listing 2: Strukturerkennung einer symmetrischen positiv definiten Skyline-Matrix

Teilaufgabe 3b:

die Cholesky-Zerlegung berechnet.

Ausarbeitung.

```
def cholesky(matrix):
  n = dim(matrix)
  l = zero matrix of dimension n by n

  for k in range(n):
    s = 0
    for j in range(k):
      s += l[k, j] * l[k, j]

    l[k, k] = sqrt(matrix[k, k] - s)

    for i in range(k+1, n):
      s = 0
      for j in range(k):
        s += l[i, j] * l[k, j]

      l[i, k] = (matrix[i, k] - s) / l[k, k]

  return l

```

Listing 3: Algorithmus für die Cholesky Zerlegung einer Matrix aus der Vorlesung

```
def cholesky_skyline(values): # input should be generated by to_spd_skyline
  n = len(values)
  l = zero matrix of dimension n by n

  # calculate maximum branch length
  max_width = max(len(branch) for branch in values)

  for k in range(n):
    # calculate index of first entry not equal to zero
    start_idx = k - len(values[k]) + 1
    # realize sum as dot product of two vectors
    s = np.dot(l[k][start_idx:k], l[k][start_idx:k])
    l[k, k] = sqrt(self[k, k] - s)

    for i in range(k + 1, min(k + max_width, n)):
      if k > i - len(values[i]):  # values outside of the branch are zero
        # realize sum as dot product of two vectors
        s = np.dot(l[i][start_idx:k], l[k][start_idx:k])
        l[i, k] = (self[i, k] - s) / l[k, k]

  return l

```

Listing 4: Optimierter Algorithmus für die Cholesky Zerlegung einer Skyline-Matrix

Aufgabe 4: Aufwand des Algorithmus und Verhalten in Spezialfällen

Teilaufgabe 4a:

Sei $A \in \mathbb{R}^{n \times n}$ eine Skyline-Matrix. Welchen Aufwand haben Ihre Algorithmen aus Aufgabe 3 in Abhängigkeit von der Größe n der Eingabedaten und Skyline-Indices $p_l = q_l$?

Ausarbeitung.

- `to_skyline` hat wegen den Schleifen in Zeile 4, 9 und 16 Aufwand

$$\sum_{l=1}^{n}(n - p_l) + (n - q_l) = 2n^2 - \sum_{l=1}^{n} p_l + q_l.$$

  - best case: Im besten Fall ist $p_l = q_l = l$ also maximal, mit Aufwand

$$2n^2 - \sum_{l=1}^{n} 2l = 2n^2 - 2\frac{n(n+1)}{2} = 2n^2 - (n^2 + n) = n^2 - n$$

  - worst case: Im schlechtesten Fall ist $p_l = q_l = 0$ also minimal, mit Aufwand

$$2n^2 - \sum_{l=1}^{n} 0 = 2n^2$$

- `to_spd_skyline` hat wegen den Schleifen in Zeile 4 und 9 Aufwand

$$\sum_{l=1}^{n}(n - p_l) = n^2 - \sum_{l=1}^{n} p_l$$

  - best case: Im besten Fall ist $p_l = l$ also maximal, mit Aufwand

$$n^2 - \sum_{l=1}^{n} l = n^2 - \frac{n(n+1)}{2} = n^2 - \frac{1}{2}n^2 - \frac{1}{2}n = \frac{1}{2}(n^2 - n)$$

  - worst case: Im schlechtesten Fall ist $p_l = 1$ also minimal, mit Aufwand

$$n^2 - \sum_{l=1}^{n} 1 = n^2 - n$$

- `cholesky` hat wegen den Schleifen in Zeile 5, 7, 12 und 14 Aufwand

$$\begin{aligned}\sum_{k=1}^{n}(k + \sum_{i=k+1}^{n} k) &= \sum_{k=1}^{n}(k + k(n-k)) = \sum_{k=1}^{n} k + \sum_{k=1}^{n} kn - \sum_{k=1}^{n} k^2 \\ &= \frac{n(n+1)}{2} + \frac{n^2(n+1)}{2} - \frac{n(n+1)(2n+1)}{6} = \frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{1}{3}n\end{aligned}$$

  also $\mathcal{O}(n^3)$.

- `cholesky_skyline` Zeilen 12 und 18 haben Aufwand $\mathcal{O}(p_k)$, da die beiden Vektoren Länge

$$k - start\_idx + 1 = k - (k - p_k + 1) + 1 = p_k$$

haben und sind somit, neben Zeile 6 mit Aufwand von $\mathcal{O}(n)$, die aufwändigsten Zeilen in der Funktion.

Durch das if-Statement in Zeile 16 wird sichergestellt, dass der Aufwand nur anfällt, falls nach Aufgabe 2b das Ergebnis nicht trivial ist.

Die Einschränkung in Zeile 15 auf den Bereich `range(k+1, k+max_width)` statt `range(k+1, n)` ermöglicht es viele der if-Statements zu ersparen, die es dem Python-Interpreter erschweren die Ausführung zu optimieren[1].

Insgesamt ist der Aufwand also

$$n + \sum_{k=1}^{n} \sum_{l=1}^{p_k} p_k = n + \sum_{k=1}^{n} p_k^2$$

was

- im besten Fall $n + \sum_{k=1}^{n} 1 = 2n = \mathcal{O}(n)$,
- im Fall, dass $p_l \approx \sqrt{n}$ ungefähr $n + \sum_{k=1}^{n} (\sqrt{n})^2 = n + n^2 = \mathcal{O}(n^2)$ und
- im schlechtesten Fall $n + \sum_{k=1}^{n} k^2 = n + \frac{n(n+1)(2n+1)}{6} = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{7}{6}n = \mathcal{O}(n^3)$ ist.

Teilaufgabe 4b:

Betrachten Sie Matrizen mit den Besetzungsstrukturen

$$\begin{pmatrix} * & * & * & * & * \\ * & * & & & \\ * & & * & & \\ * & & & * & \\ * & & & & * \end{pmatrix} \text{ bzw. } \begin{pmatrix} * & & & & * \\ & * & & & * \\ & & * & & * \\ & & & * & * \\ * & * & * & * & * \end{pmatrix}$$

Welche Besetzungsstruktur hat die Cholesky-Zerlegung für beide Matrizen? Was könnte man machen, um für Matrizen mit der "linken" Besetzungsstruktur die Cholesky-Zerlegung effizienter zu berechnen?

Ausarbeitung.

- Die linke Matrix ist vollbesetzt als Skyline-Matrix, in dem Sinne, dass

  $$\forall k \in \{1, ..., n\} : p_k = q_k = k$$

  also immer den maximal möglichen Wert annimmt.

- Die rechte Matrix hat die Skyline-Indizes

  $$\forall k \in \{1, ..., n-1\} : p_k = q_k = 1 \qquad \text{und} \qquad p_n = q_n = n$$

  und ist daher nach Aufgabe 4a effizienter in der Berechnung der Cholesky-Zerlegung.

Eine effiziente Berechnung der Cholesky-Zerlegung der linken Matrix erhält man mit folgender Überlegung:

Definieren wir eine Abbildung die einer Matrix die "gespiegelte" Matrix zuordnet:

$$\sigma : \mathbb{R}^{n \times n} \to \mathbb{R}^{n \times n} \qquad (\sigma(A))_{i,j} := A_{n+1-j, n+1-i}$$

σ hat folgende Eigenschaften: Für beliebige $A, B \in \mathbb{R}^{n \times n}$ gilt

[1] If-Statements ermöglichen es nicht die Instruction Pipeline vollständig auszunützen und verlangsamen dadurch die Ausführung. In modernen Systemen wird daher sogenannte Branch Prediction durchgeführt die versucht vorherzusagen ob die Bedingung wahr ist.

- Die Umkehrabbildung $\sigma^{-1} = \sigma$, da

$$\sigma(\sigma(A))_{ij} = \sigma(A)_{n+1-j,n+1-i} = A_{n+1-(n+1-i),n+1-(n+1-j)} = A_{i,j}$$

- Verträglich mit Transponieren

$$\sigma(A^T)_{i,j} = A^T_{n+1-j,n+1-i} = A_{n+1-i,n+1-j} = \sigma(A)_{j,i} = \sigma(A)^T_{i,j}$$

- Verträglich mit Multiplizieren

$$\begin{aligned}(\sigma(A)\sigma(B))_{i,j} &= \sum_{k=1}^{n} \sigma(A)_{i,k}\sigma(B)_{k,j} = \sum_{k=1}^{n} A_{n+1-k,n+1-i}B_{n+1-j,n+1-k} \\ &= \sum_{k=1}^{n} B_{n+1-j,k}A_{k,n+1-i} = (BA)_{n+1-j,n+1-i} = \sigma(BA)_{i,j}\end{aligned}$$

  da $k \mapsto n+1-k$ eine Permutation von $\{1,..,n\}$ ist.

- Dreiecksmatrix bleibt erhalten

  Wenn A eine untere Dreiecksmatrix ist, also $A_{i,j} = 0$ für $i < j$, so folgt, dass $\sigma(A)$ auch eine untere Dreiecksmatrix ist, da

$$i < j \iff n+1-j < n+1-i.$$

- Symmetrie bleibt erhalten

  Wenn A symmetrisch ist, so ist auch $\sigma(A)$ symmetrisch, da

$$\sigma(A)_{i,j} = A_{n+1-j,n+1-i} = A_{n+1-i,n+1-j} = \sigma(A)_{j,i}.$$

Sei A eine Matrix von der linken, ungünstigen Art. Dann ist $\sigma(A)$ von der rechten Art, und ist es ist daher möglich die Cholesky-Zerlegung von $\sigma(A)$ effizient zu berechnen. Sei L eben diese Zerlegung von $\sigma(A)$, d.h. $LL^T = \sigma(A)$.

Nach dem oben gezeigten gilt nun

$$\sigma(L)^T\sigma(L) = \sigma(L^T)\sigma(L) = \sigma(LL^T) = \sigma(\sigma(A)) = A,$$

womit wir eine Cholesky-Zerlegung von A erhalten.

Aufgabe 5: Implementierung des Algorithmus und empirische Aufwandsschätzung

Implementieren Sie Ihren modifizierten Cholesky-Algorithmus in Python und weisen Sie empirisch nach, dass der Aufwand linear in n wächst. Vergleichen Sie die Performance Ihrer Implementierung mit der Python-Funktion `scipy.linalg.cholesky`, wobei die Skyline-Matrix A als vollbesetzte Matrix gespeichert ist.

Ausarbeitung. Die gesamte Python Implementierung kann auf `https://github.com/idahoenigmann/numerical_analysis_final_project` gefunden werden. Die Implementierung von `to_spd_skyline` und `cholesky_skyline` ist außerdem im Anhang zu finden.

Der Aufwand von `cholesky_skyline` hängt, wie in Aufgabe 3a begründet, stark von den Werten p_l ab. Grafik 2 zeigt die Zeitdauer der Cholesky Zerlegung für unterschiedliche durchschnittliche p_l Werte.

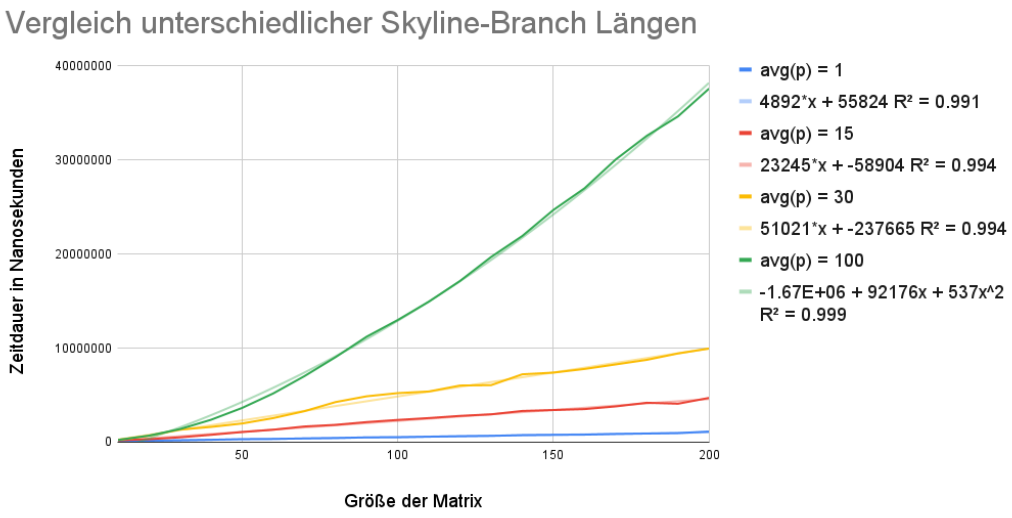


Figure 2: Für durchschnittliche p_l Werte bis 30 verhält sich der Aufwand fast linear. Bei großen durchschnittlichen p_l Werten, wie etwa 100 ist der Aufwand bereits polynomial.

Folgende Grafik vergleicht die Laufzeiten der unterschiedlichen Implementierungen aus Aufgabe 3b.

Da große Teile von numpy in C geschrieben sind, ist es nicht verwunderlich, dass `numpy.linalg.cholesky` deutlich schneller läuft als `choleksy_skyline`.

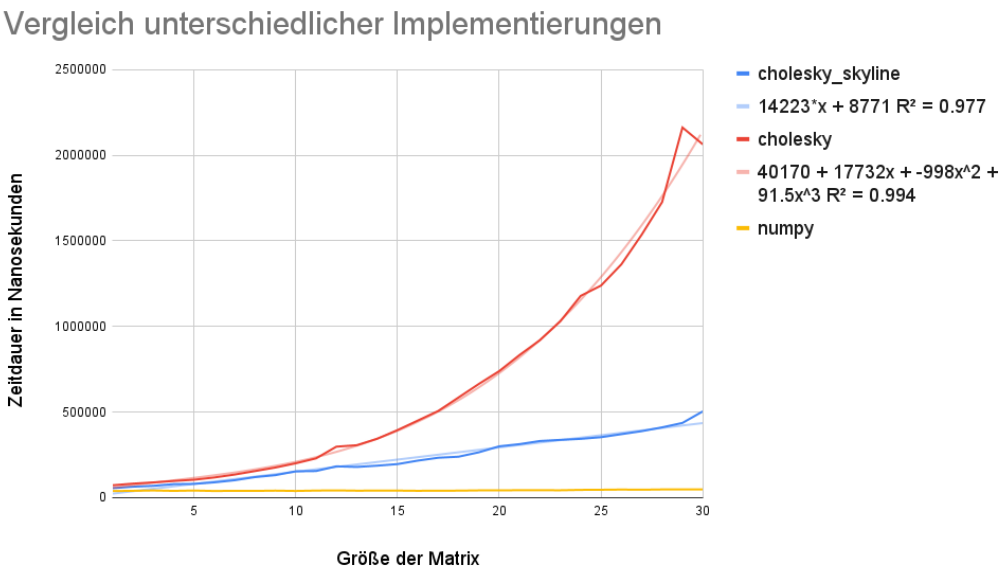


Figure 3: Vergleich des Aufwand der unterschiedlichen Implementierungen für Matrizen bis Größe 30×30 und durchschnittlichem p_l Wert von 10. Der Aufwand von `cholesky_skyline` ist deutlich besser, als jener von `cholesky`, allerdings ist `numpy.linalg.cholesky`, mit fast konstantem Aufwand, die schnellste Variante.

A Python Implementierung

```python
import numpy as np
import matrix_utils


class SPDSkylineMatrix:
  values = list()

  def __init__(self, matrix):
    if not matrix_utils.is_symmetrical(matrix):
      raise Exception("matrix must be symmetric")
    if not matrix_utils.is_positive_definite(matrix):
      raise Exception("matrix must be positive definite")

    for i in range(len(matrix)):
      branch = matrix[:, i][:i + 1].T.tolist()
      branch.reverse()
      while len(branch) > 0 and branch[-1] == 0:
        branch.pop(-1)

      self.values.append(branch)

  def __getitem__(self, item):
    row_idx, col_idx = item
    row_idx, col_idx = min(row_idx, col_idx), max(row_idx, col_idx)

    if row_idx < 0 or col_idx > len(self.values):
      raise Exception("index out of bounds")

    col = self.values[col_idx]

```

```
      if col_idx - row_idx < len(col):
        return col[col_idx - row_idx]
      else:
        return 0

  def to_matrix(self):
    matrix = np.zeros((len(self.values), len(self.values)))

    for i in range(len(self.values)):
      for j in range(len(self.values[i])):
        matrix[i - j, i] = self.values[i][j]
        matrix[i, i - j] = self.values[i][j]

    return matrix

  def cholesky(self):
    l = np.zeros((len(self.values), len(self.values)))

    max_width = np.max(list(len(lst) for lst in self.values))

    for k in range(len(self.values)):
      start_idx = k - len(self.values[k]) + 1
      s = np.dot(l[k][start_idx:k], l[k][start_idx:k])
      l[k, k] = np.sqrt(self[k, k] - s)

      for i in range(k + 1, min(k + max_width, len(self.values))):
        if k > i - len(self.values[i]):
          s = np.dot(l[i][start_idx:k], l[k][start_idx:k])
          l[i, k] = (self[i, k] - s) / l[k, k]

    return l
```

Listing 5: Implementierung der Umwandlung in Skyline-Form sowie der Cholesky Zerlegung für die Skyline-Form