Hochschule für Telekommunikation

DKMI16

Programmieren eines Kompressions Plug-In für Image-J

Author:
Colin Simon (S166159), Cecilia Launstein (S166129), Marcel Liebscher (S166213), Theresa Urbach (S166167) und Sarah Ebert (S166111)

5. Juni 2018

Inhaltsverzeichnis

1 Einleitung

In dieser Ausarbeitung wird ein neues Kompressions Plug-In für Image-J vorgestellt. Zuerst wird auf die Ziele, Anforderungen und die Vorgehensweise eingegangen. Danach werden die theoretischen Grundlagen erläutert. Im Anschluss folgt die Beschreibung des Plug-Ins. Die Funktionsweise des Plug-Ins, in der die Umsetzungen der verwendeten Kompressionscodierungen vorgestellt wird, wird daraufhin beschrieben. Danach erfolgt die Durchführung einer Testreihe, um das Plug-In zu prüfen. Mit diesen Erkenntnissen kann das Plug-In bewertet und mögliche Erweiterungen geschlussfolgert werden. Im letzten Abschnitt ist eine Zusammenfassung der gesammelten Erkenntnisse. Im Anhang befinden sich sowohl die Testbilder als auch der Quellcode für Kompression und Dekompression.

1.1 Ziel

Das Ziel ist ein funktionsfähiges Kompressions Plug-In zu erstellen. Dabei ist es wichtig zu verstehen, wie das Plug-In genau funktioniert. Die Erkenntnisse werden in dieser Ausarbeitung festgehalten. Weiterhin ist es wichtig, dass das Plug-In unabhängig von PC und Betriebssystem läuft.

1.2 Anforderung

Die Anforderung an das Plug-In beinhaltet zum einen eine verlustfreie und fehlerfreie Kompression, sodass die Qualität der Bildes erhalten bleibt. Auf die Qualität wird im nächsten Abschnitt eingegangen. Zum Anderen kann das Plug-In Bilder in verschiedenen Formaten verarbeiten. Dabei liegt der Schwerpunkt auf den Formaten, die auch von Image-J unterstützt werden. Ebenso sollen sowohl schwarz-weiß als auch farbige Bilder komprimiert werden können. Auf der Basis von Lauflängen- und Entropiecodierung kann das Plug-In synthetische Bilddaten komprimieren. Eine weitere Anforderung ist, dass die Dateigröße des komprimierten Bildes minimal sein soll. Die Umsetzung soll über die Oberfläche des Bildbearbeitungsprogrammes Image-J erfolgen. Das Plug-In soll zudem auf mehreren Betriebssystemen kompatibel sein.

1.3 Vorgehensweise

Um die Anforderungen zu erfüllen, wird für das Plug-In die nach unserer Einschätzung effizienteste Komprimierungsart verwendet. In einem Ablaufplan wird der Aufbau des Plug-Ins festgehalten. Anhand diesen erfolgt die Erstellung des Quellcodes. Um die korrekte Funktionsweise festzustellen, werden Tests mit verschiedenen Bildern, sowohl in Graustufen als auch in Farbe, durchgeführt. Die Ergebnisse werden dokumentiert. Anschließend wird die Rekonstruktion der Bilder auf Fehlerfreiheit überprüft. Dafür ist ein Entwurf eines entsprechenden Decoders, welcher das komprimierte Bild wieder dekomprimiert, notwendig.

2 Theoretische Grundlagen

In diesem Abschnitt werden die theoretischen Grundlagen erklärt. Dabei wird auf folgende Themen eingegangen. Das erste Thema beinhaltet, was man unter dem Begriff der Qualität versteht. Wichtig ist Performance und verlustfreie Kompression zu erläutern. Im Anschluss werden die Grundlagen für Lauflängencodierung und Entropiecodierung veranschaulicht, welche die Basis für den Code schaffen. Dazu gehört auch die Gegenüberstellung von Vor- und Nachteilen der jeweiligen Codierung. Das letzte Thema geht um das Programm Image-J. In diesem Programm werden später die Plug-Ins für Komprimierung und Dekomprimierung ausgeführt.

2.1 Qualität

Die Qualität einer Kompression kann an mehreren Faktoren gemessen werden. Sowohl die Performance als auch die Frage nach der verlustfreien Komprimierung stehen dabei im Vordergrund. Weitere Kriterien für die Beschreibung der Qualität sind der Wert des Kompressionsverhältnisses und der Wert des Signal-Rausch-Verhältnisses (engl.: Peak Signal-to-Noise ratio (PSNR)) [Str09c].
Das Kompressionsverhältnis lässt sich mit folgender Formel aus [Str09c] berechnen:

$$C_r = \frac{\text{Datenmenge (Originalsignal)}}{\text{Datenmenge (codiertes Signal)}} \tag{1}$$

Je größer der Wert, desto größer ist die Komprimierung. Das Signal-Rausch-Verhältnis kann mit folgender Formel aus [Str09c] berechnet werden:

$$PSNR = 10 \cdot \log_{10} \cdot \left(\frac{x_{pp}^2}{MSE} \right) \text{ in [dB]} \tag{2}$$

Dieser Wert liegt bei verlustbehafteter Bild- und Videokomprimierung zwischen 30 und 50 dB, vorausgesetzt die Bittiefe beträgt 8 Bit. Je höher dabei das Signal-Rausch-Verhältnis ist, desto besser ist das Ergebnis der Komprimierungen beziehungsweise Dekomprimierungen [Wik18b].

2.1.1 Performance

Die Performance der Kompression ergibt sich durch die möglichst minimale Dateigröße des komprimierten Bildes. Auch eine Angabe der Bitrate R in [Bits/Symbol] ist möglich. Die Formel dafür lautet aus [Str09c] :

$$R = \frac{N_B}{N_A} = \frac{\text{Anzahl der Abtastwerte im Signal}}{\text{eingesetzte Datenmenge (Anzahl der Bits)}} \tag{3}$$

2.1.2 Verlustfreie Kompression

Bei verlustfreier Kompression gehen keine Informationen verloren. Die Daten werden nur anders dargestellt beziehungsweise zusammengefasst. Dies geschieht durch das Erkennen und Eliminieren von Redundanz. Als redundant werden Dateninformationen bezeichnet, die nicht für eine Repräsentation der Information gebraucht werden [Str09d]. Bekannte Verfahren sind dafür die Lauflängencodierung, LZW oder die Huffmancodierung [Wik18a].

2.2 Lauflängencodierung

Bei der Lauflängencodierung (engl.: run-length encoding) handelt es sich um eine Datenkomprimierung. Dabei besteht das Ziel in erster Linie darin, Platz zu reduzieren und erst sekundär steht die verbrauchte Zeit im Vordergrund. Die meisten Dateien besitzen einen hohen Grad an Redundanz. Diese Tatsache von einem geringen Informationsgehalt wird bei der Komprimierung ausgenutzt. Rasterdateien wie Bilder komprimieren besser, wenn sie über viele homogene Flächen verfügen. Bei digitalen Dateien akustischer und anderer analoger Signale sind sich wiederholende Muster die Voraussetzung. Die Einsparung bei Dateien, die aus zufälligen Bits bestehen, ist dagegen sehr gering. Einige Dateien werden sogar verlängert, was keine positive Komprimierung mit sich bringt. Datenkomprimierungen, wie die Lauflängencodierung, gewinnen trotz höherer Speichermöglichkeiten an Relevanz. Dies liegt daran, dass immer mehr Daten benötigt und gesammelt werden. Dabei ist es egal, ob es sich um Videospiele, Sicherheitsinformationen oder qualitativ höherwertige Bilder handelt. Des Weiteren ist die Lauflängencodierung wichtig für schnelle Zugriffsmedien, da diese meist wenig Speicherplatz nach sich ziehen [SW11].
Das Ziel einer Lauflängencodierung ist also die Codierung von Symbolsequenzen. Als Voraussetzung für eine positive Kompression muss es sich um Signale mit homogenen Abschnitten handeln. Dies wird erreicht, indem benachbarte, identische

Symbole, oder auch runs"genannt, zu neuen (Daten-)Symbolen verbunden werden. Hierbei gibt es mehrere Varianten. Die im Plug-In benutzte Variante wird im folgenden am Beispiel der Symbolsequenz "bcaddddddcbddddddddabddddddcddddddd"dargestellt [Lip18].

$$(r; s_i) \tag{4}$$

Hierbei steht r für die Anzahl identischer Symbole s_i. Die Zeichenfolge wird ersetzt durch einmalige Angabe der Anzahl der Wiederholungen und einmalige Angabe des Zeichens. Die Beispielsymbolsequenz wird zu "1b1c1a7d1c1b8d1a1b5d1c6d"komprimiert. Mit der Formel für das Kompressionsverhältnis C_r (Formel 1) kann nun festgestellt werden, inwiefern die Datei reduziert wurde.

$$C_r = \frac{\text{Originale Symbolsequenz}}{\text{Symbolsequenz nach Lauflängencodierung}} = \frac{34}{24} = 1,41\overline{666} \tag{5}$$

In diesem Beispiel beträgt das Kompressionsverhältnis ca. 1,42 bei einem Rohformat von der Länge 34. Da das Kompressionsverhältnis größer als 1 ist, folgt, dass die Kompression in diesem Beispiel erfolgreich ist.

Vorteile:

- beseitigt Redundanzen
- einfach und schnell, da die Werte solange nur eingelesen werden, bis diese sich ändern
- leichte Implementierung

Nachteile:

- ungeeignet für detailreiche Farbbilder
- ungeeignet für Signale mit wenigen homogenen Anteilen

2.3 Entropiecodierung

Allgemein ist das Ziel der Entropiecodierung, die Codierungsredundanz zu reduzieren [Str09a].

2.3.1 Huffmancodierung

Die Huffmancodierung funktioniert nach dem Prinzip eines Codebaumes. Der Algorithmus geht dabei auf die Häufigkeit der zu codierenden Zeichen ein. Somit werden Zeichen, die häufig verwendet werden, mit einem kurzen Bitcode, also wenigen Binärstellen dargestellt und die Zeichen, die selten verwendet werden mit einer längeren Binärreihe [Str09b].

Vorteile:

- häufig verwendete Zeichen mit kurzer Binärreihe dargestellt
- relativ einfach zu implementieren
- schnell zu implementieren
- reduziert die Daten auf ein Minimum

Nachteile:

- Codierungsart nicht eindeutig

2.3.2 Arithmetische Codierung

Die Informationen werden durch Berechnungen in einer sehr langen Zahl gespeichert. Auch hier wird die Auftretenswahrscheinlichkeit der Symbole beachtet, indem sich die Größe der Subintervalle unterscheidet. Die arithmetische Codierung beseitigt einen entscheidenden Nachteil der Huffmancodierung: Redundanzen durch die Umwandlung jedes Symbols. Die arithmetische Codierung hingegen beruht auf der Abbildung eines Wortes auf ein Intervall. Es wird also einem Signal nur ein Codewort zugeordnet. Dadurch erzeugt nicht jedes Symbol eine bitstellenmäßige Redundanz, sondern nur die komplette Nachricht [BCK01].

Vorteile:

- bessere Annäherung an die Signalentropie als bei Huffmancodierung
- effizienter und bessere Komprimierung als bei Huffmancodierung, besonders wenn das Alphabet kleiner und damit die Wahrscheinlichkeiten größer sind
- einfache Anpassung durch einfach anzupassende Intervallgrenzen
- nahezu redundanzfrei

Nachteile:

- etwas langsamer als Huffmancodierung

- schwierige Implementation
- seltene kommerzielle Nutzung, da viele Patente

2.3.3 Shannon-Fano-Codierung

Die Shannon-Fano-Codierung erzeugt einen Präfix-Code. Diese ist der erste Algorithmus, der Codes mit variabler Länge konstruiert. Sie basiert auf statistischen Annahmen der Auftretenswahrscheinlichkeiten, mit denen ein Baum erzeugt wird. Dabei ist das Ziel, häufig vorkommende Symbole mit kurzen Bitfolgen und weniger häufige durch lange Bitfolgen zu repräsentieren. Dazu müssen Häufigkeitstabellen erstellt und sortiert werden, welche gespeichert und ausgelesen werden müssen [Don02].

Vorteile:

- einfach zu implementieren
- kleiner algorithmischer Aufwand
- wenig benötigte Rechenleistung

Nachteile:

- ineffizienter als Huffmancodierung
- Kompressionsergebnisse nicht optimal

2.4 Image-J

Image-J ist ein Open Source Bildbearbeitungs- und Bildverarbeitungsprogramm, welches von Wayne Rasband entwickelt wurde. Verwendet wird es primär für medizinische und wissenschaftliche Bildanalyse. Es ist mit Java programmiert und damit plattformübergreifend. Es läuft auf 32-Bit und 64-Bit Modes [Ima17b].

3 Beschreibung des Plug-Ins

In diesem Abschnitt wird das Plug-In beschrieben. Der Quellcode ist in der Programmiersprache JAVA geschrieben. Das Plug-In beinhaltet eine Lauflängencodierung und eine Entropiecodierung. Die Art der Entropiecodierung ist eine Huffmancodierung. Das Plug-In besteht insgesamt aus folgenden zehn Methoden:

- setup
- run
- setPixels
- setRGB Pixels
- greyscalecheck
- toByte
- run_length_coding
- hufmann_encoding
- finalizebytearray
- showabout

3.1 Anwendung

Um das Plug-In zu benutzen, muss zuerst die kostenlose Software Image-J [Ima17a] installiert werden. Nach der Installation müssen die Plug-Ins für Komprimierung und Dekomprimierung (siehe Abbildung 2) im Ordner „plugins“ abgelegt werden. Dazu muss in den Ordner „ImageJ“ (siehe Abbildung 1) navigiert werden und anschließend in den Unterordner „plugins“.

Dort müssen dann die Jar-Files der Plug-Ins abgelegt werden (siehe Abbildung 2).

In Abbildung 3 ist die Oberfläche von Image-J zu sehen.

Als Erstes wird unter „File“ der Menüpunkt „Open“ ausgewählt, wie in Abbildung 4 zu sehen. Mit der Tastenkombination „STRG + O“ ist dies ebenfalls zu erreichen. Danach erfolgt die Auswahl der Bilddatei, die geöffnet werden soll. In Abbildung 5 ist das Bild „DSC_0119.NEF“, welches komprimiert werden soll, auszuwählen.

In einem neuem Fenster öffnet sich das ausgewählte Bild.

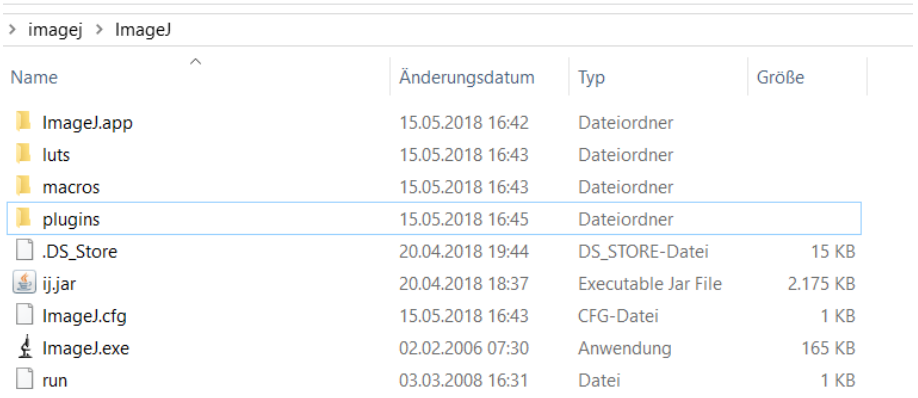

Abbildung 1: Ordner ImageJ

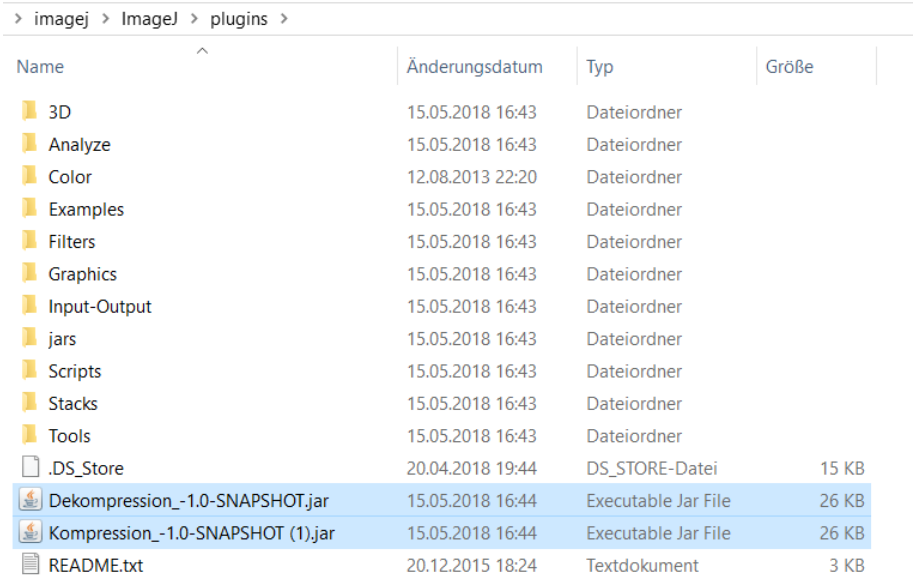

Abbildung 2: Die abgelegten Jar Files im Ordner plugins

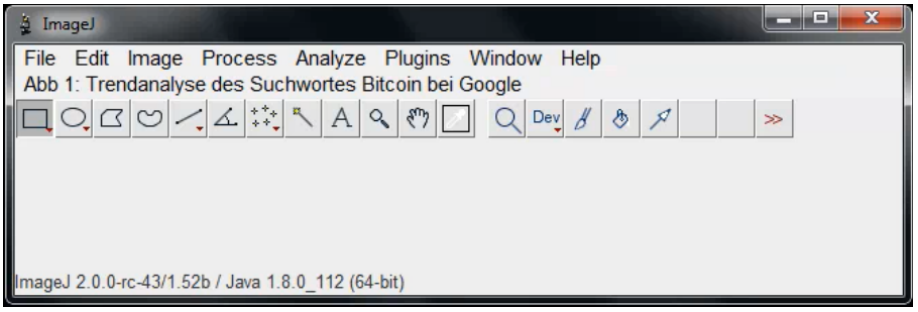

Abbildung 3: Oberfläche von Image-J

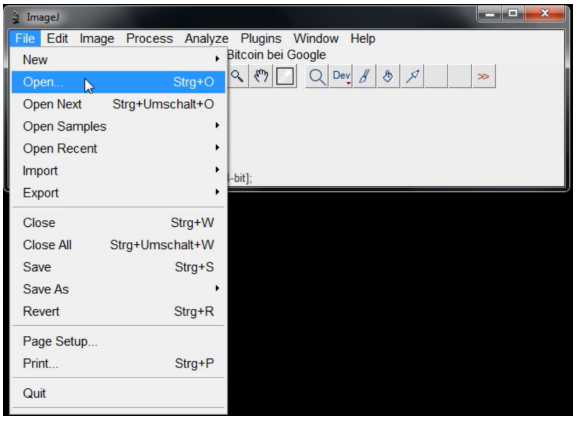

Abbildung 4: Öffnen eines Bildes

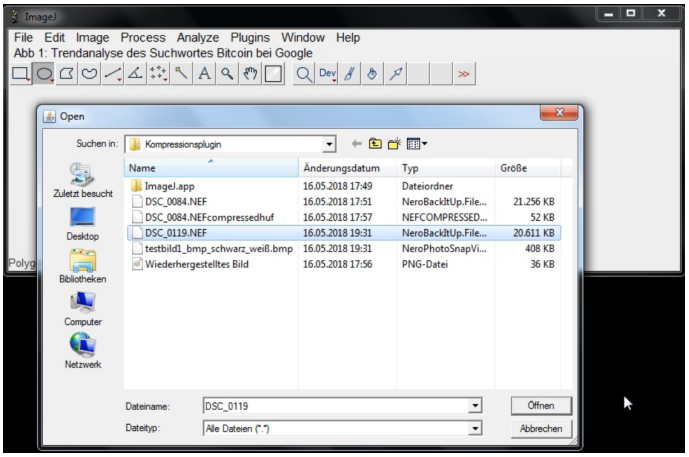

Abbildung 5: Auswahl der Bilddatei

Der nächste Schritt besteht darin, das Kompressions Plug-In zu starten. Dafür gibt es unter dem Reiter „Plugins“ die Option „RLE Kompression“, welches zuvor ein-

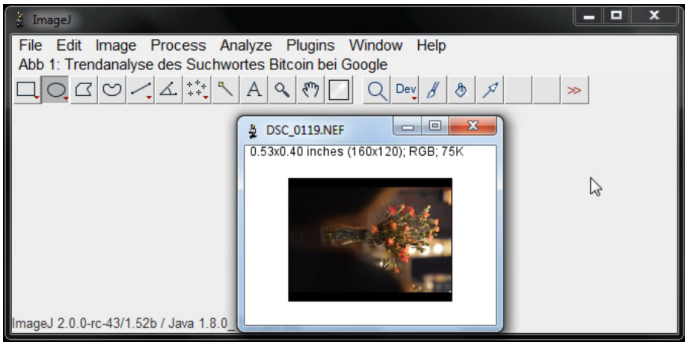

Abbildung 6: Originalbild

gebunden wurde. In Abbildung 7 ist dieser Schritt dargestellt.

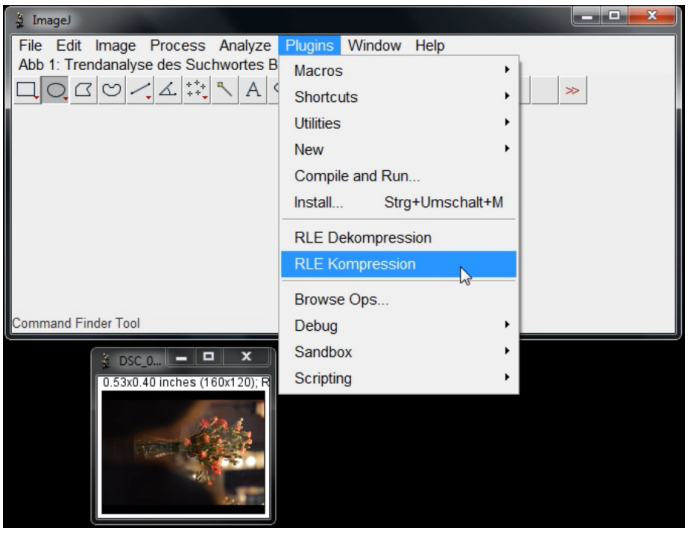

Abbildung 7: Starten der Kompression

Sobald die Kompression erfolgreich durchgeführt wurde, erscheint ein neues Meldungsfenster. In diesem steht, dass die komprimierte Datei im Ordner der Originaldatei erzeugt wurde (Abbildung 8).

In Abbildung 9 ist die neue Datei zu sehen. Die Originaldatei ist 20611 KB groß. Die komprimierte Bilddatei ist auf 47 KB reduziert worden. Auffällig ist, dass die komprimierte Datei deutlich kleiner ist.

Um dasselbe Bild wieder herzustellen, muss die Dekompression gestartet werden. Wie in Abbildung 10 zu sehen, wird die Dekompression unter dem Reiter „Plugins“ ausgelöst.

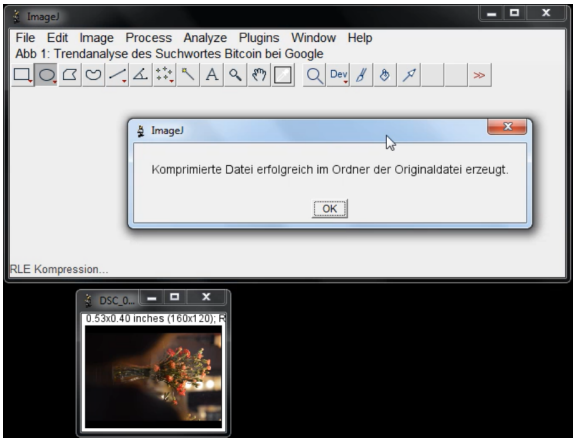

Abbildung 8: Erfolgreiche Kompression

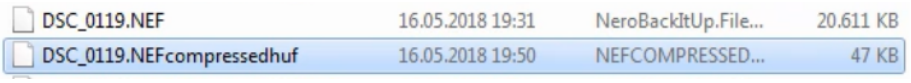

Abbildung 9: Vergleich der Dateigrößen

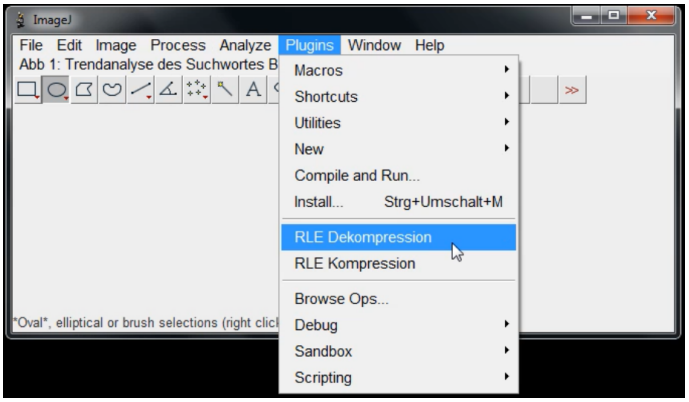

Abbildung 10: Starten der Dekompression

In Abbildung 11 ist zu erkennen, dass anschließend eine Aufforderung, eine Bilddatei auszuwählen, welche dekomprimiert werden soll, erscheint. Dabei ist zu beachten, dass es sich um eine komprimierte Datei handelt. In diesem Beispiel heißt die zuvor komprimierte Datei „DSC_0119.NEFcompressedhuf“.

Nachdem die Dekomprimierung durchgeführt wurde, öffnet sich in einem neuen Fenster das wiederhergestellte Bild (Abbildung 12).

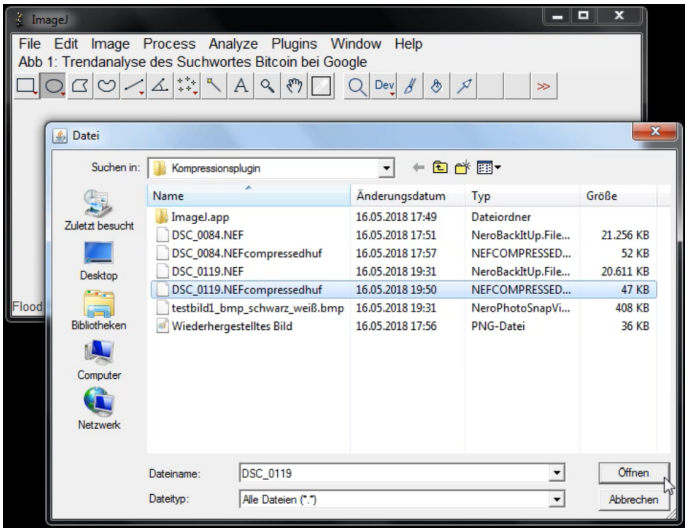

Abbildung 11: Auswahl der komprimierten Datei

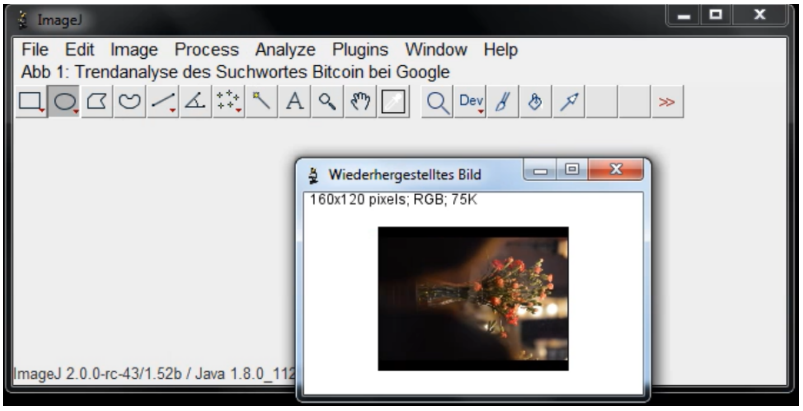

Abbildung 12: Wiederhergestelltes Bild

4 Funktionsweise des Plug-Ins

Dieser Abschnitt befasst sich mit einem Programmablaufplan und der Erläuterung notwendiger Methoden für das Plug-In.

4.1 Ablauf der Methoden

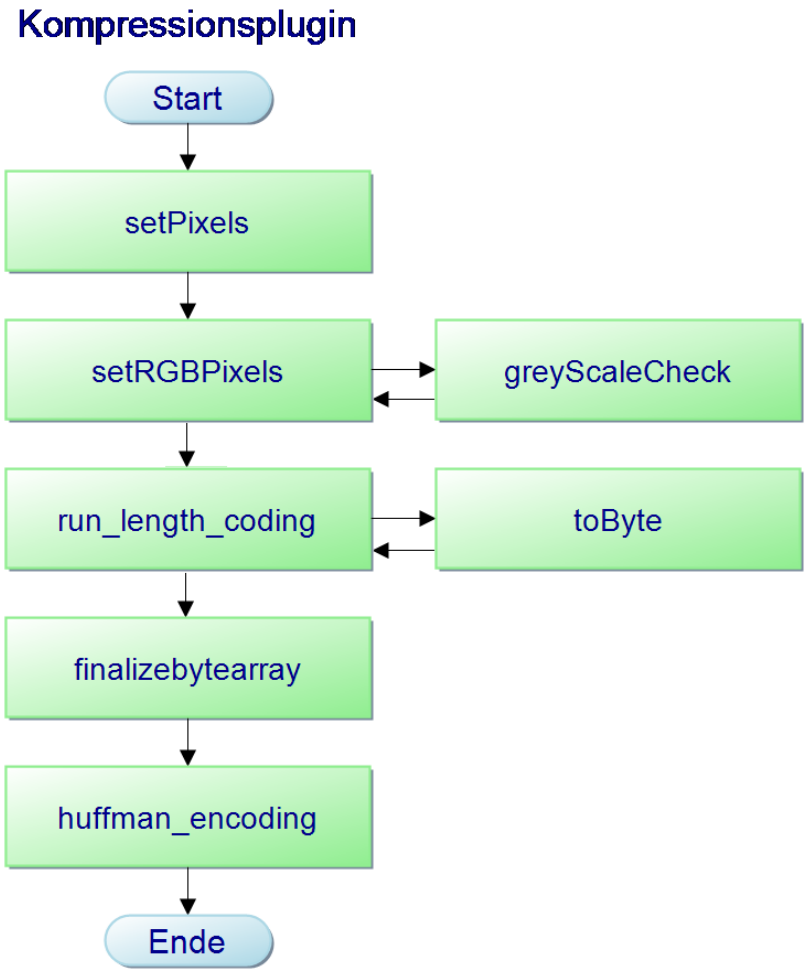

Abbildung 13: Ablauf des Plug-Ins

4.2 Einbindung in Image-J

Das Plug-In besitzt keine Mainmethode, da durch Starten des Plug-In Filters die Methoden setup und run automatisch aufgerufen werden. Bevor der Plug-In Filter gestartet werden kann, muss die zu komprimierende Datei bzw. das Bild wie in 3.1 beschrieben, vorher in ImageJ geöffnet werden. In der Methode setup wird das in ImageJ geöffnete Bild als lokale Variable gespeichert. In der run-Methode wird zuerst der Image-Prozessor dem Bild zugeordnet, welcher ein Datentyp von ImageJ ist.

4.3 Methode setRGBPixels

Nachdem das Bild in Pixel zerlegt wurde, müssen nun die einzelnen Farbwerte voneinander getrennt werden. Das Plug-In liest die verschiedenen Integerwerte der einzelnen Rot-, Grün- und Blauwerte aus. Diese werden nach Farben sortiert gespeichert.

4.4 Methode greyScaleCheck

Das Plug-In arbeitet nur mit Bildern aus dem RGB-Farbraum. Daher werden die Rot-, Grün- und Blauwerte verglichen. Wenn alle drei denselben Wert aufweisen, handelt es sich um ein Grauton. So wird das ganze Bild überprüft, ob es schwarz-weiß ist. Diese Information entscheidet, ob das Bild im folgenden Code besser komprimiert werden kann. Allerdings überprüft die Methode greyScaleCheck nicht, ob es vom Datentyp ein schwarz-weiß Bild ist.

4.5 Umsetzung der Lauflängencodierung

Die Lauflängencodierung wird in der Methode run_length_coding realisiert. Diese Methode wird vor der Huffmancodierung ausgeführt. Bevor die Lauflängencodierung durchgeführt wird, wird zuerst die Methode toByte aufgerufen, die die Pixelwerte in ein ByteArray schreibt. Dabei werden die einzelnen Pixelwerte aber nicht zusammen aufgeschrieben. Sondern es werden alle Rotwerte des gesamten Bildes von oben links nach unten rechts in das Array eingelesen und danach alle Blau- und Grünwerte auf dieselbe Art und Weise. Das liegt der Überlegung zu Grunde, dass es wahrscheinlicher ist, dass mehrfach derselbe Pixelwert einer Farbe auftritt, als dass der genaue Farbton mehrfach übereinstimmt. Dadurch ist die Lauflängencodierung effizienter.

4.6 Methode finalizebytearray

Bevor das Plug-In die Huffmancodierung ausführt, wird die Methode finalizebytearray aufgerufen. In dieser Funktion werden nur zusätzliche Informationen, die das Wiederherstellen des Bildes erleichtern, hinzugefügt.

4.7 Umsetzung der Huffmancodierung

Die Huffmancodierung erfolgt in der Methode huffman_encoding. Dabei wird ein öffentliches package genutzt, welches die Huffmancodierung als Befehl nutzen kann. Am Ende wird ein Bitstrom ausgegeben.

5 Testreihe 1

Um die Funktionalität des Plug-Ins nachweisen zu können, wurden einige Tests mit verschiedenen Bildern und Formaten durchgeführt. Dazu gehörten schwarz-weiß Bilder, einfarbige Grafiken und bunte Bilder in den Formaten PNG, JPG und NEF.

5.1 Kompression

Die Kompression war in fast allen Fällen erfolgreich. Bei einem Bild führte die Kompression zu einem Fehler. Dies geschah bei der Datei testbild1_jpg_schwarz_weiß.jpg. Die Fehlermeldung ist in Abbildung 12 zu sehen.

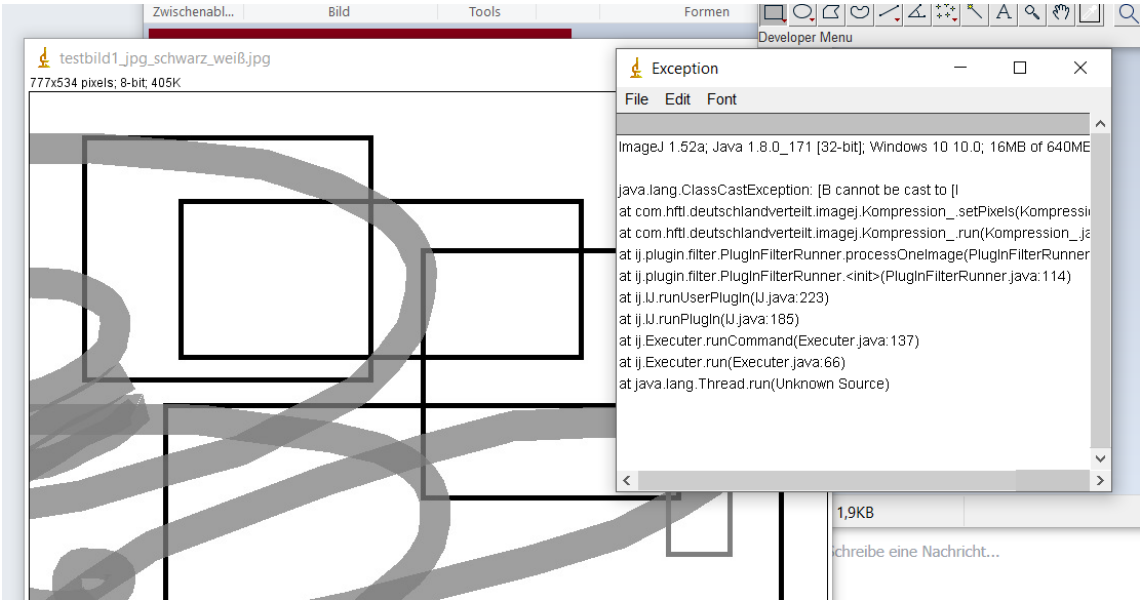

Abbildung 14: Fehlermeldung bei der nicht erfolgten Kompression

Allerdings wurden farbig aufwendige Bilder beim Komprimieren größer als die Originaldatei. Der Grund dafür war, dass eine Lauflängencodierung ineffizient ist, sobald keine oder nur sehr kleine homogene Abschnitte vorliegen.

Name	Größe vorher	Größe nachher
einfarbig.png	1.967 Byte	1.289 Byte
einfarbig.jpg	4.062 Byte	1.289 Byte
DSC_0119.NEF	21.104.751 Byte	47.992 Byte
free-abstract-background-11-vector.jpg	18.693 Byte	499.688 Byte
html-color-codes-color-tutorials-hero-00e10b1f.jpg	145.205 Byte	2.345.184 Byte
testbild1_png_schwarz_weiß.png	21.706 Byte	10.889 Byte

Tabelle 1: Vergleich Vorher-Nachher-Größe der komprimierten Bilder

Einfarbige und schwarz-weiß Bilder konnten mit einer Kompressionsrate um 50% verkleinert werden.

Auch Bilder im Format .NEF konnten erfolgreich komprimiert werden. Das Beispielbild wurde von 21.104.751 Bytes zu 47.992 Bytes komprimiert.

5.2 Dekompression

Die Dekompression erfolgte bei allen erfolgreich komprimierten Bildern fehlerfrei. Jedes Bild konnte problemlos wiederhergestellt werden. Bei einem Vergleich zwischen Originalbild und wiederhergestellten Bild waren keine Veränderung mit bloßen Auge wahrzunehmen.

6 Testreihe 2

Für die 2. Testreihe werden nur synthetische Bilder im Rohdatenformat verwendet, da sonst das Kompressionsverhältnis nicht eindeutig bewertbar ist. PNG und JPG Bilder sind bereits mehrfach komprimiert. Deshalb sind die ungeeignet, um die Effizienz des Plug-Ins festzustellen. Die Grafiken dieser Testreihe sind zum Teil um Firmenlogos. Die Rechte an diesen Bildern / Logos besitzen die entsprechenden Firmen.

6.1 Kompression

Alle Bilder der Testreihe konnten erfolgreich mit dem Plug-In Filter komprimiert werden.

Name	Größe vorher	Größe nachher
audio_speakers_PNG11165.tif	355.354 Byte	200.249 Byte
battery_PNG12034.tif	3.893.914 Byte	2.217.439 Byte
bmw_logo_PNG19714.tif	1.920.154 Byte	458.074 Byte
coffee_machine_PNG17306.tif	2.413.378 Byte	1.730.884 Byte
headphones_PNG7657.tif	4.070.554 Byte	2.452.162 Byte
recycle_PNG31.tif	4.179.754 Byte	52.415 Byte
robot_PNG96.tif	13.320.154 Byte	180.616 Byte
samsung_logo_PNG12.tif	3.954.154 Byte	90.701 Byte
server_PNG52.tif	17.280.154 Byte	2.970.186 Byte
wifi_PNG50.tif	480.154 Byte	12.084 Byte

Tabelle 2: Vergleich Vorher-Nachher-Größe der komprimierten Bilder

6.2 Dekompression

Das Plug-In konnte alle Grafiken erfolgreich komprimieren und wiederherstellen. Die erfolgreiche Wiederherstellung wurde geprüft, indem die wiederhergestellte Datei und die Originaldatei in ImageJ geöffnet wurden. Dann wurde über den Menüpunkt Prozess zu Image Calculator weiter navigiert. Für Image1 und 2 wurden die Originaldatei und das wiederhergestellte Bild ausgewählt und als Operation wurde Subtract gesetzt. (siehe Abb. 13).
Substract dividiert die einzelnen Pixelwerte der beiden Grafiken voneinander. Wenn diese gleichgroß sind sollte also für jeden Pixel der Wert 0 erscheinen. Der Farbcode

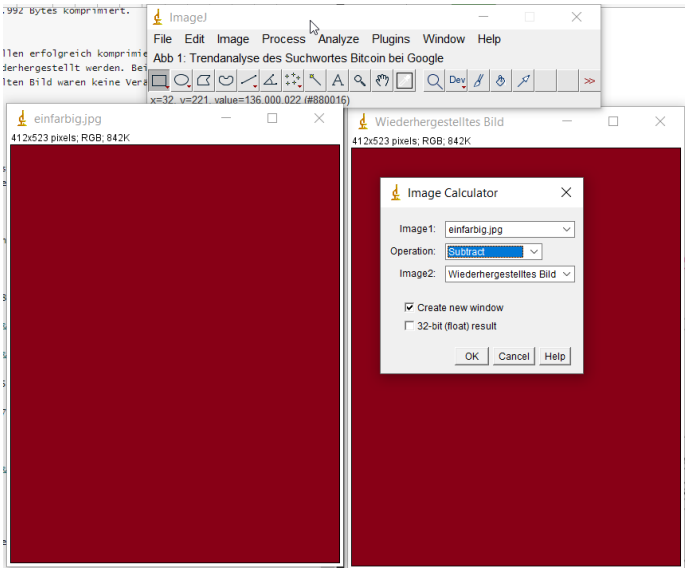

Abbildung 15: Anwendung Image Calculator

#000000 ist die Farbe schwarz. Daraus lässt sich ableiten, dass bei erfolgreicher Wiederherstellung des Bildes, mit der Operation Subtract ein schwarzes Bild erscheint. (Abb. 14).

Dass, das entstandene Bild tatsächlich ganz schwarz ist, wurde über Analyze, Mesaure bewiesen. Weil die Mean-, Min- und Maxwerte alle den Wert null haben, handelt es sich um ein komplett schwarzes Bild und somit wurde bewiesen, dass das komprimierte Bild verlustfrei wiederhergestellt werden konnte.

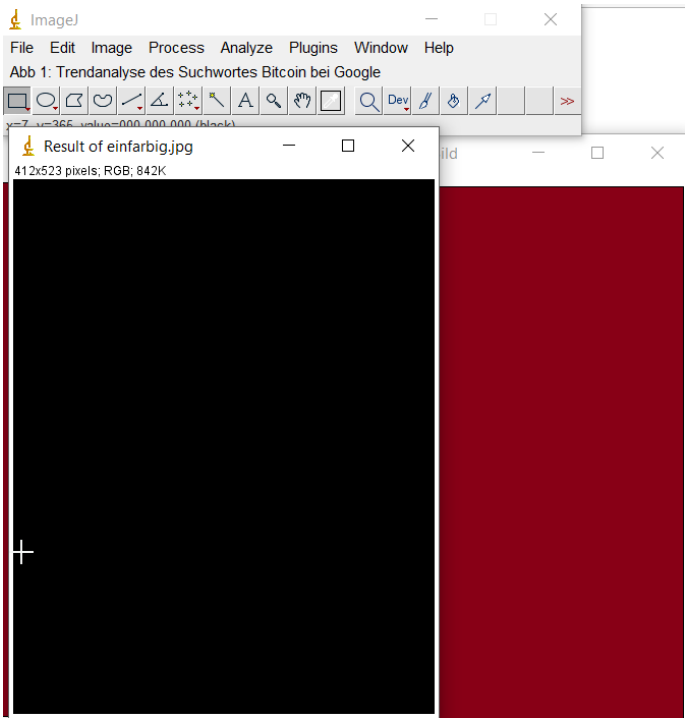

Abbildung 16: Subtract

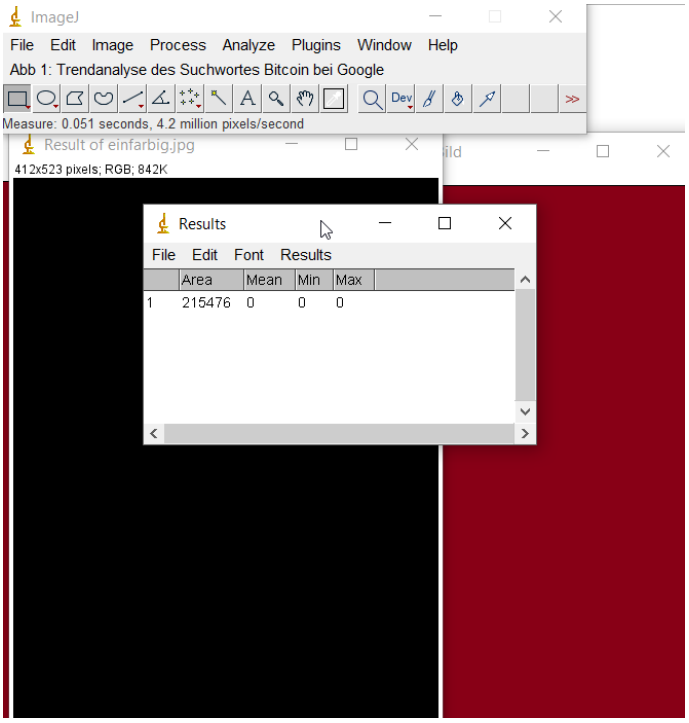

Abbildung 17: ideale Messwerte

7 Bewertung des Plug-Ins

7.1 Probleme

Die Lauflängencodierung funktioniert nur für eine maximale Lauflänge von 255. Das liegt daran, dass für die Anzahl für die Anzahl der aufeinanderfolgenden Pixelwerte nur 1 Byte reserviert ist.

Ein JPG Bild der Testreihe wurde leider nicht erfolgreich komprimiert. Allerdings wurde dasselbe Bild als PNG erfolgreich komprimiert. Die Ursache für diesen Fehler konnte nicht festgestellt werden.

Außerdem können nur Bilder im Farbbereich RGB komprimiert werden.

7.2 Effizienz

Das Plug-In funktionierte den Anforderungen gemäß.Rohdatenformate komprimiert das Plug-In sehr gut und fehlerfrei. Ein NEF Bild aus der Testreihe 1 war 439,7 mal kleiner nach der Kompression. Allerdings waren die eingelesenen Bilder im Format JPG/PNG nach der Komprimierung zum Teil größer als vorher. Die Ursache dafür ist, dass die gängigen Bildformate wie PNG oder JPG bereits mehrfach komprimiert und deshalb auch effizienter sind. Unkomprimierte Bildformate wurden deutlich kleiner als vorher.

Ein anderes Effizienzproblem trat noch bei der Lauflängencodierung auf, da im Worst Case Szenario die Datei doppelt so groß wird. Wenn ein Pixelwert nur einmal hintereinander auftritt, wird die Anzahl (also eins) trotzdem davor gesetzt.

8 Zusammenfassung

Zusammenfassend ist zu sagen, dass das erstellte Plug-In den festgelegten Anforderungen und Zielen entspricht. Bilder wurden verlustfrei mit Huffman- und Lauflängencodierung komprimiert. Unter dem Gesichtspunkt der zweifachen Kompression wurde ein gutes Kompressionsverhältnis erzielt. Verbesserungsmöglichkeiten bestanden noch bei der Lauflängenkompression, wie in 6.2 in Effizienz beschrieben. Es konnten alle von ImageJ unterstützten Bildformate und die Farbstufen RGB und schwarzweiß komprimiert werden. Das Plug-In ist in ImageJ einbindbar und auch über dessen Nutzeroberfläche startbar.

9 Literaturverzeichnis

Literatur

[BCK01] Bodden, Eric ; Clasen, Malte ; Kneis, Joachim: Arithmetische Kodierung. In: *Lehrstuhl für Informatik IV der RWTH Aachen* (2001)

[Don02] Donat, Lars: *Downloads*. `https://www.tu-chemnitz.de/informatik/ThIS/downloads/courses/ws02/datkom/Huffman-ShannonFano.pdf`. Version: 2002. – [Online; Stand 9. Mai 2018]

[Ima17a] ImageJ: *Downloads*. `https://imagej.net/Downloads`. Version: 2017

[Ima17b] ImageJ: *Welcome*. `https://imagej.net/Welcome`. Version: 2017

[Lip18] Lipinski, Klaus: *Downloads*. `https://www.itwissen.info/RLE-run-length-encoding-Lauflaengencodierung.html`. Version: 2018

[Str09a] Strutz, Tilo: Codierungstheorie. In: *Bilddatenkompression*. Springer, 2009, S. 31–33

[Str09b] Strutz, Tilo: Huffman-Codierung. In: *Bilddatenkompression*. Springer, 2009, S. 35–38

[Str09c] Strutz, Tilo: Kriterien zur Kompressionsbewertung. In: *Bilddatenkompression*. Springer, 2009, S. 10–14

[Str09d] Strutz, Tilo: Redundanz und Irrelevanz. In: *Bilddatenkompression*. Springer, 2009, S. 15–16

[SW11] Sedgewick, Robert ; Wayne, Kevin: *Algorithms*. Addison-Wesley Professional, 2011

[Wik18a] Wikipedia: *Datenkompression*. `https://de.wikipedia.org/w/index.php?title=Datenkompression&oldid=177081427"`. Version: 2018. – [Online; Stand 12. Mai 2018]

[Wik18b] Wikipedia contributors: *Peak signal-to-noise ratio — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Peak_signal-to-noise_ratio&oldid=841928208`. Version: 2018. – [Online; accessed 20-May-2018]