Datenkompression mit adaptiven Wavelets

Praktikumsbericht "Wissenschaftliches Rechnen"
WS 2010/2011 Universität Heidelberg

Wilma Trick, Lutz Hofmann, Manuel Hofmann, Florian Sonner

24. Januar 2011

Inhaltsverzeichnis

1 Problemstellung

1.1 Ausgangspunkt

In den Naturwissenschaften kommt es häufig vor, dass eine Messreihe eine große Anzahl von Messwert-Tupeln (x, y) liefert. Eine solche Messwerttabelle kann als stückweise konstante Funktion $f(x) := y$ betrachtet werden. Die Schwankungen dieser Funktion können in verschiedenen Bereichen des Messbereichs unterschiedlich stark ausgeprägt sein. Dies führt dazu, dass das Abspeichern von Messpunkten (x, y) in gleich großen Messintervallen Δx sehr ineffizient ist: In Bereichen großer Schwankungen muss Δx ausreichend klein sein, in Bereichen mit nahezu konstantem Funktionsverlauf genügen wenige Messwerte mit großem Δx.

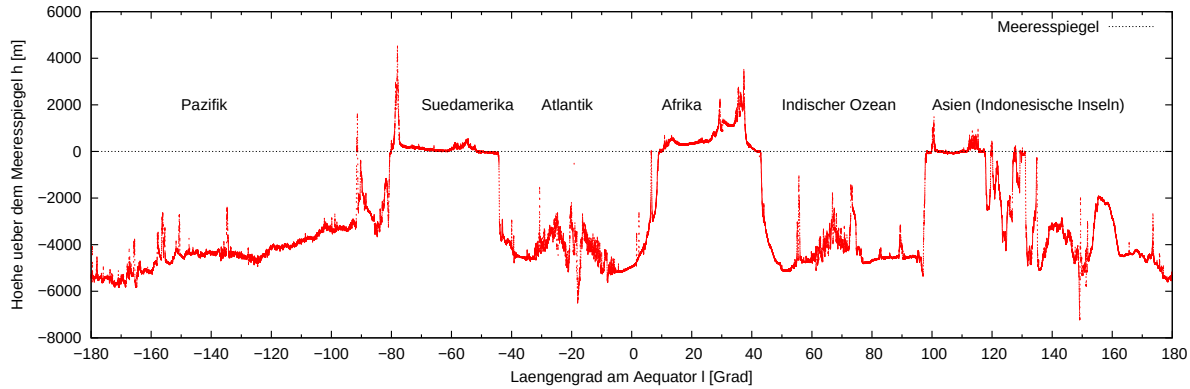

Abb. 1: Tiefen- bzw. Höhenprofil der Erde am Äquator [3]

Beispiel. In Abb. 1 wurde das Höhen- bzw. Tiefenprofil der Erde am Äquator als Höhe h in Metern über dem Meeresspiegel in Abhängigkeit vom Längengrad l dargestellt. Die Daten aus [3] listen Werte im Messintervall $\Delta l \approx 0.008°$ auf - insgesamt also 43200 Messwerte (l, h). Offensichtlich könnte diese Datenmenge ohne großen Informationsverlust reduziert werden, wenn Δl auf den Kontinenten und in manchen Bereichen des Ozeans größer gewählt werden würde.

1.2 Ziel

Ziel des Sofware-Projekts "Wissenschaftliches Rechnen für Anfängerßoll sein, mithilfe geeigneter Algorithmen eine effiziente Methode zur Komprimierung solcher Daten zu finden und in einem Programm umzusetzen.

Hierzu wird die Methode der **Gauß-Approximation** verwendet und die sogenannten **Haar-Wavelets** als Basisfunktionen gewählt. Diese Haar-Wavelets sind stückweise konstante Funktionen deren Träger geschachtelt sind: Je größer also die Schwankungen einer zu approximierenden Funktion in einem Bereich sind, desto mehr Basisfunktionen werden dort benötigt und desto kleiner muss Δx gewählt werden, um Informationsverlust zu vermeiden.

$\Longrightarrow$ Die Theorie der Gauß-Approximation und Haar-Wavelets wird im Kapitel *"Approximation mit Haar-Wavelets"* vorgestellt und im Kapitel *"Haar-Wavelets: Algorithmen"* werden Algorithmen zur Umsetzung dieser Methode erläutert.

1.2.1 Anforderungen an das Programm

- Das Programm soll Daten, wie z.B. die in Abb. 1 dargestellten, einlesen können.
- Für verschiedene Toleranzen[1] soll die Anzahl der Basisfunktionen ausgeben werden, die benötigt werden, um die Daten ausreichend genau zu approximieren.
- Die Daten sollen in komprimierter Form ausgegeben werden, sodass sie mithilfe von Grafikprogrammen (z.B. gnuplot) dargestellt werden können.

[1] Die Toleranz ist eine Zahl, die die Größe des erlaubten Informationsverlustes beschreibt.

2 Approximation durch Haar-Wavelets

2.1 Theorie

Um eine Funktion zu approximieren, möchte man ein System von Funktionen mit folgenden Eigenschaften:

- $(\psi_i, \psi_j) = \delta_{i,j}$, Orthonormalität
- $\Psi_N \subset \Psi_{N+1}$, Geschachteltheit
- $diam(tr(\psi_i)) = 0, \quad f\ddot{u}r \quad i \to \infty$

Die erste Eigenschaft erleichtert die Berechnung der Koeffezienten der Näerhungsfunktion, wie wir später sehen werden. Das Hinzufügen neuer Basisfunktionen sollte möglichst einfach sein, um seine Approximation eventuell verbessern zu können. Deswegen fordern wir als zweite Eigenschft, dass Basisfunktionen der Stufe N in der der Stufe $N+1$ enthalten sein soll. Die dritte Eigenschaft dient dazu, Funktionen mit starken Ausschlägen zu approximieren.

Folgende Notation wird benutzt:

$(f, g) = \int_a^b f(x)g(x)dx$ ist das Skalarprodukt für Funktionen.

$\Psi_N = span\{\psi_1, ..., \psi_N\}$ ist der Vektorraum aufgespannt durch die Basisfunktionen ψ_i

$tr(f) = \{x \in \mathbb{R} \mid f(x) \neq 0\}$ ist der Träger einer Funktion und

$diam(I) = b - a$,wobei $I = [a, b]$,also das Intervall der Länge $b - a$

Diese Eigenschaften besitzen die sogenannten Haar-Wavelets.

Definition 1 *Mother-Wavelet*

$$\psi(x) = \begin{cases} 1 & , \quad 0 < x \leq 1 \\ -1 & , \quad 1 < x \leq 2 \\ 0 & \quad sonst \end{cases}$$

wird als Mother-Wavelet bezeichnet und

$$\chi(x) = \begin{cases} 1 & , \quad 0 < x \leq 1 \\ 0 & \quad sonst \end{cases}$$

als Abschneidefunktion.

Für jedes $l \in \mathbb{N}_0$ (Stufe des Haar-Wavelets) gibt es insgesamt 2^{l-1} Haar-Wavelets, welche durch

$$\psi_i^l(x) = max(1, \sqrt{2^{l-1}})\psi(2^l x - 2i)\chi(x) \quad , 0 \leq i < 2^{l-1} \tag{2.1}$$

gegeben sind.
Die Haar-Wavelet Basis der Stufe l ist damit gegeben durch:

$$\Psi^l = \{\psi_0^0\} \cup \bigcup_{j=1}^{l} \bigcup_{i=0}^{2^l-1} \{\psi_i^j\} \tag{2.2}$$

Alternativ kann man das i-te Wavelet der Stufe $l > 0$ darstellen als:

$$\psi_i^l(x) = \begin{cases} 0 & , \quad x \leq \frac{2i}{2^l} \quad oder \quad x > \frac{2i+2}{2^l} \\ \sqrt{2^{l-1}} & , \quad \frac{2i}{2^l} < x \leq \frac{2i+1}{2^l} \\ -\sqrt{2^{l-1}} & , \quad \frac{2i+1}{2^l} < x \leq \frac{2i+2}{2^l} \end{cases}$$

Die Haar-Wavelets haben die Eigenschaft

$$tr(\psi_i^l) = tr(\psi_{2i}^{l+1}) \cup tr(\psi_{2i+1}^{l+1}) \tag{2.3}$$

Das bedeutet, dass man den Träger von ψ_i^l als Vereinigung der Träger seiner beiden Kinder $\psi_{2i}^{l+1}, \psi_{2i+1}^{l+1}$ sehen kann. Die Haar-Wavelets sind alle orthogonal zueinander.

$$(\psi_i^l, \psi_k^j) = \delta_{i,j}\delta_{l,k} = \begin{cases} 1 & , \ i = j \quad und \quad l = k \\ 0 & , \ sonst \end{cases}$$

Wie man durch Fallunterscheidung für i, j, k und l zeigen kann. Wegen (2) kann man erkennen, dass die Haar-Wavelet Basisfunktionen der Stufe l in der von Stufe $l + 1$ enthalten sind: $\Psi^l \subset \Psi^{l+1}$. Da die Träger zweier benachbarter Wavelets disjunkt sind, gilt wegen (3) weiterhin $\lim_{i\to\infty} diam(tr(\psi_i)) = 0$. Somit sind alle 3 Eigenschaften von oben erfüllt.
Möchte man eine (stückweise) stetige Funktion f durch Haar-Wavelets approximieren, so projeziert man f orthogonal in den Haar-Wavelet-Raum $S = span\{\Psi^l\}$. Damit ist sicher gestellt, dass der Fehler $||f - g|| = \sqrt{(f - g, f - g)}$ minimal wird. Die Aufgabe besteht also darin, dass man ein $g \in S$ findet mit

$$||f - g|| \to min \tag{2.4}$$

Dies gelingt mit Hilfe von

Satz 1 *Gauss-Approximation*

$g \in S$ erfüllt genau dann (4) wenn

$$(g, \phi) = (f, \phi) \qquad \forall \phi \in S \quad dimS < \infty \tag{2.5}$$

gilt. Ein solches g existert immer und ist eindeutig. (ohne Beweis)

Sei also $g = \sum_{j=1}^{N} \alpha_j \psi_j$. Die Koeffizienten α_j bestimmt man, indem bei (4) für ϕ alle Basisfunktionen aus S einsetzt.

$$(g, \psi_i) = (f, \psi_i) \quad , i = 1, \ldots, N \tag{2.6}$$

$$\iff \sum_{j=1}^{N} \alpha_j (\psi_j, \psi_i) = (f, \psi_i) \tag{2.7}$$

$$\iff \alpha_i = (f, \psi_i) \tag{2.8}$$

$$\Rightarrow \quad g = \sum_{i=1}^{N} (f, \psi_i) \psi_i \tag{2.9}$$

Welches genau der orthogonalen Projekion entspricht.
Der Fehler dieser Approximation ist damit

$$||f - g||^2 = (f, f) - \sum_{i=1}^{N} (f, \psi_i)^2 \tag{2.10}$$

Daraus lässt sich auch ableiten, dass durch Hinzufügen weitere Basisfunkionen der Fehler höchstens kleiner wird.

2.2 Algorithmen

2.2.1 Funktionsprinzip

Um das Prinzip hinter den unten beschriebenen Algorithmen zu verstehen, ist vor allem das Verst ndnis von Haar-Wavelets als Baum hilfreich. Dieser wird von den Tr gereigenschaften der Wavelets induziert: ψ_i^{j+1} ist Kind von ψ_i^j genau dann wenn $tr(\psi_i^{j+1}) \subset tr(\psi_i^j)$. F r ψ_0^0 ist ψ_0^1 das einzige Kind, f r einen Knoten ψ_i^j $(j \geq 1)$ sind es ψ_{2i}^{j+1} und ψ_{2i+1}^{j+1}.

2.2.2 Auswertung von Funktionen in einer Haar-Wavelet-Basis

Sei eine Haar-Wavelet-Basis gegeben durch (ψ_i^j) mit $j = 0, \ldots, l$, $i = 0, \ldots, 2^{j-1} - 1$. Da die einzelnen Basisfunktionen auf gro en Bereichen identisch mit der Nullabbildung sind, l ssst die Auswertung einer Funktion f gegeben in dieser Basis an einer Stelle $x \in (0, 1]$ dadurch optimieren, dass nur Basisfunktionen ψ_i^j betrachtet werden, f r die $x \in tr(\psi_i^j)$ gilt, das hei t $\psi_i^j(x) \neq 0$. Diese Erkenntnis alleine ist nat rlich nicht wirklich hilfreich, erst durch die Baumstruktur und die disjunkte Zerlegung des Intervalls durch die Tr germengen der Wavelets einer Stufe l sst sich die Auswertung optimieren: Geht man ausgehend vom Wurzelnkoten durch den Baum und betrachtet ab dem ersten Level stets die beiden Kinder des Knotens, so kann wegen der disjunkten Zerlegung x nicht in beiden Tr germengen enthalten sein, f r die weitere Auswertung ist also genau ein Knoten relevant. Das nicht nur der eine Knoten, sondern der gesamte an

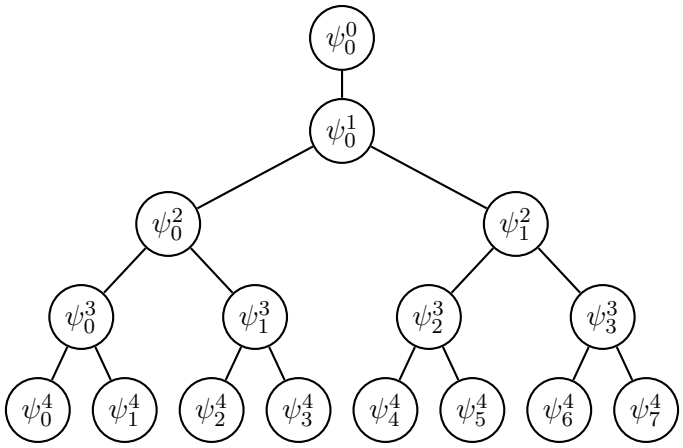

Abb. 2: Haar-Wavelets lassen sich als Baum anordnen, wenn man die Tr ger der einzelnen Funktionen betrachtet: ψ_i^{j+1} ist Kind von ψ_i^j genau dann wenn $tr(\psi_i^{j+1}) \subset tr(\psi_i^j)$.

diesem Knoten h ngende Teilbaum von der Betrachtung ausgeschlossen werden kann, liegt an der Inklusionseigenschaft der Tr germengen, ber welche der Baum definiert wurde.
In jedem Schritt wird also ein kompletter Teilbaum von der Betrachtung ausgeschlossen, die letztendlich benutzen Wavelets bilden demnach einen unverzweigten Pfad im Baum. Statt alle 2^l Wavelets auswerten zu m ssen, sind durch die Optimierung nur noch $2l$ Wavelets berhaupt in Betracht zu ziehen.

Gegeben: (ψ_i^j), (c_i^j) mit $j = 0, \ldots, l$, $i = 0, \ldots, 2^{j-1} - 1$, wodurch $f = \sum_{j=0}^{l} \sum_{i=0}^{2^{l-1}-1} c_i^j \psi_i^j$ definiert ist.

Gesucht: $f(x) = \sum_{j=0}^{l} \sum_{i=0}^{2^{l-1}-1} c_i^j \psi_i^j(x)$, mit $x \in (0, 1]$

Algorithmus:

```
f = c_0^0 ψ_0^0(x);
j = 1; i = 0;
while true do
  f = f + c_i^j ψ_i^j(x);
  if j == l then
    break;
  end if
  if x ∈ tr(ψ_{2i}^{j+1}) then
    i = 2 * i;
  else
    i = 2 * i + 1;
  end if
  j = j+1;
end while
```

2.2.3 Datenkompression mit Wavelets

Dieser Algorithmus dient dazu, eine m glichst kleine Wavelet-Basis f r eine Funktion f zu finden, so dass der Approximationsfehler unter einer bestimmten Toleranz liegt. Wiederum wird die Baumstruktur der Wavelet-Funktionen ausgenutzt: Da jedes Kind eines bereits in der Basis vorkommenden Knotens den Fehler h chstens reduziert, werden im Baum solange Knoten ausgew hlt, bis die Fehlertoleranz erreicht ist. Das Kriterium, mit welchem neue Knoten ausgew hlt werden, unterliegt einer gewissen Heuristik, in dem unten beschriebenen Algorithmus wird allein die Fehlerreduktion berpr ft.

Gegeben: Darstellung von $f \in S^l$ in Basis ψ^l

Gesucht: Teilraum $\tilde{S} \subset S^l$ so dass $||f - \tilde{f}|| \leq TOL \cdot (f, f)$, $\tilde{f} \in \tilde{S}$ mit $\tilde{S}$ m glichst klein. Die Ausgabe ist eine Indexmenge $I \subset \mathbb{N} \times \mathbb{N}$, mit $\tilde{S} = \{\psi_i^j | (j, i) \in I\}$ wird durch sie eine der m glichen Wavelet-Basen beschrieben.

Algorithmus:

$I = \{(0, 0)\}$;
$S = (f, \psi_0^0)^2$;
while $S < (1 - TOL)(f, f)$ **do**
  W hle Kind $(k, l + 1)$ eines $(i, l) \in I$ mit (f, ψ_k^{l+1}) maximal
  $I = I \cup \{(k, l + 1)\}$;
  $S = S + (f, \psi_k^{l+1})^2$;
end while

3 Benutzung des Programmes

Dieses Kapitel beschreibt die Funktionalität des Programmes aus der Sicht des Benutzers, dabei wird aber natürlich ein gewisses Grundverständnis der Wavelet-Approximation aus den letzten Kapiteln vorausgesetzt.

3.1 Funktionsprinzip

Bevor die Einzelheiten der Bedienung erläutert werden, ist ein grober Überblick über das Funktionsprinzip des Programmes hilfreich: Wie in dem Kapitel 'Implementierung' auf technischer Ebene näher erörtert wird, liest das Programm die Ausgangsdaten für die zu approximierende Funktion aus verschiedenen Quellen ein, berechnet die Approximation und gibt diese und eventuell weitere Informationen in verschiedenen Formaten aus. Das, für die Eingabe zuständige, Modul des Programmes wird `InputReader` und das, für die Ausgabe zuständige, `OutputWriter` genannt.
Eine mögliche Eingabequelle ist beispielsweise eine Textdatei, welche die XY-Koordinaten eines Funktionsgraphen enthält. Es kann aber auch eine Funktion sein (z.B. $sin(x^2)$), welche dann von dem Programm in einem äquidistanten Gitter ausgewertet wird. Das für den praktischen Gebrauch des Programmes gedachte Ausgabeformat schreibt dann den Graphen der Approximation in eine weitere Textdatei. Andere Ausgabeformate sind dagegen zum Visualisieren des generierten Wavelet-Basis-Baums oder zur Beleuchtung technischer Aspekte des Algorithmus gedacht. Genauere Informationen und eine vollständige Liste aller Eingabequellen und Ausgabeformate befinden sich in den entsprechenden Abschnitten weiter unten.

3.2 Bedienung

Das Programm wird allein über die Kommandozeile bedient und alle benötigten Daten, welche später noch detailliert erläutert werden, sind bereits beim Start als Parameter[1] an das Programm zu übergeben. Falls Daten fehlen, erhält der Benutzer einen entsprechenden Hinweis und die Ausführung des Programmes wird abgebrochen. Bei jedem Programmaufruf ist genau eine Eingabequelle und genau ein Ausgabeformat anzugeben. Die Angabe mehrerer Eingabequellen oder Ausgabeformate wird zur Zeit nicht unterstützt.

3.2.1 Programmaufruf

Die Syntax für den Programmaufruf lautet: `./program input output tolerance [parameters]`

[1] In diesem Bericht werden die Begriffe *Argument*, *Programmoptionen* und *Parameter* synonym verwendet.

Dabei steht `input` für die Eingabequelle, z.B. dem Pfad einer Textdatei, `output` für den Pfad zu der Datei, welche als Ausgabe generiert werden soll und `tolerance` für den Toleranzwert, welcher bei der Kompression verwendet werden soll, z.B. 0.02.
Diese drei Parameter sind auf jeden Fall anzugeben, allein im speziellen Hilfsmodus des Programmes, aufrufbar durch `./program --help`, der alle Parameter mit Beschreibungstexten auflistet, sind diese nicht notwendig.
In der praktischen Anwendung benötigt das Programm allerdings oft mehr als diese drei Parameter, diese können nach dem Toleranzwert angegeben werden. Die einzelnen Parameter besitzen dabei die Form `--name value` und werden durch Leerzeichen getrennt aufgelistet. Einige Parameter können auch in der abkürzenden Schreibweise `-n value` aufgelistet werden und einige benötigen auch keinen Wert, haben daher die Form `--name` bzw. `-n`.
Das Programm versucht selbständig aus den Eingaben für `input` und `output` abzulesen, um was für eine Art von Eingabe- und Ausgabeformat es sich wohl handelt. So wird zum Beispiel eine Eingabequelle mit der Endung .txt als Textdatei interpretiert und entsprechend eingelesen. Natürlich kann diese Wahl aber auch falsch sein, oder nicht genügend Informationen vorhanden sein, damit das Programm diese Wahl automatisch vornehmen kann. Für diese Fälle existieren die Parameter `-F value` für das Eingabe- und `-Ovalue` für das Ausgabeformat. Die Werte für diese Parameter sind aus der Liste mit Ein- und Ausgabeformaten zu entnehmen.
Die meisten Ein- und Ausgabeformate lassen sich in ihrem Verhalten von weiteren Parametern steuern, insgesamt existieren sechs Parameter, welche unabhängig von der Wahl der Formate angegeben werden können:

`--help` – gibt eine Liste mit allen Parametern und ihren Beschreibungen aus

`-F value` – legt das Eingabeformat fest (siehe Absatz oben)

`-O value` – legt das Ausgabeformat fest (siehe Absatz oben)

`--fast` – benutze schnellere Kompressionsmethode, die allerdings nicht immer optimal ist

`--info` – als Konsolenausgabe des Programmes erscheinen einige Informationen zur Kompression

`--W` – überschreibe Ausgabedatei falls diese vor der Ausführung bereits existierte

3.3 Eingabeformate

3.3.1 Textdateien

Dateien mit den Endungen .txt und .dat werden automatisch von diesem `InputReader` eingelesen, allgemein sind allerdings alle einfachen Textdatei-Formate einlesbar, wenn sie dem folgenden Format entsprechen:

- Die Daten müssen **in Spalten angeordnet** sein, d.h. nach jeder Zeile muss ein Zeilenumbruch erfolgen.
- Die Daten in einer Zeile können durch **beliebige Separatoren** getrennt sein: Leerzeichen, Tab, Komma, Semikolon, aber auch beliebige Zeichenfolgen, die allerdings keine Zahlen enthalten dürfen.

- Vorsicht bei diesen Zeichen:
  - Das Zeichen “-” direkt vor einer Zahl wird stets als Minuszeichen und nie als Separatorzeichen interpretiert, z.B. wird ”4-3äls die Daten ”4ünd 3ëigelesen.
  - Die Zeichen “e” und “E” direkt hinter einer Zahl werden als Ankündigung eines Exponenten interpretiert, z.B. wird “4E3” als “4000” eingelesen.
- **Kommentarzeilen** sollten keine Zahlen enthalten. Enthalten sie dennoch Zahlen müssen sie mit einem vorstehenden “#”-Zeichen als Kommentarzeile gekennzeichnet werden.
- Die Tabelle darf in Spalten *vor* den einzulesenden x- und y-Spalten **keine leeren Einträge** haben, da der Reader leere Felder nicht als solche erkennen kann. In der Zeile *hinter* den einzulesenden x- und y-Einträgen dürfen jedoch beliebige Zeichenfolgen stehen, da diese bis zum nächsten Zeilenumbruch ignoriert werden.
- Es ist egal, in welcher Spalte die x- und in welcher die y-Werte stehen.

Parameter

Beispiel: `./program -F text input.txt -O graph output.txt -x 1 -y 2 0.1`

`input.txt` liest die Daten aus der Datei input.txt ein.

`-F text` wählt den Text-Input-Reader als den für diese Datei zuständigen Inputreader aus.

`-x 1` bestimmt, dass die einzulesenden x-Werte in der 1. Spalte stehen.

`-y 2` bestimmt, dass die einzulesenden y-Werte in der 2. Spalte stehen. Die y-Werte sind die Daten, die in den Wavelet-Baum gespeichert werden sollen.

`0.1` gibt die Toleranz an.

3.3.2 GEBCO-Dateien

Der Gebco-Input-Reader die NetCDF Dateien des General Bathymetric Chart of the Oceans (GEBCO) einlesen. Er kann insbesondere das *1-Minute-Grid* und das *30 Arc-Second-Grid* einlesen. Dies erkennt er automatisch indem er die Variable *dimension* einliest. Entspricht dieser Wert 43200 so geht er davon aus, dass er das *30 Arc-Second-Grid* einlesen soll, andernfalls das *1-Minute-Grid*. Das bedeutet er kann auch nur diese beiden Formate einlesen. Dieser Reader bekommt als Übergabewert einen Breitengrad und liest dann alle Höhendaten auf diesem Breitengrad aus. Dabei ist der Übergabewert in einem gewissen Format (d:m:s). Wobei d dem Grad, m den Minuten und s den Sekunden entspricht.Möchte man z.B. den Breitengrad $45°34'0''N$ einlesen so lautet der Übergabewert `-coord 45:34:0`. Schwieriger wird es nur, wenn man Breitengrade einzulesen will, die unterhalb des Äquators liegen. Möchte man z.B. den Breitengrad $0°1'0''S$ einlesen so lautet der Übergabewert `-coord -1:59:0`. Das liegt daran, dass der GebcoReader Nord und Süd nur anhand des Vorzeichen der Grad Angabe unterscheidet. Dabei wird wie folgt gerechnet: $-1°59'0'' := -1°0'0'' + 0°59'0'' = 0°1'0''S$ Will man also einen Breitengrad auf der Südhalbkugel auslesen, so muss man den Grad-Wert zum nächst höheren Wert ”aufrunden”.

Der GebcoReader überprüft weiterhin den Übergabewert auf Korrektheit, so würde z.B. der Wert $96°1'0''S$ keinen Sinn ergeben, da der Grad Wert zwischen 0° und 90° liegen muss. Ist der Wert nicht korrekt so bricht das Programm mit einer Fehlermeldung ab.
Weiterhin überprüft der Reader, ob der Übergabewert mit dem verwendeten Grid übereinstimmt. So ist z.B. $10°45'\mathbf{15}''N$ inkompatibel mit dem *1-Minute-Grid*, jedoch kompatibel mit dem *30 Arc-Second-Grid*. Das liegt daran, da das *1-Minute-Grid* die Höhendaten nur in Minutenschritten gespeichert hat. Analog wäre $10°45'\mathbf{0}''N$ inkompatibel mit dem *30 Arc-Second-Grid*, jedoch kompatibel mit dem *1-Minute-Grid*, was daran liegt, da die Sekundenangabe bei dem *30 Arc-Second-Grid* immer entweder $15''$ oder $45''$ sein muss.

Beispiel

```
./program gebco08.nc gebco1.txt 0.1 -F gebco -O graph --coord 70:10:15 -W
```

`gebco08.nc` liest die Datei ein.

`gebco1.txt` gibt die Output Datei an.

`0.1` gibt die Toleranz an.

`-F gebco` gibt das zu lesende Format an.

`-O graph` gibt das Output Format an (in diesem Fall eine von gnuplot lesbare Datei).

`--coord 70:10:15` gibt den zu lesenden Breitengrad an. ($70°10'15''N$).

`-W` Überschreibt die Output Datei

3.3.3 Funktionen

Dieser `InputReader` generiert die Eingabedaten für das Programm automatisch aus einem übergebenen Funktionsterm, z.B. `sin(x2)`. Die freie Variable im Funktionsterm ist stets x, neben den üblichen Operationen `+`, `-`, `*` und `/` sind auch eine Reihe von Funktionen bereits vordefiniert, wie `sin(·)`, `cos(·)`, `exp(·)`, `abs(·)`, `sqrt(·)` und `pow(·,·)` [2]. Desweiteren kann im Funktionsterm die Zeichenkette `pi` für die Wert der Konstante π benutzt werden.

Parameter

Neben dem Funktionsterm wird das Intervall benötigt, auf welchem die Funktion ausgewertet werden soll, so wie die Anzahl der durchzuführenden Auswertungen. All diese Informationen lassen sich in einem einzigen Parameter unterbringen:

`--range a:b:steps` gibt den Bereich an, in dem die Funktion ausgewertet werden soll. Dabei stehen `a` und `b` für die Grenzen des Auswertungsintervalls $[a, b]$ und `steps` für die Anzahl der für die Auswertung zu benutzenden Schritte.

[2] Eine vollständige Liste mit gültigen Operationen und vordefinierten Funktionen befindet sich auf der folgenden Webseite: `http://warp.povusers.org/FunctionParser/fparser.html#functionsyntax`

Beispiele

`./program 'sin(x^2)' test.txt 0.1 -F func -O graph --range 1:4:1024` wertet die Funktion $sin(x^2)$ auf dem Intervall $[1, 4]$ in 1024 äquidistanten Schritten aus, d.h. mit einer Schrittweite von $\frac{4-1}{1024} \approx 0.0029$.

`./program 'exp(x) + pi * 2^x' test.txt 0.1 -F func -O graph --range -10:-2:2048` wertet die Funktion $exp(x) + \pi 2^x$ auf dem Intervall $[-10, -2]$ in 2048 äquidistanten Schritten aus, d.h. mit einer Schrittweite von $\frac{|-10-(-2)|}{2048} \approx 0.0039$.

3.4 Ausgabeformate

3.4.1 Histogramm

Dieser OutputWriter bietet die Möglichkeit die Fehler aller Wavelet Basisfunktionen graphisch darzustellen. Dabei werden die Fehlerintervalle logarithmisch auf der x-Achse und die entsprechende Anzahl in y Richtung abgetragen. Der OutputWriter bieten zwei Optionen an:

`--depth (default:-8)` Gibt die untere Schranke der Ausgabe an.

`--intervals (default:10)` Gibt die Anzahl der Unterteilungen an.

Die x-Achse läuft von der unteren Schranke $10^{\text{depth}}\%$ (default: $10^{-8}\%$) bis zu 100%. Wobei die 100% die gesamte Fehlernorm im Quadrat ist.
In der Regel gibt es noch ein paar Fehler, die zwischen 0% und $10^{\text{depth}}\%$ liegen. Diese werden zum letzten Balken bei $10^{\text{depth}}\%$ dazugezählt. Korrekterweise endet dieser Balken also nicht bei $10^{\text{depth}}\%$ sondern erst bei 0%, was in einer logarithmischen Anzeige jedoch nicht dargestellt werden kann.
Als Ausgabe wird eine von gnuplot lesbare Textdatei mit vier Spalten erzeugt.

Spalte 1: Gibt die Ober- und Untergrenzen der Fehlerintervalle als Punkte auf der x-Achse aus. Format: 10^a

Spalte 2: Wie Spalte 1, jedoch nur den Exponent. Format: a

Spalte 3: Gibt die Höhe der Histogramm-Balken **für den unkomprimierten Baum** aus (unabhängig von der eingestellten Toleranz).

Spalte 4: Gibt die Höhe der Histogramm-Balken **für den mit der angegebenen Toleranz komprimierten Baum** aus.

Für eine schöne Darstellung ruft man in gnuplot z.B. folgenden Befehl auf:

```
plot 'datei.txt' using 2:3 with lines
```

Wählt man Spalte 2 für die x-Werte, kann linear geplottet werden. Bei Spalte 1 für die x-Werte empfiehlt sich eine logarithmische Darstellung (`set logscale x`). Die y-Achse kann analog mit `set logscale y` logarithmisch dargestellt werden. Je nachdem ob man Spalte 3 bzw. 4 für die y-Werte angibt, erhält man das Histogramm für den unkomprimierten bzw. komprimierten Baum.

Beispiel - sin(x)

Abbildung 3 wurde mit folgenden Einstellungen erzeugt:
`./program -F func 'sin(x)' --range -3.14:3.14:2048 -W 'out.txt' -O hist --info --intervals 100 --depth -10 0.01`

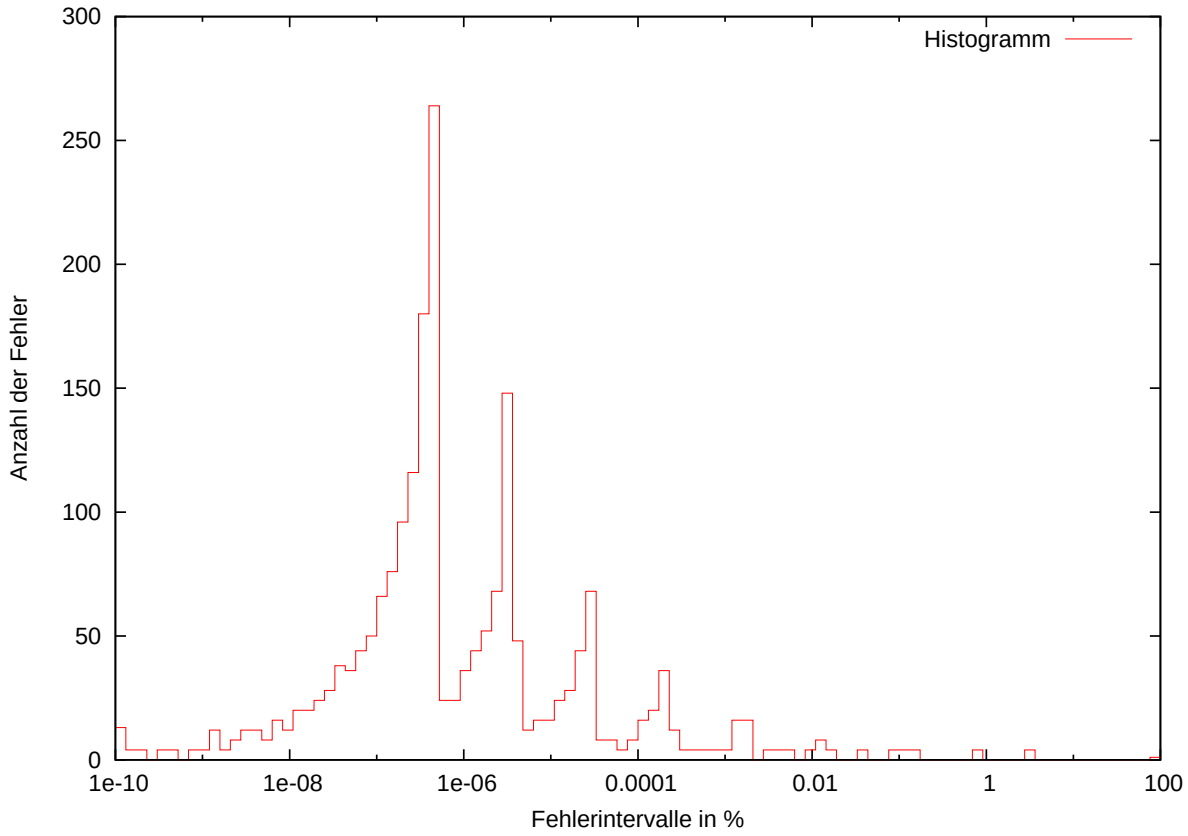

Abb. 3: Fehlerhistogramm von $sin(x)$ auf dem Intervall $[-\pi, \pi]$.

3.4.2 Ausgabe eines Graphen

Der GraphOutputWriter ist dafür zuständig den Graphen der Waveletbasisfunktionen zu gegebener Toleranz in eine von gnuplot lesbare Datei zu schreiben. Dazu schreibt er die x- und y-Werte jeweils in eine Spalte und speichert dies in eine Textdatei. Es werden keine weiteren Parameter benötigt, es genügt also OutputWriter `graph` anzugeben.

Beispiel

`./program 'x' graph.txt 0.1 -F func -O graph --range -1:1:1024`
speichert die Approximation von der linearen Funktion $f(x) = x$ mit einer Toleranz von 0.1 in die Datei `graph.txt`. Der GraphOutWriter dient hauptsächlich dazu, die Original Daten mit den komprimierten Daten bzw. komprimierte Daten verschiedener Toleranzen optisch zu vergleichen.

3.4.3 DOT Graphen

Dieses Ausgabeformat dient primär zur Visualisierung der Algorithmen: Es werden alle Wavelet-Basisfunktionen als Graph in der DOT Sprache gespeichert. Die DOT Sprache ist ein textuelle

Beschreibungssprache für Graphen[3], welche von einer Reihe von Programmen visualisiert und dann bspw. als Grafik gespeichert werden kann. In jedem Knoten, d.h. zu jeder Basisfunktion, werden Level, Scale und Koeffizient als Beschriftung verwendet.
Achtung: Dieses Ausgabeformat wird bei mehr als 50 Basisfunktionen aus Gründen der Übersichtlichkeit nicht empfohlen. Die Visualisierung der Graphen stößt außerdem sehr schnell an technische Grenzen, auch wenn bei der Generierung der DOT-Datei selbst keine Probleme auftreten.

Parameter

`--markdiffs` aktiviert die Markierung von Basisfunktionen, welche nicht zur Kompression gehören. In diesem Fall werden die Knoten von solchen Funktionen durch gepunktete Linien dargestellt, ansonsten besitzen alle Funktionen die gleiche Darstellung.

Beispiel

Die Diskretisierung der Funktion $f(x) = x$ auf dem Intervall $[0, 1]$ in 8 konstante Funktionen wird mit einem Toleranzwert von 0.5 komprimiert und der erzeugte DOT-Graph in einer Textdatei gespeichert:

```
./program 'x' graph.dot 0.5 -F func -O dot --range 0:1:8 --markdiffs
```

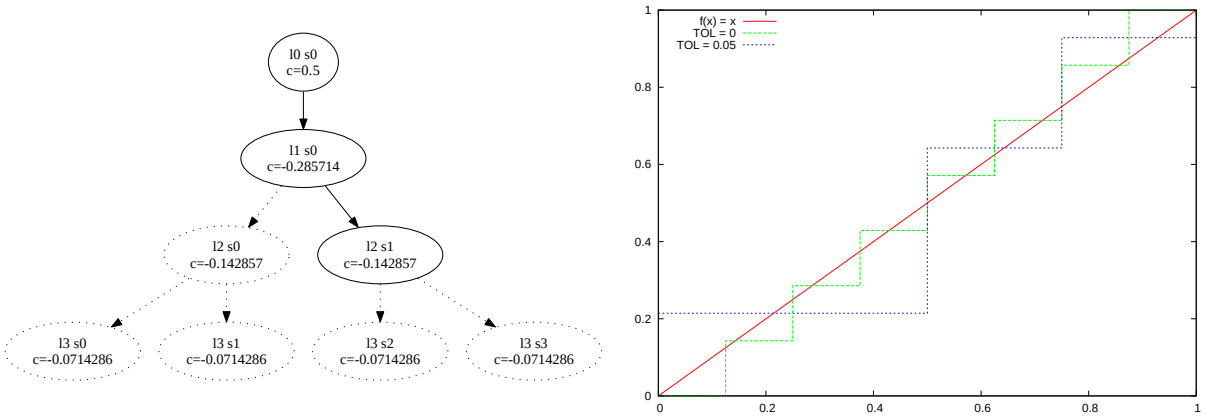

Abb. 4: Links die Visualisierung des mit obigen Befehl erzeugten DOT-Graphen. Die bei der Kompression ausgelassenen Basisfunktionen werden durch gepunktete Linien dargestellt. Rechts sind die Funktionsgraphen der kontinuierlichen Funktion (rot), deren Diskretisierung (grün, 10 Basisfunktionen) und Kompression (blau, 3 Basisfunktionen) dargestellt.

[3]Nähere Informationen zur DOT Sprache: `http://en.wikipedia.org/wiki/DOT_language`

4 Validierung

In diesem Abschnitt soll die korrekte Funktionsweise des (optimalen) Algorithmus' anhand verschiedener Beispiele nachgewiesen werden.

4.1 Validierung anhand eines Fehlerhistogramms

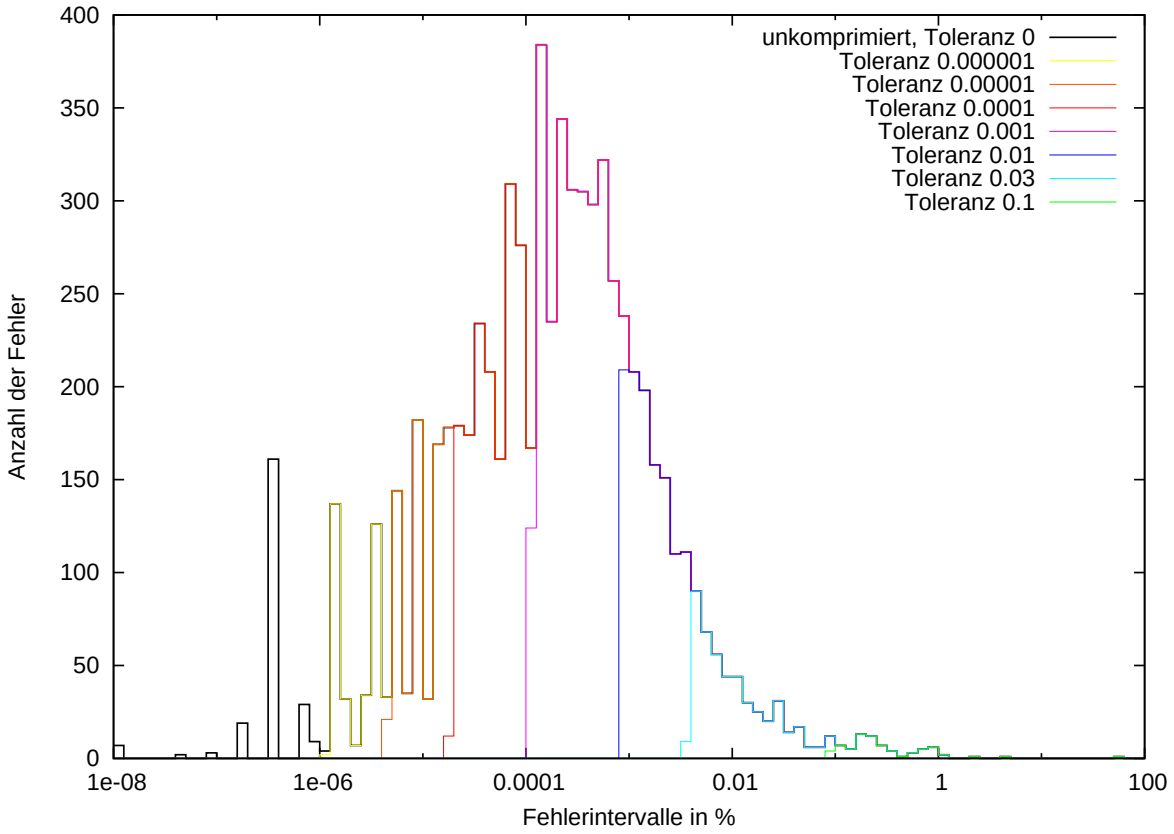

Abb. 5: Validierung der Wirksamkeit des "optimalen Algorithmus" anhand eines Fehlerhistogramms mit den Beispieldaten vom Deutschen Wetterdienst [4] (siehe auch Analyse und Anwendungen).

Beispielhistogramm. In Abbildung 5 wurden die Beispieldaten mit der mittleren Tagestemperatur am Frankfurter Flughafen von 1991 bis 2010 vom Deutschen Wetterdienst [4] in einem Histogramm umgesetzt. Der Bereich von 0% bis 100% des Gesamtfehlers wurde in 100 logarithmische Intervalle unterteilt und darüber jeweils die Anzahl von Basisfunktionen mit Fehlern im entsprechenden Intervall abgetragen. Es werden die Histogramme für den unkomprimierten Bau mit denen für komprimierte Bäume mit den Toleranzen 0.1, 0.03, 0.01, 0.001, 0.0001, 0.00001

und 0.000001 verglichen.
Die Fehlerverteilung entspricht, da es sich um eine unregelmäßige, statistische Ausgangsfunktion handelt, ungefähr einer Gauß'schen Verteilung.

Diskussion. Aus diesem Histogramm ist ersichtlich, dass bei der Komprimierung nacheinander stets jene Basisfunktion mit dem größten Fehler zur Basis hinzugenommen wird. Von "rechts" (der Bereich mit Fehlern, die den größten Beitrag liefern) wird im Histogramm ein ganzer Balken nach dem anderen hinzugefügt; vom letzten Balken "links" im entsprechenden Histogramm jedoch nur ein Teil des Balkens. Dies ist genau, was erwartet wird: Der Algorithmus stoppt mit dem Hinzufügen von Basisfunktionen, sobald die Toleranz erreicht ist. Alle Funktionen mit Beiträgen größer als das letzte hinzugefügte Element, sind bereits in der Basis enthalten.
Je kleiner die Toleranz wird, desto mehr Basisfunktionen werden hinzugefügt.

Fazit. Der (optimale) Algorithmus verfährt genau wie erwartet: Die Basisfunktionen werden in Reihenfolge der Größe ihrer Fehlerbeiträge zur Basis hinzugenommen. (Einen Vergleich mit dem Histogramm für den "schnellen Algorithmus" findet sich in Abbildung 11.)

4.2 Validierung anhand einer Stufenfunktion

Als nächstes wird die Funktionsweise des Algorithmus' an einer einfachen Stufenfunktion demonstriert. Diese Funktion soll so einach sein, dass man die Funktionsweise der Kompression voraussagen kann. Diese Voraussage soll anschließend mit dem Ergebnis des Programms verglichen werden und die Korrektheit des Programms damit bestätigt werden.
Es wird folgende Stufenfunktion auf dem Intervall $[0, 1]$ verwendetet:

$$f(x) = \begin{cases} 1 & \text{für } x \leq \frac{1}{8} \\ -1 & \text{für } x > \frac{1}{8} \end{cases}$$

Da der FunctionInputReader nicht für unstetige Funktionen wie eine Stufenfunktion geeignet ist, wurde die Funktion als .txt-Datei mit $32768 = 2^{15}$ Datenpunkten eingelesen. Es wurde eine Zweierpotenz gewählt, um die Stufe bei $\frac{1}{8}$ möglichst exakt in der Datenreihe darstellen zu können. Selbstverständlich hätte dies auch mit jeder anderen Zweipotenz funktioniert.

Voraussage. Für die exakte Darstellung in der Waveletbasis werden genau folgende vier Basisfunktionen benötigt:

$$\begin{aligned}
\psi_0^0(x) &= \begin{cases} 1 & \text{für } 0 < x \leq 1 \\ 0 & \text{sonst} \end{cases} \\
\psi_0^1(x) &= \sqrt{2^0} \cdot \begin{cases} 1 & \text{für } 0 < x \leq \frac{1}{2} \\ -1 & \text{für } \frac{1}{2} < x \leq 1 \\ 0 & \text{sonst} \end{cases} \\
\psi_0^2(x) &= \sqrt{2^1} \cdot \begin{cases} 1 & \text{für } 0 < x \leq \frac{1}{4} \\ -1 & \text{für } \frac{1}{4} < x \leq \frac{1}{2} \\ 0 & \text{sonst} \end{cases} \\
\psi_0^3(x) &= \sqrt{2^2} \cdot \begin{cases} 1 & \text{für } 0 < x \leq \frac{1}{8} \\ -1 & \text{für } \frac{1}{8} < x \leq \frac{1}{4} \\ 0 & \text{sonst} \end{cases}
\end{aligned}$$

Die “Kompression” mit nur einer Basisfunktion wird durch die konstante Funktion ψ_0^0 erreicht. Es wird erwartet, dass bei einer Kompression mit zwei Basisfunktionen als nächstes $\psi 0^3$ zur Basis hinzugenommen wird, da hiermit die Stufe bei $\frac{1}{8}$ modeliert werden kann.

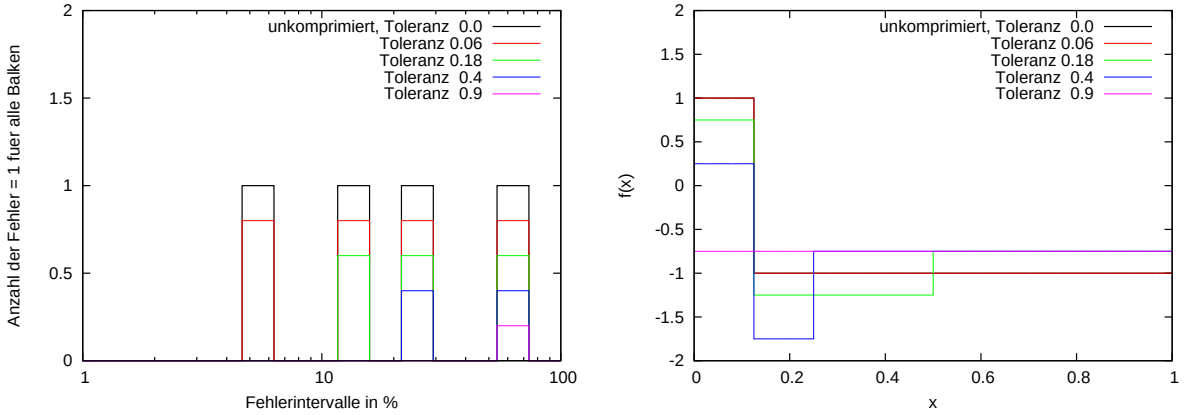

Abb. 6: Kompression einer Stufenfunktion auf dem Intervall $[0, 1]$ mit Stufe bei $\frac{1}{8}$. *Links:* Fehlerhistogramm. (In Wahrheit enthält jeder Balken genau ein Fehlerelement (Anzahl = 1). Um jedoch anschaulich zu machen, dass mit größer werdender Toleranz immer eine Basisfunktion weniger zur Basis dazugenommen wird, wurden die Balken unterschiedlich hoch geplottet.) *Links:* Kompression der Sufenfunktion für die Toleranzen 0.0, 0.06, 0.18, 0.4 und 0.9.

Diskussion. In Abbildung 6 wurde die Stufenfunktion nun mithilfe des Programms ausgewertet. Im Fehlerhistogramm (links) für den unkomprimierten Baum (schwarze Linie) erkennt man sofort, dass das Programm nur insgesammt vier Fehlerelemente benötigt, um die Funktion darzustellen. Dies deckt sich mit unserer Vorhersage. Die Toleranzen 0.06 (rot, 4 Basisfunktionen), 0.18 (grün, 3 Basisfunktionen), 0.4 (blau, 2 Basisfunktionen) und 0.9 (magenta, 1 Basisfunktion) wurden so gewählt, dass jeweils eine Basisfunktion weniger dazugenommen wird. Für Toleranzen ≤ 0.06 bekommt man bereits die exakte Darstellung der Stufenfunktion.
An den beiden Kurven für die Toleranzen 0.9 (magenta) und 0.4 (blau) erkennt man genau das, was oben angenommen wurde: Für eine Kompression mit zwei Basisfunktionen erhalten wir eine Superposition aus der konstanten Funktion $\psi_0^0(x)$ und der Basis $\psi_0^3(x)$, die der Stufe bei $\frac{1}{8}$ entspricht. Auch die Koeffizienten lassen sich bei dieser einfachen Superposition ablesen:

$$f_{\text{TOL:0.4}}(x) = -0.75 \cdot \psi_0^0(x) + 1 \cdot \psi_0^3(x)$$

An dem Koeffizient der Basisfunktion $\psi_0^3(x)$ mit dem Wert 1 lässt sich bereits erkennen, dass sich bei einer verlustfreien “Kompression” mit 4 Basisfunktionen die Basisfunktionen $\psi_0^0(x)$, $\psi_0^1(x)$ und $\psi_0^2(x)$ im Intervall [0,0.25] zu Null aufaddieren werden.
Die blaue Kurve hat im Vergleich zur Ursprungsfunktion bei $\frac{1}{4}$ noch eine Stufe zu viel. Dies kann durch das Hinzunehmen der Basisfunktionen $\psi_0^2(x)$ (grüne Kurve) und $\psi_0^1(x)$ (rote Kurve) ausgeglichen werden.

Fazit. Anhand einer einfachen Stufenfunktion wurde gezeigt, dass das Programm die Kompression genau so durchführt, wie erwartet wird: Obwohl die einzulesende Datei mit $32768 = 2^{15}$ Datenpunkten recht umfangreich ist, ermittelt das Programm korrekterweise, dass diese stückweise konstante Funktion mit der Stufe bei $\frac{1}{8}$ von nur vier Basisfunktionen exakt dargestellt werden kann. Auch die Reihenfolge, in der die Basisfunktionen hinzugenommen werden, entspricht dem,

was zuvor angenommen wurde. Besonders an der magentafarbenen (1 Basisfunktion) und an der blauen (2 Basisfunktionen) Kurve in Abbildung 6 (rechts) ist die Superposition von Basisfunktionen zur komprimierten Funktion zu erkennen. Dies bestätigt die korrekte Berechnung der Koeffizienten.
Ferner hat dieses Beispiel auch gezeigt, dass sich sehr einfache, stückweise konstante Funktionen bereits für noch vergleichsweise große Toleranzen (hier: 0.06) exakt darstellen lassen.

4.3 Validierung anhand einer linearen Funktion

In diesem Abschnitt soll durch die Erzeugung künstlicher Daten die Korrektheit des Algorithmus' bestätigt werden. Dazu wird mit Hilfe des FunctionInputReader eine reelle Funktion eingelesen. Diese Funktion soll komplex genug sein (nicht stückweise konstant), um mögliche Fehler zu entdecken. Die Funktion soll jedoch einfach gehalten werden, damit man leicht überprüfen kann, ob das Wavelet Approximationsprogramm korrekt funktioniert.
Wir haben uns dazu entschieden die Funktion $f(x) = x$ auf dem Intervall $[0, 1]$ zu nehmen. Zur Vereinfachung wird eine 2er Potenz an Datenpunkten gewählt (1024 Stück), da diese der Algorithmus am effektivsten verarbeiten kann. Das Programm wurde dabei wie folgt aufgerufen:
`./program -F func 'x' -O OUTPUT 'out.txt' --range 0:1:1024 --intervals 100 -W --info --depth -3 TOL`
Wobei für `OUTPUT` jeweils `graph` und `hist` benutzt wurde, also einmal die Ausgabe als normaler Graph und Fehlerhistogramm.

Diskussion Man würde bei "geschickt" (siehe Tabelle 4.3) gewählten Toleranzen (`TOL`) folgendes erwarten:

- Die Funktion wird gleichmässig approximiert, d.h. es werden nur die Waveletbasisfunktionen hinzugenommen, die sich im Wavelet-Tree bis zu einem bestimmten Level befinden.

- Alle Fehler auf demselben Level sind exakt gleich.

- Im Fehlerhistogramm sind nur vereinzelte "Türme" sehen, die nach links immer grösser werden.

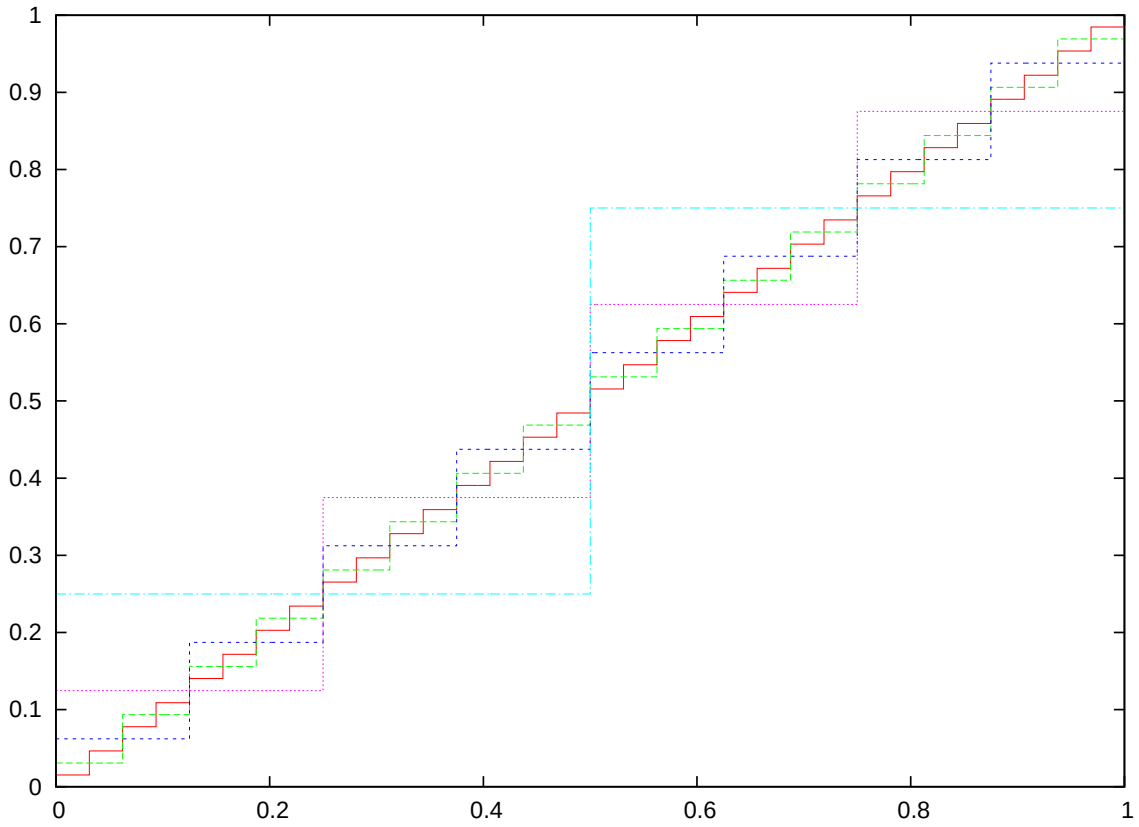

Abb. 7: Approximation von $f(x) = x$ mit verschiedenen Toleranzen. Beginnend mit 2 Waveletbasisfunktionen werden zur nächst kleineren Toleranz die Basisfunktionen verdoppelt.

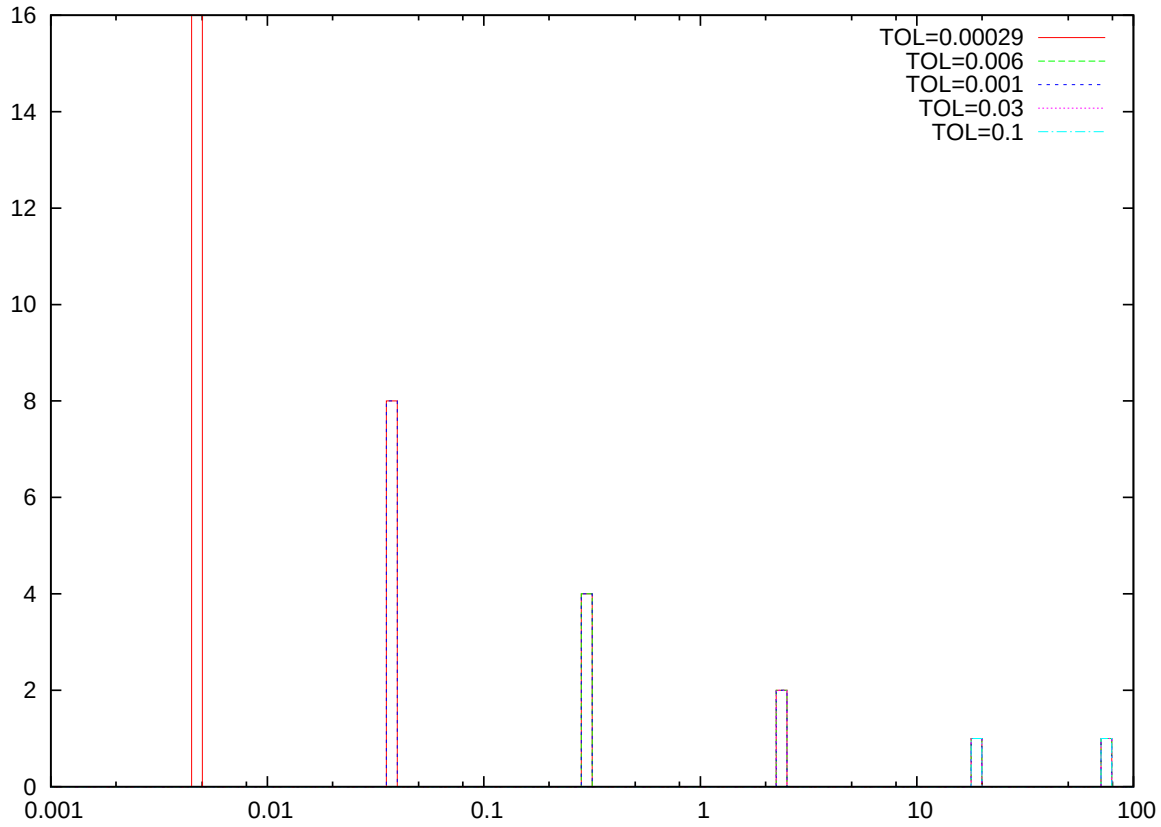

Abb. 8: Fehlerhistogramm zu Abbildung 7.

`TOL` (gewählte Toleranz)	tatsächlich erreichte Toleranz	Anzahl Basisfunktionen
0.1	0.0625914	2
0.03	0.0156477	4
0.006	0.00391174	8
0.001	0.000977756	16
0.00029	0.00024426	32

Abb. 9: Vergleich gewählter Toleranzen zu erreichter Toleranz und Waveletbasisfunktionen.

Fazit Die erwarteten Eigenschaften von oben sind eingetreten und man sieht sogar in Abbildung 7 die Hierarchie der Waveletbasisfunktionen. Dieses Beispiel bestätigt die Korrektheit des Algorithmus'.

5 Analyse und Anwendungen

In diesem Kapitel werden konkrete Anwendungen des Programmes vorgestellt und die Ausgaben des Programmes analysiert und diskutiert.

5.1 Mittleren Tagestemperatur am Frankfurter Flughafen

Beschreibung der Daten. Die in diesem Abschnitt ausgewerteten Daten geben die mittlere Tagestemperatur an der Wetterstation am Frankfurter Flughafen in 2 Metern Höhe über dem Erdboden vom 1. Januar 1991 bis zum 8. Dezember 2010 wieder. Die Daten stammen von der Homepage des Deutschen Wetterdienstes [4], wurden kopiert und als .txt-Datei gespeichert. Die auszuwertenden Daten für das Tagesmittel stehen bei [4] in Spalte 6 unter "TM".
Die erste Zeile muss mit "#äuskommentiert werden (da Zahl in Kommentarzeile), ansonsten kann die Datei genau so eigelesen werden.

In den Abbildungen 10 und 12 werden die Originaldaten (grün) mit den Waveletkomprimierungen durch den "schnellen Algorithmus" (blau) und den "optimalen Algorithmus" (rot) für verschiedene Toleranzen verglichen. Der Ausschnitt rechts stellt jeweils die vergrößerte Temperaturentwicklung für den Winter 1996/1997 dar.
Abbildung 11 stellt die Fehlerhistogramme für den schnellen und den optimalen Algorithmus für die Toleranzen 0.3, 0.2, 0.1, 0.05, 0.01 und 0.001 einander gegenüber

Diskussion. *Schneller Algorithmus:* Bei einer Toleranz von 0.05 werden die Temperaturpeaks für alle Sommer und Winter vor dem Winter 2007/2008 sehr genau nachmodeliert. Nach diesem Winter wird die Temperatur als konstant bei 12°C angegeben.
Ist die Toleranz 0.1, stellt man für den Bereich Winter 2004/05 bis Winter 1991/2000 schon eine starke Approximation fest, während das restliche Diagramm noch keine allzugroßen Unterschiede zum Diagramm mit 0.05 Toleranz aufweist. Dieser Trend lässt sich auch in dem Ausschnitt erkennen: Manche Abschnitte werden noch sehr genau, manche schon eher grob approximiert.
Für eine Toleranz von 0.2 nimmt die Genauigkeit schon stark ab und variiert stark. Der Bereich Winter 2010/11 bis Sommer 2008 wird nach wie vor durch einen konstanten Wert approximiert, der Bereich Winter 2004/05 bis Sommer 1998 ebenso. Die Bereiche Winter 2007/08 bis Sommer 2005 und ab Winter 1997/1998 sind noch relativ genau nachgebildet; die Komprimierung wird aber auch schon in diesen Bereichen sehr stark deutlich, wie der Ausschnitt für Toleranz 0.2 zeigt.

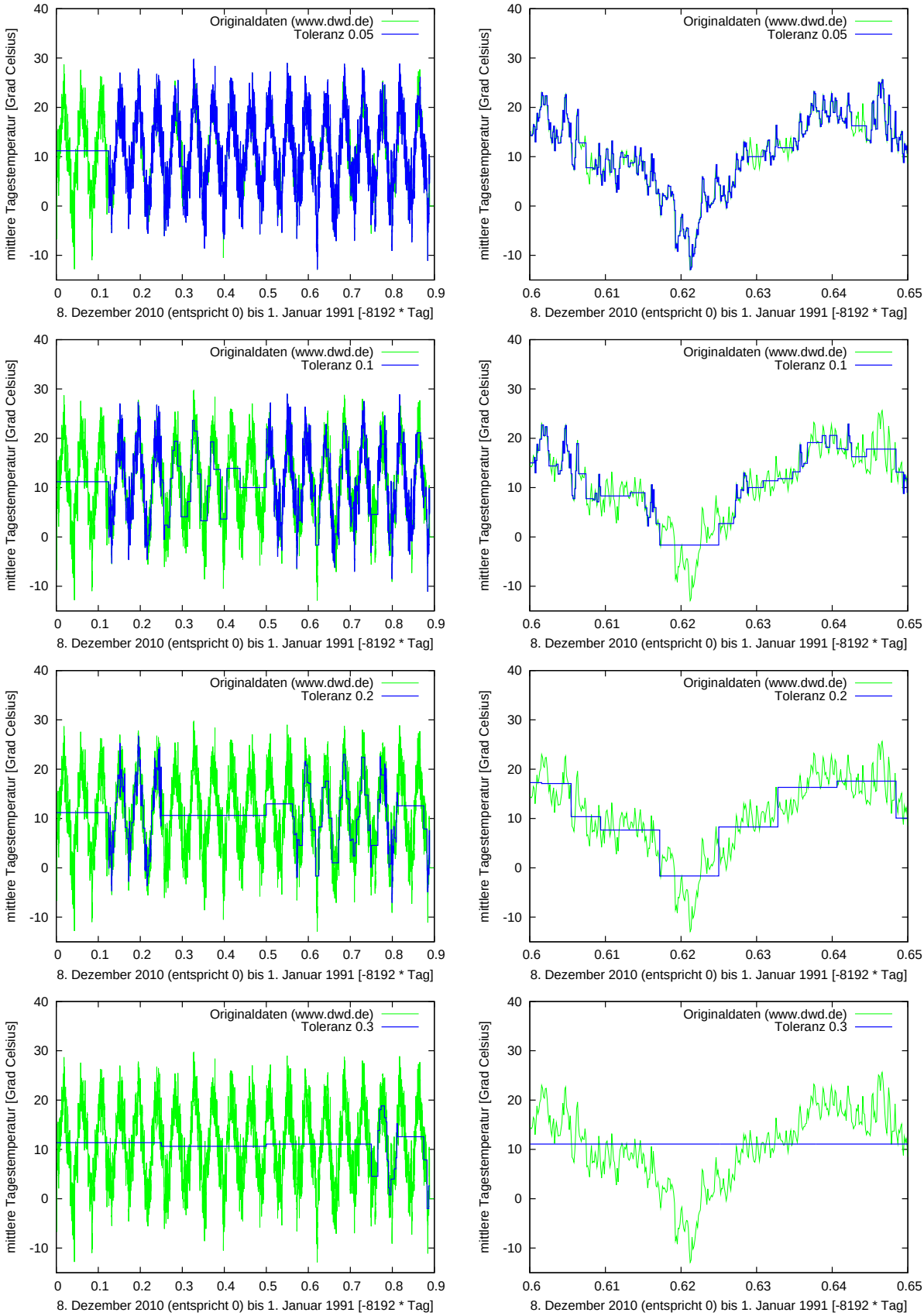

Abb. 10: Mittlere Tagestemperatur, Dezember 2010 - Januar 1991. Vergleich der Originaldaten von [4] mit Waveletkomprimierungen durch den **"schnellen Algorithmus"** mit unterschiedlichen Toleranzen (0.05, 0.1, 0.2, 0.3).

Bei einer Toleranz von 0.3 werden die Daten schon über weite Teile durch konstante Funktionen ersetzt, nur an einzelnen Stellen (z.B. Winter 1993/1994 bis Sommer 1992) erreicht die Approximation noch eine gewisse Ähnlichkeit mit den Daten.
optimaler Algorithmus: Während für den schnellen Algorithmus die Toleranzen 0.3, 0.2, 0.1 und 0.05 betrachtet wurden, lässt sich die Wirkungsweise des optimalen Algorithmus besonders anschaulich für die Toleranzen 0.1, 0.05, 0.01 und 0.001 dargstellen. Der größte Unterschied zum schnellen Algorithmus ist, das stehts jeder Sommer und jeder Winter in den komprimierten Daten erkennbar sind. Die Genauigkeit der Approximation nimmt mit kleiner werdender Toleranz gleichmäßig in allen Bereichen des Diagramms zu. Dies lässt sich ebenso in dem kleinen Ausschnitt des Winters 1996/1997 feststellen.
Fehlerhistogramme: Beim optimalen Algorithmus werden stets die größten Fehlerelemente hinzugefügt. Für sinkende Toleranzen sind, ausgehend von der rechten Seite des Histogramms, immer mehr komplette Balken im Histogramm enthalten. Nur jeweils der Balken ganz links wird nur zum Teil hinzugefügt. Beim schnellen Algorithmus hingegen stellt man zwar ebenfalls eine Tendenz fest, dass die Fehlerelemente vor allem von rechts aus hinzugefügt werden (d.h. große Fehlerelemente), jedoch werden stets Fehler aus dem gesamten Bereich zwischen 0% und 100% hinzugefügt. Noch bevor alle Elemente aus einem Fehlerintervall hinzugefügt wurden, werden auch schon Funktionen mit kleineren Fehlern zur Basis hinzugenommen.

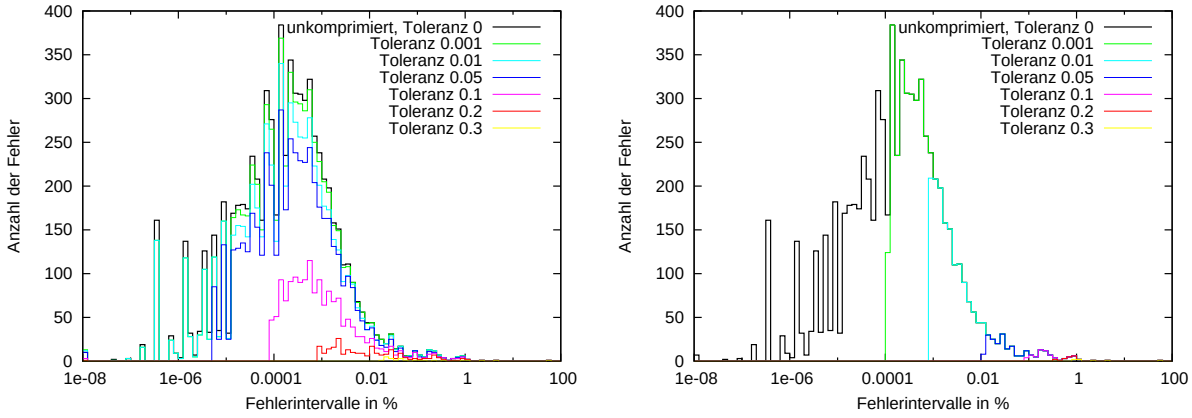

Abb. 11: Vergleich der Fehlerhistogramme für die Daten vom Deutschen Wetterdienst [4] (mittlere Tagestemperatur, Dezember 2010 - Januar 1991). Der **"schnelle Algorithmus"** (links) wird dem **"optimalen Algorithmus"** (rechts) gegenübergestellt. Es sind die Histogramme für die Toleranzen 0.3, 0.2, 0.1, 0.05, 0.01 und 0.001 abgebildet.

Fazit. Das Einlesen von Daten aus Textdateien mithilfe des TextInputReaders funktioniert, ebenso die Waveletapproximation mit verschiedenen Toleranzen. Allerdings ist die Approximation für den schnellen Algorithmus sehr, sehr ungleichmäßig. Dieses Problem tritt beim optimalen Algorithmus jedoch nicht mehr auf. Die Kompression ist hier gleichmäßig. Der Grund für die ungleichmäßige Kompression des schnellen Algorithmus ist sofort aus dem Vergleich der Fehlerhistogramme ersichtlich: Der Unterschied zwischen den beiden Algorithmen liegt in der Reihenfolge, in der Basisfunktionen zur Basis hinzugenommen werden.
Bei Datenreihen wie dieser, die nur ungefähr 7000 Werte enthält, ist der Aufwand für die optimale Kompression vertretbar und daher dem schnellen Algorithmus immer vorzuziehen.

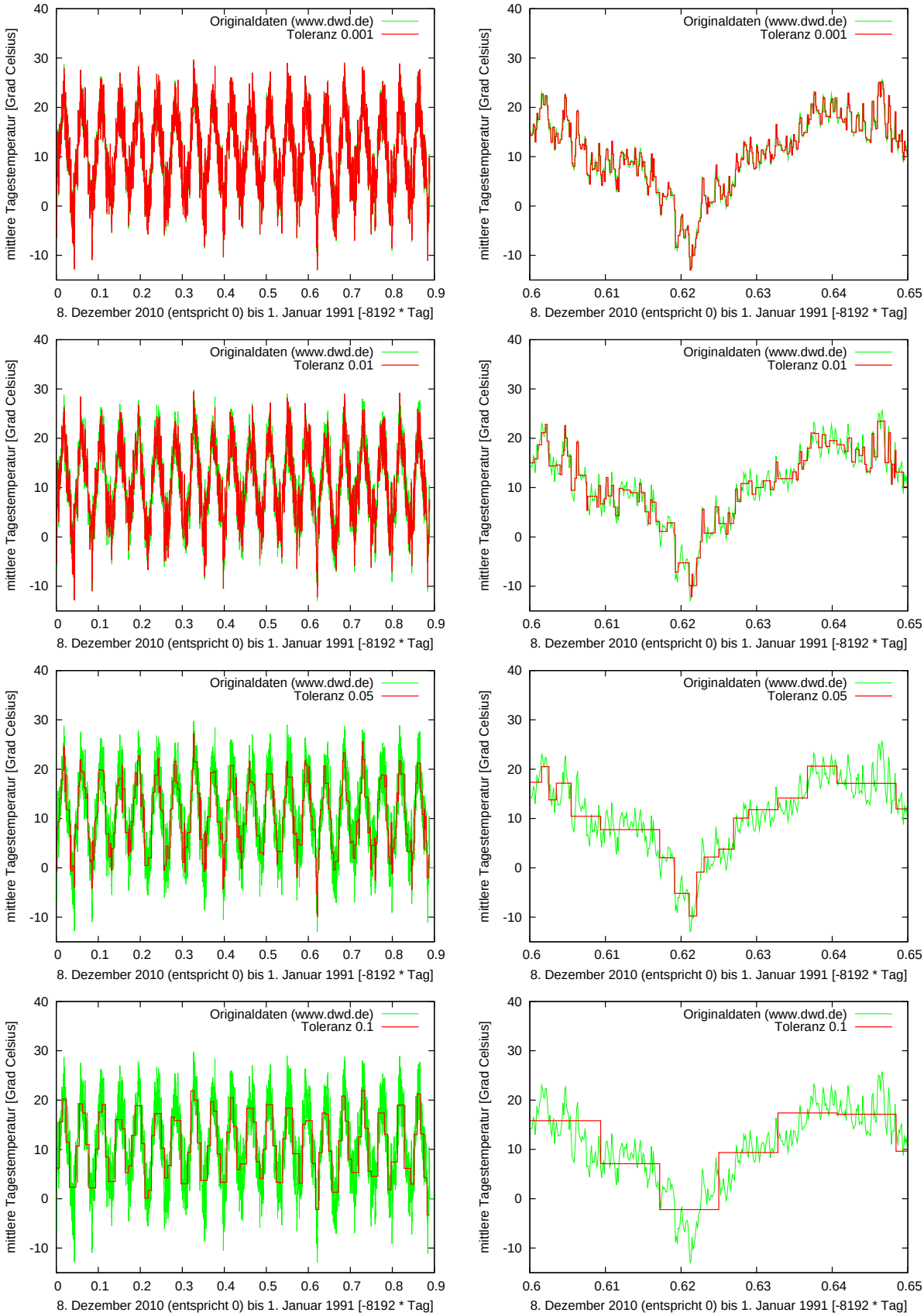

Abb. 12: Mittlere Tagestemperatur, Dezember 2010 - Januar 1991. Vergleich der Originaldaten von [4] mit Waveletkomprimierungen durch den **"optimalen Algorithmus"** mit unterschiedlichen Toleranzen (0.001, 0.01, 0.05, 0.1).

5.2 Gebco Daten

In diesem Abschnitt werden mit Hilfe des Gebco-Input-Readers Datenpunkte aus den NetCDF Dateien des General Bathymetric Chart of the Oceans (kurz GEBCO) eingelesen und diskutiert. Es wird der Breitengrad nahe am Äquator ($0°0'15''N$) eingelesen. Insgesamt sind es immer 43200 Datenpunkte, was man mit genau der selben Anzahl an Waveletbasisfunktionen exakt approximieren kann. Also ist ein Vergleich von Toleranz und verwendeter Basisfunktion ein guter Schätzer zur Güte der Kompression.

Diskussion. Bei einer Toleranz von 0.1 werden die Daten nur sehr grob approximiert. Das einzige was sich aus Abbildung 13 ablesen lässt ist, wo sich ungefähr die Landmassen und die Meere befinden, wobei selbst Asien (0.5-0.56) unterhalb des Meeresspiegels liegt. Dafür werden immerhin nur 12 Waveletbasisfunktionen verwendet um einen Fehler von 10% zu erreichen.
Setzt man die Toleranz auf 0.01 werden die Urprungsdaten schon recht gut genähert. Es lassen sich einige Konturen erkennen und einige Peaks erscheinen bereits wie z.B. die Anden (ungefähr bei 0.18 in der Abbildung) in Südamerika. Dabei werden nur 94 Basisfunktionen benutzt. Verkleinert man also die Toleranz um das 10-fache so werden in diesem Beispiel nur knapp das 8-fache an Basisfunktionen benötigt. Setzt man die Toleranz auf 0.001 so lassen sich in der gröbsten Ansicht von gnuplot kaum noch Unterschiede zwischen den Ursprungsdaten und der Waveletfunktion erkennen. Nur dort wo die Original GEBCO Daten auf kleinen Intervallen stark oszillieren bleibt die Waveletfunktion konstant. Bei einer Toleranz von 0.001 werden bereits insgesamt 589 Basisfunktionen verwendet. Möchte man stark oszillierende Bereiche in der Approximation haben, so muss man die Toleranz noch weiter runter setzen.
In Abbildung 14 werden die beiden Algorithmen verglichen. In diesem Beispiel wurde eine Toleranz von 0.01 benutzt und das Schaubild zeigt nur die Fehler von den Basisfunktionen, die hinzugefügt wurden bis die Toleranz erreicht wurde. Man sieht sehr gut, wie der optimale Algorithmus die Fehler von rechts auffüllt, da er ja im gesamten Baum nach den grössten Fehlern sucht. Anders sieht es beim schnellen Algorithmus aus. Bei dieser Toleranz schafft er es nicht den einen der beiden Fehler bei 0.1 zu erreichen. Um dennoch unter die Toleranz zu kommen, muss es dadurch viel mehr Basisfunktionen hinzufügen (im Vergleich 94 zu 150 Funktionen).
In Abbildung 15 sieht man die gesamte Fehlerverteilung im Baum. Wie im Histogramm oberhalb wurde eine Toleranz von 0.01 verwendet. Wie zu erwarten gibt es einen Bereich, in dem sich die meisten Fehler befinden und links davon einige Peaks. Für den Histogram-Output-Writer musste man eine Tiefe von -12 benutzen, damit man alle Fehler sehen kann.
In beiden Beispielen wurde `--intervals 100` benutzt.

Fazit. Das Einlesen der Gebco NetCDF Dateien funktioniert und die Daten werden jetzt mit beliebigen Toleranzen approximiert. Anhand de Diskussion am Beispiel der Fehlerhistogramme ist es sinnvoll den optimalen Algorithmus vorzuziehen, da er weniger Basisfunktionen benötigt und gleichmässiger die Fehler verteilt.

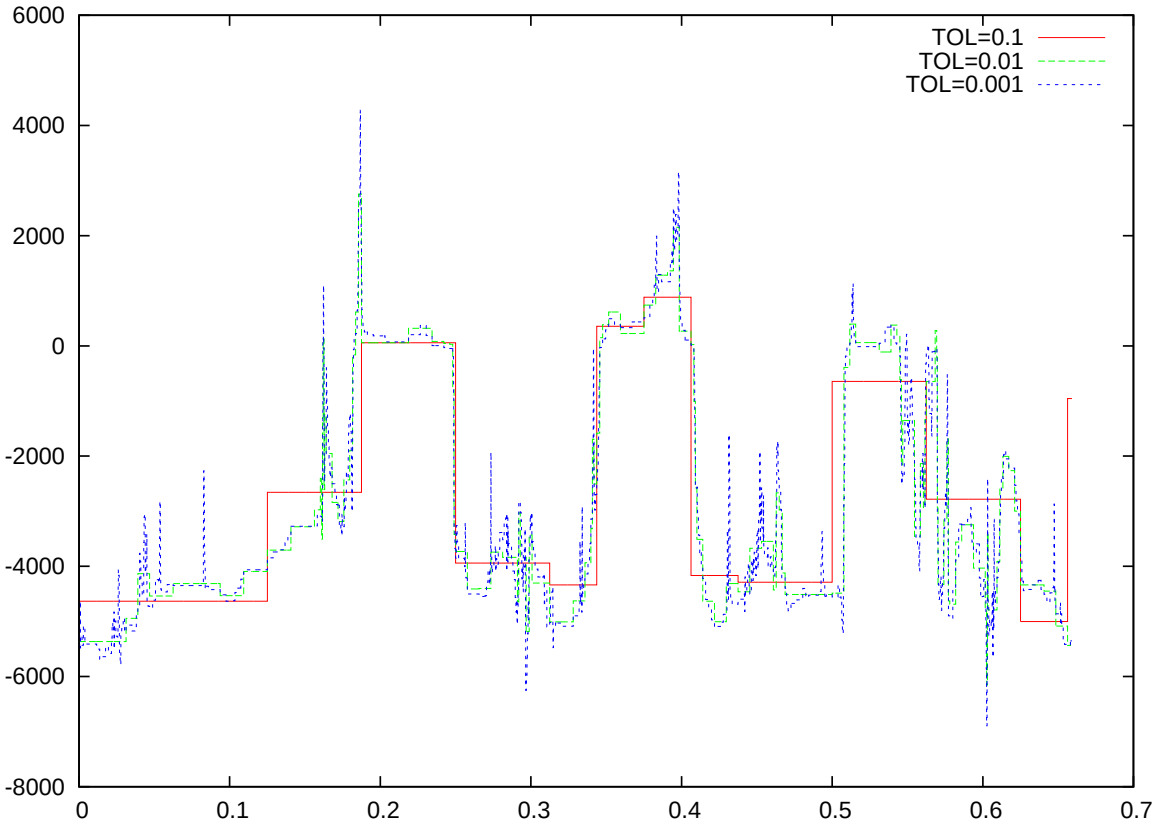

Abb. 13: GEBCO Daten mit verschiedenen Toleranzen.

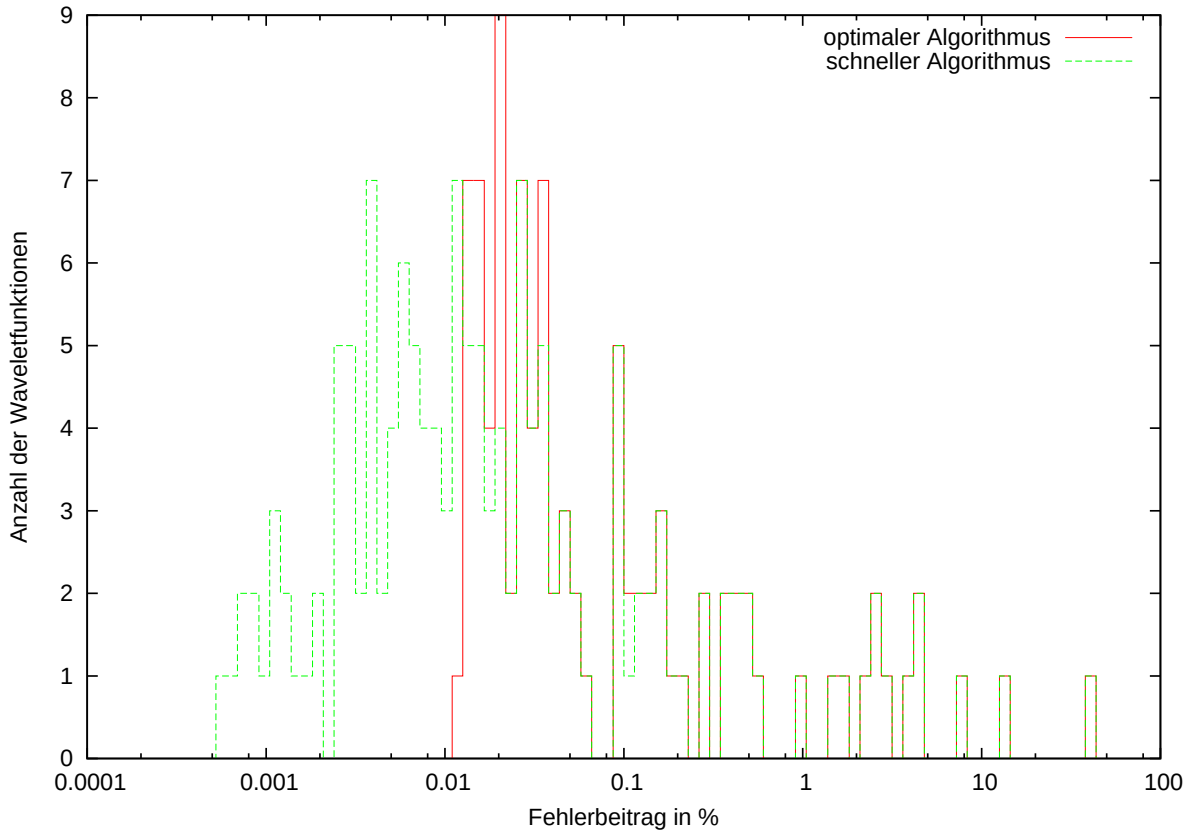

Abb. 14: Im Vergleich: schneller und optimaler Algorithmus im Beuzug auf Fehleranteile

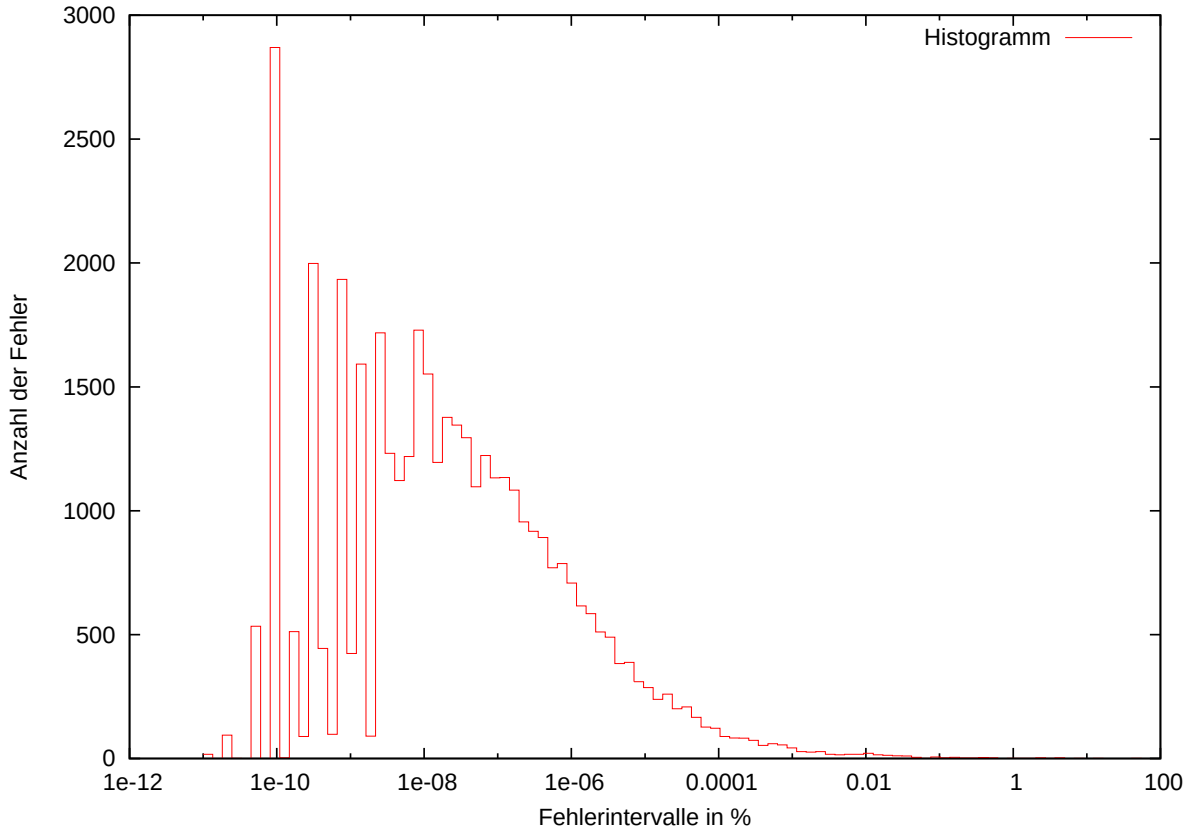

Abb. 15: Das gesamte Fehlerhistogramm

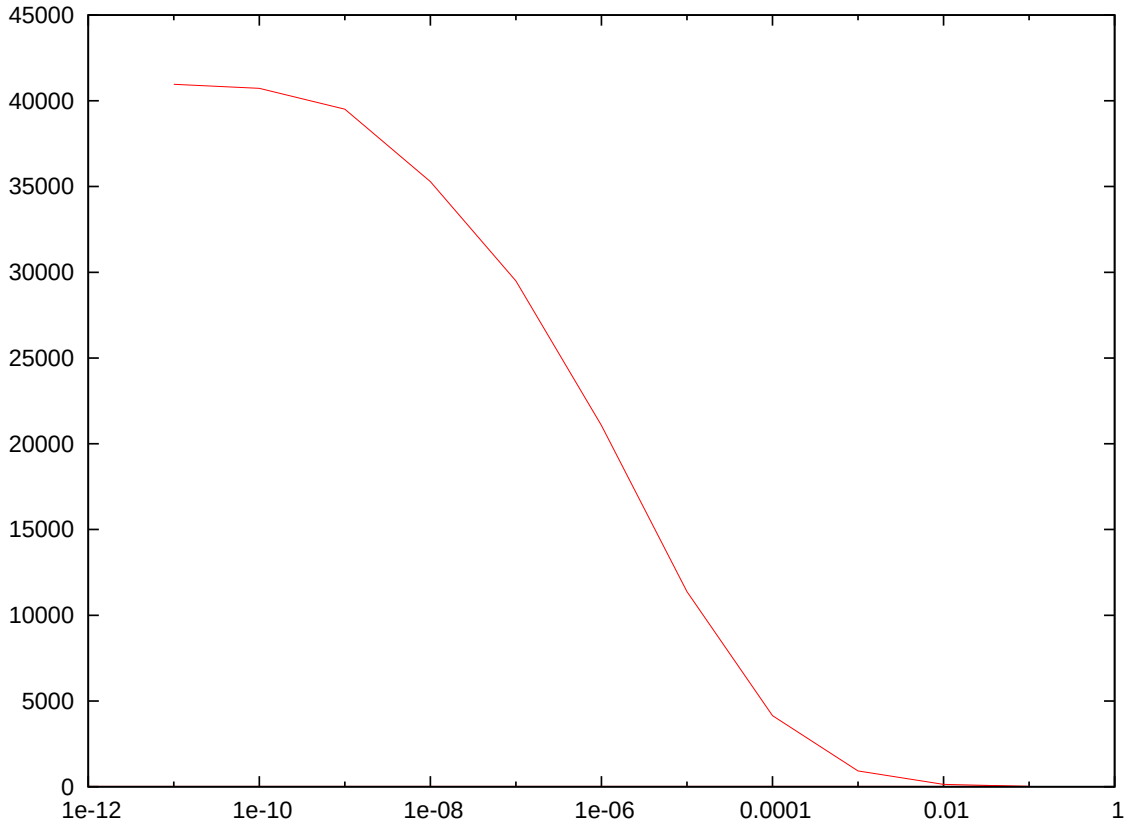

Abb. 16: Kompression von GEBCO Daten. Anzahl verwendeter Basisfunktion in Abhängigkeit der Toleranz (optimaler Algorithmus)

5.3 Sinus-Kurve

Neben Eingaben aus realen Datenquellen, ist es auch möglich automatisch Daten generieren. In diesem Beispiel wurde der Graph der Funktion $sin(x)$ in 1024 äquidistanten Schritten auf dem Intervall $[0, 2\pi]$ generiert und mit einem Toleranzwert von 0.1 mit Wavelet-Funktionen approximiert.
Die mit *A1* betitelten Werte wurden dabei von einem Algorithmus erzeugt, welcher bei der Suche nach dem nächsten Wavelet-Knoten zur Hinzunahme zur Basis allein die Kinder der aktuellen Knoten betrachtet. Der zweite Algorithmus (*A2*) sucht hingegen alle Knoten im Baum ab. Das Schaubild (Abbildung 17) verdeutlicht die Unterschiede zwischen den Algorithmen.

Diskussion. Es ist sehr deutlich zu sehen, dass der Algorithmus 1 (*A1*) genaue Approximationen nur für die linken Hälfte des Graphen vornimmt, die andere Hälfte jedoch komplett vernachlässigt. Zwar schafft es der Algorithmus durch sehr genaue Approximation in der linken Hälfte, unter den Toleranzwert zu gelangen, allerdings ist dieses Ergebnis sehr unbefriedigend, da durch einige wenige Approximationen in der rechten Hälfte ein deutlich größeren Toleranzgewinn mit weniger Basisfunktionen zu erwarten ist.
Eine mögliche Quelle für dieses Verhalten liegt darin, dass in diesem Algorithmus bei der Suche von neuen Knoten für eine Verbesserung der Approximation stets nur Kinder der bereits gewählten Knoten in Betracht gezogen werden. Da das Kriterium für die Wahl eines neuen Knotens (der zu erwartende Fehlerverlust) vereinfacht gesprochen ein Mittelwert ist, können Knoten, de-

ren beiden Kinder betragsmäßig gleichgroße Werte, aber unterschiedliche Vorzeichen besitzen, alleine nur einen sehr geringen Fehlerverlust erzeugen und werden daher von diesem Algorithmus als unwichtig angesehen. Da die Werte der Kinder des Knotens allerdings betragsmäßig sehr groß sein können, ist dies ein ungeschicktes Verhalten.
Der Algorithmus 2 umgeht dies, in dem stets alle möglichen Knoten in Betracht gezogen werden. Diese Herangehensweise erfordert natürlich im Allgemeinen erst einmal etwas mehr Ressourcen, da die Liste der zu betrachtenden Knoten entsprechend größer ist, allerdings wird dieser Nachteil in obigen Situationen wohl mehr als ausgeglichen, da der Algorithmus statt vieler kleiner Approximationen stets die insgesamt Beste wählt, also mit sehr viel weniger Basisfunktionen auskommt.

Fazit. Auch wenn der Algorithmus 2 mehr Ressourcen zum Verwalten der Knotenliste benötigt, wird dies in den meisten Fällen wahrscheinlich durch die sinnvollere Approximation ausgeglichen, der Algorithmus 2 ist also dem Algorithmus 1 vorzuziehen. Weitere mögliche Varianten des Grundalgorithmus ergeben sich, wenn man statt auf nur einer weiteren Ebene (*A1*) oder auf allen Ebenen (*A2*) nach weiteren Knoten zu suchen, dies auf n Ebenen zu tun (*A1* entspräche dann $n = 1$, *A2* $n =$ Tiefe des Baums). Es ist allerdings sehr zu bezweifeln, ob die daraus resultierenden komplexeren Algorithmen den Mehraufwand rechtfertigen, insbesondere da die Liste der Knoten in sehr effizienten Datenstrukturen gespeichert und verwaltet wird.

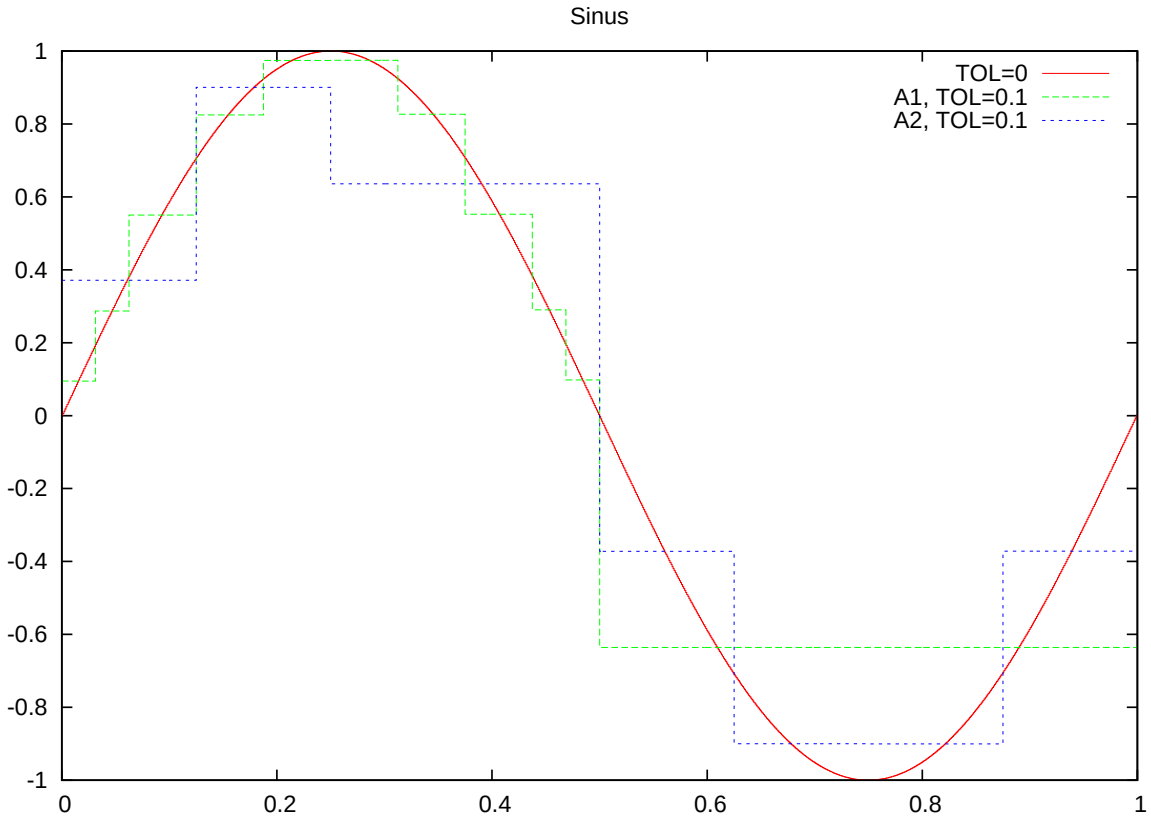

Abb. 17: $sin(x)$ auf dem Intervall $[0, 2\pi]$.

5.4 Laufzeit

Um n Datenpunkte zu speichern, werden n Basisfunktionen benötigt. Der Kompressionsalgorithmus verwendet eine Priority Queue, in die alle Basisfunktionen eingefügt werden. Um n Elemente einzufügen werden $\mathcal{O}(n \cdot \log(n))$ Operationen benötigt. Um dieses zu verifizieren, messen wir die Laufzeit der Kompression von $sin(x)$ im Intervall $[0, 2\pi]$ mit 10^6 bis 10^7 Datenpunkten.

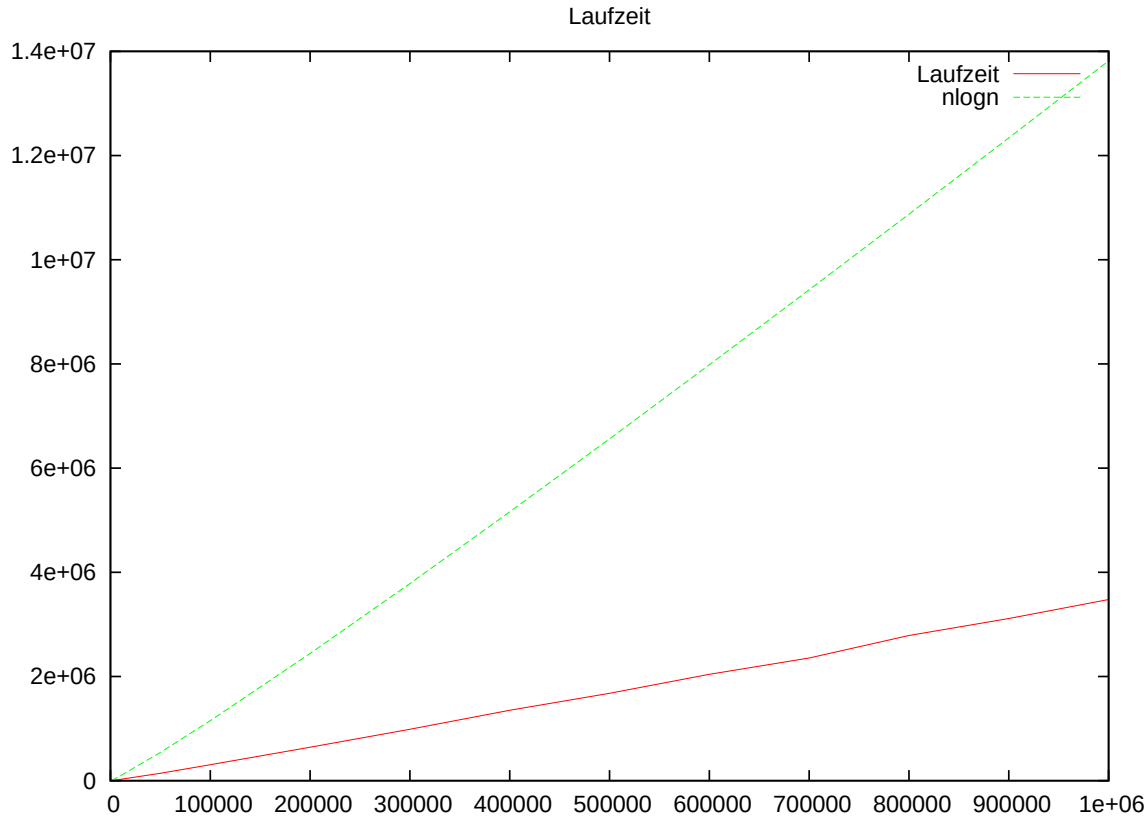

Abb. 18: Laufzeit des Kompressions Algorithmus im Vergleich mit n log n

6 Implementierung

6.1 Programmstruktur

Ganz im Sinne des EVA-Prinzips (Eingabe-Ausgabe-Verarbeitung) ist die Implementierung des Programmes in drei Komponenten geteilt: Eine Komponente zum Einlesen der Daten als Eingabe für den Algorithmus, der Algorithmus selbst und eine dritte Komponente zum Speichern der Ausgabe. Alle drei Komponenten sind nur loose miteinander verbunden, so dass ein leichtes Austauschen und ein einfaches Testen der Einzelkomponenten gewährleistet ist.
Der Algorithmus wurde bereits in den vorherigen Kapiteln ausführlich besprochen, daher wird hier auf eine weitere Darstellung dieses Teils verzichtet. Ein Klassendiagramm, welches alle Elemente des Programmes enthält, befindet sich auf Seite 36.
Im obigen Absatz wurde auf eine genaue Spezifikation der Terme Eingabe und Ausgabe verzichtet: Dies liegt daran, dass die Eingabe und Ausgabe für verschiedene Dateiformate und Verwendungszwecke des Programmes durchaus unterschiedlich sein können. Das Programm ist daher so aufgebaut, das diese Komponenten zur Laufzeit ausgetauscht werden können. Implementiert ist dies über abstrakte Klassen, deren konkreten Unterklassen jeweils spezifische Funktionalitäten zur Verfügung stellen.
Für die Eingabe ist dies beispielsweise die abstrakte Oberklasse `InputReader`, welche unter anderem die Unterklassen `TextInputReader` und `GebcoInputReader` besitzt. Dabei dient der `TextInputReader` dem Einlesen von ASCII Textdateien und der `GebcoInputReader` dem Einlesen von netCDF Dateien aus dem General Bathymetric Chart of the Oceans (kurz GEBCO, `http://www.gebco.net/`).
Die Ausgabe ist analog aufgebaut, mit einer abstrkaten Oberklasse `OutputWriter`. Die Ausgabe kann einerseits die Wavelet-Basis selbst sein, d.h. Koeffizienten und andere Daten der Wavelet-Basis-Funktionen, andererseits aber auch die komprimierten Datensätze in einem einfachen Textformat, so dass sich diese mit Programmen wie beispielsweise gnuplot darstellen lassen. Auch exotischere Ausgabeformate, wie die Ausgabe des Wavelet-Basis-Baums als `.dot`-Datei[1] sind so möglich.
Detailliertere Informationen zu den Ein- und Ausgabemethoden finden sich im Kapitel 'Benutzung des Programmes' und, näher an der Implementierung, in der Doxygen-Dokumentation.
Jede Ein- und Ausgabemethode kann auch weitere Kommandozeilenargumente vom Benutzer fordern und diese genau spezifizieren, so dass diese im Hilfsmodus des Programmes angezeigt werden (siehe Kapitel 'Benutzung des Programmes'). Dies und die automatische Auswahl einer Eingabemethode anhand der Eingabequelle (z.B. durch Überprüfung der Dateiendung) werden von den `Input-` bzw. `OutputManager` Klassen zur Verfügung gestellt.

[1] Siehe beispielsweise `http://en.wikipedia.org/wiki/DOT_language`.

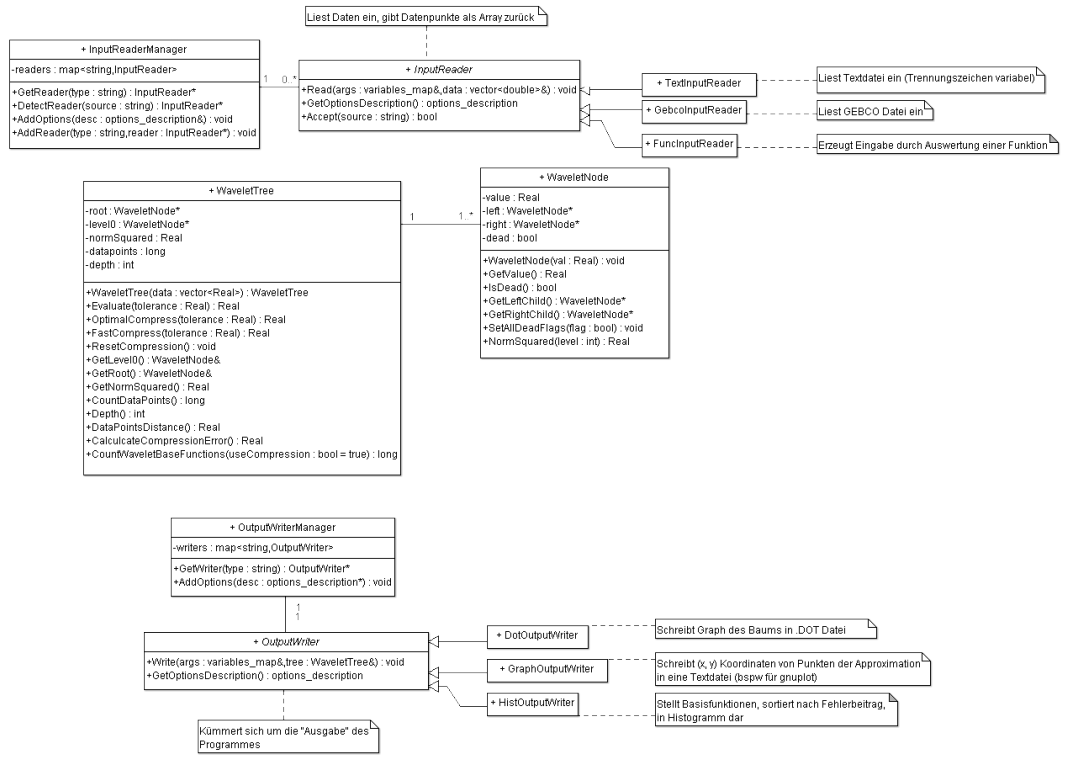

Abb. 19: Das Klassendiagramm der Implementierung.

6.2 Implementierung der Algorithmen

6.2.1 Aufbau des Binärbaums

Modifizierte Wavelet Basis Wie in 3.1 gesehen, lassen sich Linearkombinationen von Wavelet Basis Funktionen als Binärbaum darstellen. Um eine effiziente Auswertung zu gewährleisten, verwenden wir nicht-normierte Funktionen, welche den Wert 1 in der linken Hälfte des Trägers und -1 in der rechten Hälfte annehmen:

$$\varphi_i^l(x) = \begin{cases} 0 & , \quad x \leq \frac{2i}{2^l} \quad oder \quad x > \frac{2i+2}{2^l} \\ 1 & , \quad \frac{2i}{2^l} < x \leq \frac{2i+1}{2^l} \\ -1 & , \quad \frac{2i+1}{2^l} < x \leq \frac{2i+2}{2^l} \end{cases}$$

Wobei wir φ_0^0 mit dem Mother-Wavelet identifizieren.

Aufbau des Binärbaums Sind zwei Datenpunkte $a, b \in \mathbb{R}$ gegeben, kann man statt beide Punkte zu speichern, die halbe Differenz $d := (a - b)/2$ und den Mittelwert $m := (a + b)/2$ speichern. Man erhält daraus die ursprünglichen Werte durch Addition bzw. Subtraktion:
$a = m + d$ und $b = m - d$.
Fasst man die Punkte a und b als Funktion f auf, welche auf dem Intervall $[0, 0.5]$ den Wert a

und auf dem Intervall $(0.5, 1]$ den Wert b annimmt, lässt diese sich schreiben als

$$f = \frac{1}{2}(a-b) \cdot \varphi_0^1 + \frac{1}{2}(a+b) \cdot \varphi_0^0$$

Dies lässt sich fortsetzen für beliebige Datenmengen mit $n = 2^k, k \in \mathbb{N}$ Elementen, wobei jeweils die halbe Differenzen neue Knoten bilden und die Mittelwerte als Datenpunkte der nächstes Ebene aufgefasst werden.

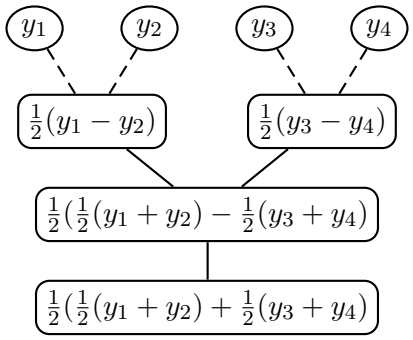

Abb. 20: Binärbaum für vier Datenpunkte. Die Knoten $y_1, \ldots, y_4$ werden nicht gespeichert.

Um so einen Binärbaum aufzubauen geht man wie folgt vor:

1. Initialisiere Array mit den Datenpunkten $y_1, \ldots, y_n$
2. Für je zwei benachbarte Datenpunkte, erzeuge Knoten mit halber Differenz
3. Neue Datenpunkte sind die Mittelwerte der gerade verarbeiteten Datenpunkten
4. Wiederhole diesen Vorgang, bis genau ein Datenpunkt übrig. Dieser ist der Koeffizient des Mother-Wavelets.

Da $n = 2^k$ Datenpunkte gegeben sind, sind in jedem Durchgang eine gerade Anzahl Datenpunkte vorhanden.

Beliebige Datenenmengen Ist eine Menge von beliebig vielen Punkten $y_1, \ldots, y_n$ gegeben, mit in n nicht notwenidig eine Zweierpotenz, gibt es bei diesem Algorithmus nicht in jedem Schritt eine gerade Anzahl Datenpunkte. Desehalb ergänzen wir in diesem Fall jeweils eine Null. Dadurch wird der Baum mit Null-Punkten aufgefüllt, sodass jede Datenmenge auf eine 2-Potenz zurückgeführt wird.
Mit dieser Methode lassen sich beliebige Datenmengen in einer Wavelet Basis darstellen. Die Tiefe des Baumes ist bestimmt durch $d := \lceil log_2(n) \rceil$. Dadurch erhält man den Definitionsbereich $[0, n \cdot 2^{-d}] \subset \mathbb{R}$.

6.2.2 Auswertung

Durch die Wahl der nicht-normierten Wavelet Basis kann der Baum an einer Stelle x ausgewertet werden, indem man den Baum von der Wurzel aus durchläuft und jeweils den Wert des Knotens addiert, falls x in der linken Hälfte des Trägers enthalten ist, oder subtrahiert, falls x in der rechten Hälfte des Trägers liegt. Genauso wählt man für x aus der linken Hälfte den linken Kind Knoten und im anderen Fall den rechten. Dies führt man fort bis kein Knoten mehr vorhanden ist.

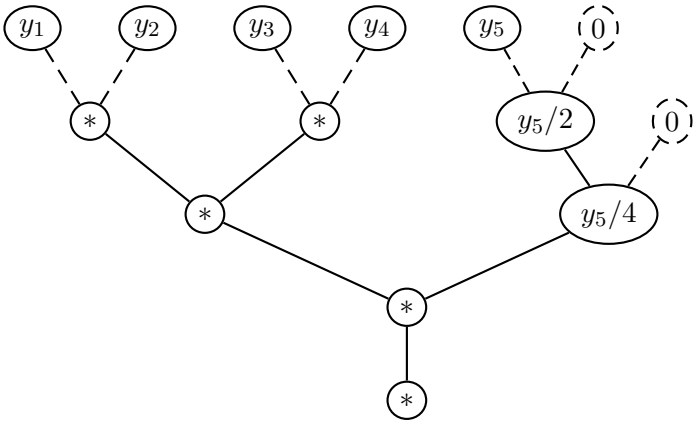

Abb. 21: Binärbaum für fünf Datenpunkte.

6.2.3 Kompression

Rücktransformation in Orthonormalbasis Die verwendete Basis ist wie bereits erwähnt nicht mehr normal. Man erhält eine Orthonormalbasis durch Normierung:

$$\psi_i^l = \frac{\varphi_i^l}{||\varphi_i^l||}$$

Wie bereits gesehen erhält man die Normen $||\varphi_i^l||$ durch $\sqrt{2^{1-l}}$.

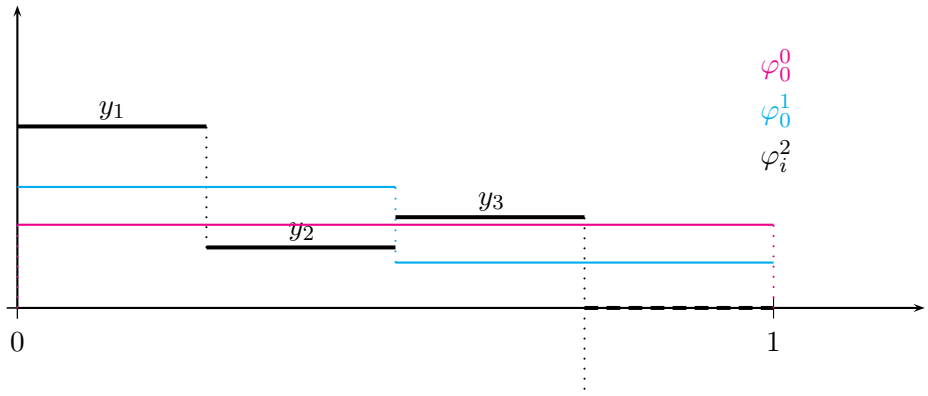

Abb. 22: Zerlegung von drei Datenpunkten in vier Wavelet Basis Funktionen. Def.bereich: $[0, \frac{3}{4}]$

Kompression Die Kompression verläuft analog zum Algorithmus 3.3. Dabei Durchläuft man den Baum angefangen bei der Wurzel und nimmt solange Kinder hinzu, bis die gewünschte Toleranz erreicht ist. Dabei wird in jedem Schritt der Knoten mit dem größten Skalarprodukt zuerst hinzugenommen. Knoten mit Wert Null werden dabei übersprungen. Die Skalarprodukte errechnen sich wie bereits gesehen durch Normierung der im Baum gespeicherten Koeffizienten.

Optimale Kompression Der beschriebene Kompressionsalgorithmus liefert selten die optimale Lösung, ist dafür in den meisten Fällen sehr effizient. Um die optimale Lösung auf Kosten der Laufzeit zu finden, sortiert man alle Wavelet Basisfunktionen nach ihren Skalarprodukten.

Dann nimmt man wie im anderen Algorithmus in absteigender Reihenfolge so lange Funktionen hinzu, bis die gewünschte Toleranz erreicht ist. Dadurch werden immer genau die Funktionen hinzugenommen, die den größten Fehlerbeitrag haben.

6.2.4 Pseudocode

```
1  function BuildTree(y_1, ... ,y_n)
2     data : Queue := <y_1, ... ,y_n>
3     nodes : Queue := <>
4
5     // Algorithmus terminiert, wenn genau ein Datenpunkt
6     // und genau ein Knoten in den Queues
7     while not (data.size = 1 and nodes.size = 1) do
8        // Falls Datenpunkte ungerade, mit Null ergänzen
9        if data.size is odd then
10          data.push( 0 )
11       end if
12
13       // über aktuelle Anzahl Punkte und Knoten iterieren
14       nData := data.size
15       nNodes := nodes.size
16
17       // Datenpunkte in Zweierschritten durchlaufen
18       while nData > 1 do
19          nData := nData - 2
20
21          // nächsten beiden Datenpunkte holen
22          a := data.pop
23          b := data.pop
24
25          // Knoten erzeugen mit Differenz
26          node := Node( (a - b) / 2 )
27
28          // Falls vorhanden, Kinder verlinken
29          if nNodes > 0 then
30             node.left = nodes.pop
31             node.left.parent := node
32             nNodes--
33          end if
34          if nNodes > 0 then
35             node.right = nodes.pop
36             node.right.parent := node
37             nNodes--
38          end if
39
40          // neuen Knoten und Datenpunkt einfügen
41          nodes.push(node)
42          data.push( (a + b) / 2 )
43       end while
44    end while
45
```

```
46      result.root := nodes.pop
47      result.level0 := data.pop
48
49      // quadrate der Datenpunkte summieren
50      result.norm := 0
51      for i := 1 to n do
52          result.norm := result.norm + y_i^2
53      end for
54
55      // normieren
56      depth := ceil(log(n) / log(2))
57      result.norm := result.norm / 2^depth
58  end function
```

```
1   function Evaluate(tree, x)
2       if x < 0 or x > 1 then
3           // x außerhalb Definitionsbereich
4           return 0
5       end if
6
7       result := tree.level0
8       node := tree.root
9
10      level := 1
11      scale := 0
12
13      // Baum durchlaufen, solange lebende Knoten vorhanden
14      while node and not node.dead do
15          if x <= (2*scale + 1) / (2^level) then
16              // x in linker Hälfte des Trägers
17              result := result + node.value
18              node := node.left
19              scale := 2 * scale
20          else
21              // x in rechter Hälfte des Trägers
22              result := result - node.value
23              node := node.right
24              scale := 2 * scale + 1
25          end if
26
27          level++
28      end while
29  end function
```

```
1   // berechnet Normierungsfaktor
2   function Norm(level)
3       if level = 0 then
4           result := 1
5       else
6           result := sqrt( 2^(level-1) )
7       end if
8   end function
```

```
1  function Compress(tree, TOL)
2     // level 0 muss immer enthalten sein
3     S := tree.level0 * Norm(0)
4
5     levels : map<Node, x>   // ordnet Knoten Level zu
6     norms : map<Node, x> // ordnet Knoten Fehlernorm zu
7     nodes : PriorityQueue   // Sortiert nach norms[node]
8
9     // level 1 einfügen
10    nodes.push( tree.root )
11    norms[tree.root] := (Norm(1) * tree.root.value)^2
12    levels[tree.root] := 1
13
14    // alle dead flags auf false setzen
15    tree.ResetDeadFlags
16
17    while S < (1 - TOL) * tree.norm and not nodes.empty do
18       // vordersten Knoten holen (= größtes Skalarprodukt)
19       node := nodes.pop
20       level := levels[node]
21
22       // Kinder in queue einfügen falls vorhanden
23       // GetChilds gibt alle Kinder zurück und überspringt dabei
24       // alle mit value = 0
25       foreach node_ in GetChilds(node) do
26          nodes.push( node_ )
27          levels[node_] := level + 1
28          norms[node_] := (Norm(level + 1) * node_.value)^2
29       end foreach
30
31       // Fehler addieren und Knoten aus allen Containern löschen
32       S := S + norms[node]
33       levels.erase(node)
34       norms.erase(node)
35    end while
36
37    // übrige Knoten als tot markieren
38    foreach node in nodes do
39       node.dead := true
40    end foreach
41 end function
```

6.3 Quellen

[1] Bastian, P., "6.7 Approximation von Funktionen", Universität Heidelberg 2010. Auszug aus Vorlesungsskript.

[2] Schaback, R., & Wendland, H., *Numerische Mathematik*, 5. Auflage, Springer, 2004.

[3] Chart of the Oceans (GEBCO), www.gebco.net.

[4] Deutscher Wetterdienst, www.dwd.de, unter: Klima + Umwelt → Klimadaten → Daten online - frei → Klimadaten Deutschland → Zeitreihen an Stationen → Tageswerte → 10637 Frankfurt/Main Flughafen, abgerufen am 9. Dezember 2010.