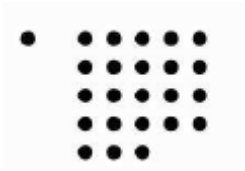


Fachhochschule Köln
University of Applied Sciences Cologne
Campus Gummersbach

Fachbereich Informatik
Studiengang Medieninformatik

Bachelorarbeit

Implementierung eines Echtzeitverfahrens zur Erstellung von Bildmosaiken aus endoskopischen Videosequenzen

von Martin Naderi
Matrikelnummer: 11039223

Erstprüfer: Prof. Dr. Wolfgang Konen
Zweitprüfer: Prof. Dr. Lutz Köhler

Abgabedatum: 03.04.2007

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1

Einleitung

Durch Image Mosaicing[1] ist es möglich, aus mehreren Bildern einer Bildfolge ein einzelnes (Mosaik-) Bild zu erstellen, das den Inhalt aller Einzelbilder (ohne Redundanzen) aufweist. Ziel dieser Arbeit ist es, mittels eines bestehenden Mosaicing-Algorithmus, eine ausbaufähige, objektorientierte Software zu entwickeln, die gegebene Videodaten in Echtzeit verarbeiten, und zur Anzeige bringen kann.

1.1 Motivation

Bei endoskopischen Operationen sind die Einsatzgebiete der digitalen Bildverarbeitung vor allem im Bereich der digitalen Bildoptimierung anzutreffen. So finden sich häufig Arbeiten[2] die durch Bildverbesserungen versuchen die Störeffekte, wie Reflektionen oder Überbelichtungen, von endoskopischem Bildmaterial zu beseitigen.
Eine weitere Problemstellung, der bisher aus Sicht der Bildverarbeitung noch keine[3] Aufmerksamkeit geschenkt wurde, ergibt sich durch das beengte Sichtfeld, das dem Chirurg bei der Durchführung einer endoskopischen Operation zur Verfügung steht. Durch diesen Umstand kann es zu Orientierungsproblemen des Chirurgen kommen.
Aus diesem Grund wäre es nützlich und wünschenswert, eine bildverarbeitende Software zur Verfügung zu haben, die automatisch und in Echtzeit, aus endoskopischen Bildern ein Panorama bildet. Es existieren viele Algorithmen im Bereich des Image Mosaicing, allerdings arbeiten nur wenige von ihnen vollautomatisch[4] bzw. in Echtzeit[5]. Der in dieser Arbeit verwendete Algorithmus[6] basiert auf der Methode von [Kou99], und erfüllt diese Kriterien.

---

1 [Jac03], [KK04]
2 Beispielsweise [FVL04]
3 Mit Ausnahme von [SLH06]
4 U.a.[Kou99], [Sze94], [SLH06], [Rob03]
5 [Kou99], [SLH06], [Rob03]

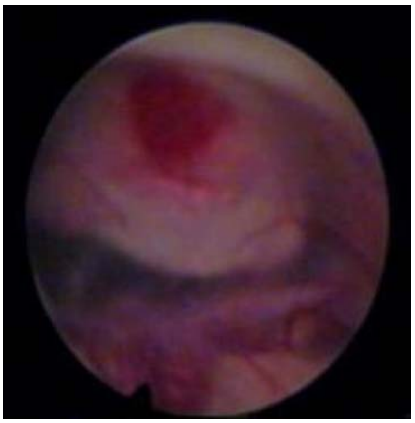


Abbildung 1: Endoskopiebild

1.2 Zielsetzung

Wie bereits erwähnt, ist es das Ziel dieser Arbeit eine ausbaufähige, objektorientierte Software zu entwickeln, die in Echtzeit (d.h. 5 fps (frames per second) und mehr) aus grauen oder farbigen Bildsequenzen ein Panorama bildet. Die Software soll als ImageJ-Plugin[7] umgesetzt werden. Hierbei besteht die Möglichkeit auf eine bestehende, nicht echtzeitfähige Matlab[8]-Implementierung zugreifen zu können. Für die Echtzeitfähigkeit des Plugins muss neben der reinen Realisierung natürlich noch die Optimierung des Codes erfolgen, um die Performance möglichst stark zu steigern.
Die Ergebnisse der Umsetzung, sowie die Erhöhung der Geschwindigkeit sollen ausführlich dokumentiert werden.

1.3 Aufbau der Arbeit

Um dem Leser vorab einen Überblick zu verschaffen, soll im Folgenden kurz auf die Inhalte der einzelnen Kapitel dieser Arbeit eingegangen werden.
Zunächst werden in Kapitel 2 die allgemeinen Grundlagen angesprochen, die für das Verständnis der Arbeit wichtig sind. Das nächste Kapitel widmet sich der genauen Beschreibung des ImageJ-Plugins. Hier wird neben den Klassenbeschreibungen auch auf die Vorgehensweise des Plugins eingegangen. Neben Erläuterungen zur graphischen Oberfläche, befinden sich in dem Kapitel

---

[6] [BKS07]
[7] ImageJ - Java-basiertes Bildverarbeitungsprogramm, das primär in der medizinischen und biologischen Bildverarbeitung eingesetzt wird.
[8] MATLAB - Software die zur Lösung mathematischer Probleme und zur graphischen Darstellung der Ergebnisse
eingesetzt wird. Sie ist für Berechnungen mit Matrizen ausgelegt.

noch die Beschreibungen der wichtigsten Methoden. Das vierte Kapitel enthält zum einen alle Schritte, die zur Optimierung der Performance gemacht wurden, zum anderen eine Betrachtung der Eingabeparameter, und ihre Auswirkungen auf Performance und Genauigkeit. Das letzte Kapitel fasst die Ergebnisse der Arbeit zusammen, und gibt einen Ausblick auf mögliche Erweiterungen des Plugins.
Der Anhang beinhaltet, neben der kompletten Auflistung des Java Codes, die generierte Javadoc Dokumentation.

Kapitel 2

Grundlagen

Für das Verständnis des Themengebiets bedarf es einiger einführender Erläuterungen. Auf diese soll in dem folgenden Kapitel näher eingegangen werden.

2.1 Der optische Fluss

Der optische Fluss ist ein Verfahren aus dem Bereich der Bildverarbeitung, und stellt die sichtbare Änderung von Helligkeit in aufeinander folgenden Bildern dar. Wird das Verfahren angewendet, erhält man als Ergebnis ein Vektorfeld, das für jeden Pixel in einem Bild die Bewegungsrichtung und -weite enthält.
Abbildung 2.1 soll den Unterschied zwischen optischem Flussfeld und Bewegungsfeld verdeutlichen, die nur im idealen Fall übereinstimmen. In Bild a rotiert eine Kugel, die von einer feststehenden Lichtquelle beleuchtet wird. Durch die Rotation der Kugel wird ein Bewegungsfeld erzeugt. Da sich aber durch diese Anordnung keine Änderung der Helligkeit ergibt, ist der optische Fluss gleich Null. In Bild b rotiert die Kugel nicht, allerdings wird sie nun von einer bewegten Lichtquelle angestrahlt. Durch diesen Sachverhalt ergibt sich wegen der Helligkeitsänderung ein optischer Fluss, und ein Bewegungsfeld das nun gleich Null ist. Abbildung 2.2 hingegen zeigt ein Objekt, dessen Bewegungsfeld und optischer Fluss übereinstimmen.

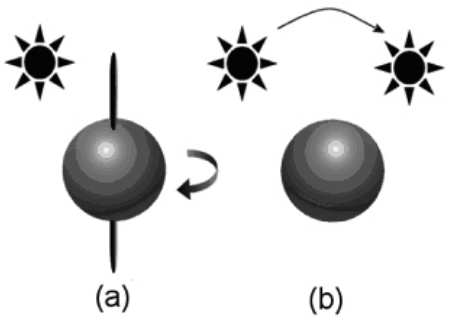


Abbildung 2.1: Bewegungsfeld und optischer Fluss

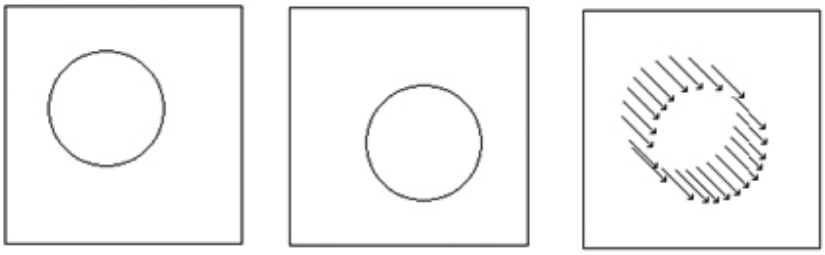


Abbildung 2.2: Optischer Fluss und Bewegungsfeld (2)

Normalerweise existieren zur Bestimmung des optischen Flusses zwei Voraussetzungen. Diese werden in der Literatur (siehe [HS81]) als *Optical Flow Constraints* bezeichnet (optische Fluss-Bedingungen).
Anstatt des Regularisierers aus der Literatur (die zweite optische Fluss-Bedingung) kommt, in dem hier verwendeten Verfahren, allerdings ein regularisierendes Geschwindigkeitsfeld zur Anwendung. Als Geschwindigkeitsfeld wird, bei der Berechnung der Compensated Motion (in Kapitel 2.4), die affine Transformation verwendet.

Der ersten optischen Fluss-Bedingung (der sog. Dataterm) liegt die Annahme zugrunde, dass sich zwischen zwei konsekutiven Frames die Helligkeit (der einzelnen Grauwerte) nicht verändert, sondern lediglich verschiebt. Als weitere Annahme wird vorausgesetzt, dass sich keine richtungsabhängigen Beleuchtungseffekte in der Szene befinden.
Betrachten wir einen Bildpunkt in einem zweidimensionalen Bild (Abbildung 2.3), der sich zum Zeitpunkt *t* an der Stelle *(x,y)* befindet. Wandert dieser im nächsten Zeitschritt *t+dt* an eine andere Stelle *(x+u,y+v)*, so ergibt sich eine Verschiebung um den Bewegungsvektor *(u,v)*.
Die schon erwähnte erste Bedingung zur Berechnung des optischen Fluss-Vektors wird durch folgende Gleichung dargestellt, wobei die Funktion *I(x,y,t)* die Intensität der Lichtstärke des Punktes *(x,y)* zum Zeitpunkt *t* angibt:

$$I\left(x+dx, y+dy, t+dt\right) = I\left(x, y, t\right) \tag{2.1}$$

Durch Taylor-Entwicklung um den Punkt *(x,y,t)* erhalten wir:

$$I\left(x+dx, y+dy, t+dt\right) = I\left(x, y, t\right) + dx\frac{\partial I}{\partial x} + dy\frac{\partial I}{dy} + dt\frac{\partial I}{dt} + \varepsilon \tag{2.2}$$

Nach einer Division durch *dt*, der Vernachlässigung des Restgliedes ε und Kürzen von *I(x,y,t)* erhält man:

$$\frac{\partial I dx}{\partial x dt} + \frac{\partial I dy}{\partial y dt} + \frac{\partial I}{\partial t} \approx 0 \tag{2.3}$$

Ersetzt man jetzt die partiellen Ableitungen $\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} und \frac{\partial I}{\partial t}$ durch eine andere Schreibweise, nämlich I_x, I_y, I_t, und die entstandenen Terme $\frac{dx}{dt} und \frac{dy}{dt}$ durch die Vektorkomponenten des optischen Flusses $u = \frac{dx}{dt}$ und $v = \frac{dy}{dt}$, so erhält man die erste optische Fluss-Bedingung:

$$I_x u + I_y v + I_t = 0 \tag{2.4}$$

Eine ausführliche Herleitung der Gleichung ist in [Chr04] zu finden.

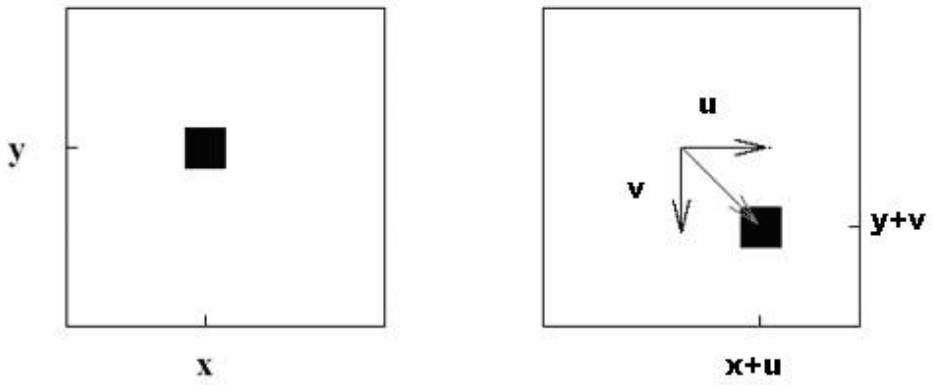


Abbildung 2.3:Beispiel eines Bewegungsvektors

2.2 Die bilineare Interpolation

Als Interpolation wird ein Vorgang bezeichnet, Werte von diskreten Funktionen für Positionen abseits ihrer Stützstellen zu schätzen.
Die Notwendigkeit der Interpolation ergibt sich, bei geometrischen Bildoperationen in der Bildverarbeitung dadurch, dass Gitterpunkte häufig nicht auf Gitterpunkten, sondern auf den Räumen zwischen den Gitterpunkten abgebildet werden.
Es existieren im Allgemeinen drei Ansätze, die für das Resampling der Grauwerte in Frage kommen, nämlich nächste Nachbarschaft, bilineare und bikubische Interpolation.
Die bilineare Interpolation wurde als Verfahren der Subpixel-Approximation gewählt, da sie den besten Kompromiss aus Rechenaufwand und Bildqualität bietet.

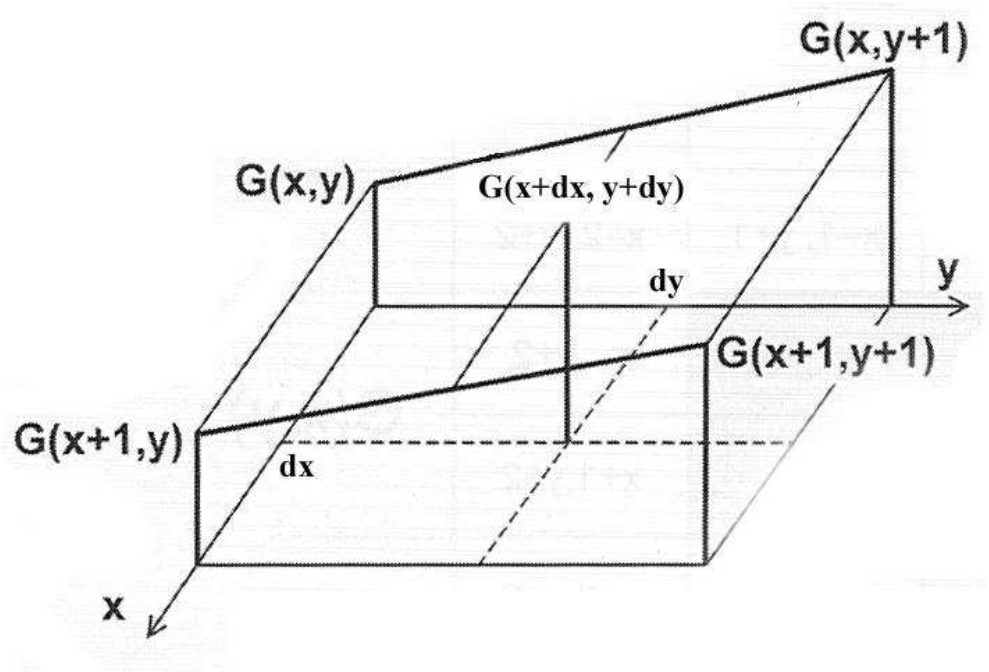


Abbildung 2.4: Bilineare Interpolation des Grauwertes am Zwischenbildpunkt *(x+dx, y+dy)*.
Quelle: [Neu05]

In Abbildung 2.4 ist zu sehen wie die bilineare Interpolation arbeitet. Der Wert an Zwischenbildpunkt *G(x+dx,y+dy)* wird mit Hilfe der vier Nachbarbildpunkte errechnet:

$$G(x+dx, y+dy) = (1-dy) \cdot G(x+dx, y) + dy \cdot G(x+dx, y+1) \qquad (2.5)$$

mit:

$$G(x+dx, y) = (1-dx)\cdot G(x, y) + dx \cdot G(x+1, y)$$

und

$$G(x+dx, y+1) = (1-dx)\cdot G(x, y+1) + dx \cdot G(x+1, y+1)$$

2.3 Die affine Abbildung

Eine affine Abbildung ist eine lineare Koordinatentransformation. Sie beschreibt vollständig die Bewegung einer ebenen Fläche, unter orthographischer Projektion auf die Bildfläche. Die elementaren Transformationen, die durch die affine Abbildung umfasst werden, sind neben Translation, Rotation und Dilatation auch die Stauchung und Scherung eines Objektes. In Abbildung 2.5 werden alle anwendbaren Transformationen dargestellt.

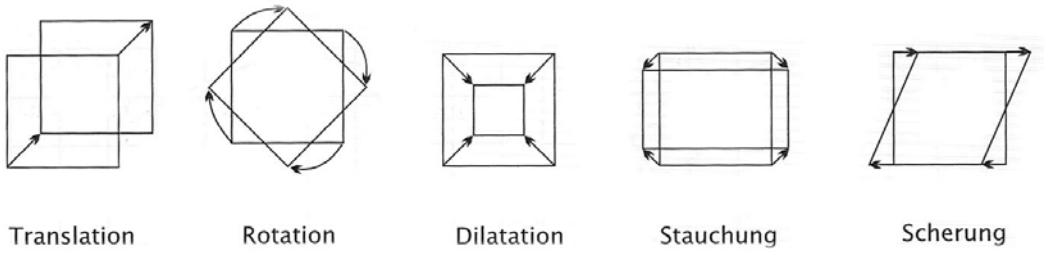


Abbildung 2.5: Elementare geometrische Transformationen. Quelle:[Jäh05]

Bei Verwendung homogener[9] Koordinaten, kann die affine Abbildung durch eine einzige Matrizenmultiplikation beschrieben werden:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{2.6}$$

Die affine Abbildung besitzt insgesamt sechs Freiheitsgrade a_{11} … a_{23}. Vier dieser Freiheitsgrade (a_{11}, a_{12}, a_{21}, a_{22}) definieren zusammen die Rotation, Dilatation, Stauchung und Scherung. Die zwei anderen (a_{13} und a_{23}) sind für die Translation zuständig.

[9] [BB06, S. 365 f.] oder [Jäh05, S. 224 f.]

Durch die affine Abbildung werden Geraden in Geraden, Dreiecke in Dreiecke und Rechtecke in Parallelogramme überführt. Daher wird sie auch als Dreipunkt-Abbildung bezeichnet.
Verformungen wie die Abbildung eines Rechtecks auf ein beliebiges Viereck sind keine affinen Abbildungen.

2.4 Die Methode nach Kourogi99

Das Verfahren von Kourogi [10] baut auf der Idee des optischen Flusses auf, um aus Bildern einer Videosequenz komplette Panoramen bilden zu können. Der Ausgangspunkt der Überlegung ist die (umgestellte) Gleichung (2.7) des optischen Flusses:

$$I(x+u, y+v, t) - I(x, y, t-1) = 0 \tag{2.7}$$

Die beiden unbekannten *u* und *v*, die den Verschiebungsvektor bilden, können durch diese Gleichung alleine allerdings nicht bestimmt werden. Deshalb bedient sich Kourogi sogenannter Pseudo Motion Vektoren, die eine grobe Schätzung des optischen Flusses an jedem Bildpunkt darstellen. Um die Pseudo Motion Vektoren berechnen zu können, wird bei der Berechnung von *u* der *v*-Term zu Null gesetzt. Nach dem gleichen Schema gestaltet sich die Berechnung von *v*. Somit ergeben sich die Gleichungen zur Berechnung der Vektoren:

$$\begin{gathered} u_p = -I_t / I_x \\ \text{und} \\ v_p = -I_t / I_y \end{gathered} \tag{2.8}$$

I_x, I_y und I_t, die partiellen Ableitungen der Lichtstärke *I* nach *x*, *y* und *t*, werden numerisch approximiert durch:

$$\begin{aligned} I_t &= I(x, y, t) - I(x, y, t-1) \\ I_x &= \big(I(x+1, y, t-1) - I(x-1, y, t-1)\big)/2 \\ I_y &= \big(I(x, y+1, t-1) - I(x, y-1, t-1)\big)/2 \end{aligned} \tag{2.9}$$

[10] siehe [Kou99], [Kon06]

Hier ist zu beachten, dass die Vektoren nur berechnet werden können, wenn I_x und I_y nicht Null sind. Da bei der Berechnung der Vektoren oftmals unsinnige Ergebnisse entstehen, wird folgender Test durchgeführt:

$$\left| I\left(x+u_p, y+v_p, t\right) - I\left(x, y, t-1\right) \right| < T \tag{2.10}$$

T gibt eine beliebige Grauwertschwelle an, wobei Kourogi hier einen Wert von 5 gewählt hat. Alle Vektoren, die diesen Test bestehen, können nun zur weiteren Berechnung verwendet werden.
Durch die Pseudo Motion können nur Verschiebungen von etwa einem Pixel geschätzt werden. Bei allen größeren Verschiebungen wird das Ergebnis sehr ungenau, was folgende Tabelle verdeutlichen soll:

T=5	wahre Translation		Pseudo Motion		
	u	v	$\bar{u}_p$	$\bar{v}_p$	% accept
	-1.5	-2.5	-0.6 ± 4.7	-1.4 ± 5.0	30%
	-5.2	-3.5	-1.5 ± 6.8	-0.8 ± 8.5	21%
	-10.5	7.6	-1.5 ± 10.0	2.2 ± 12.3	14%

Tabelle 2.1: Ergebnisse der Pseudo Motion bei größeren Translationen. Quelle:[Kon06]

Der Grund für die steigende Ungenauigkeit, ist das häufigere Auftreten von Sprungstellen (Diskontinuitäten) bei größeren Verschiebungsvektoren. Die Pseudo Motion ist nämlich nur dann genau, wenn in einem Bild eine rein lineare Grauwertrampe (im eindimensionalen Fall z.B.: 130, 132, 134, 136…) am Beobachtungsort (x,y) vorbeiwandert. Ist dieser Fall nicht gegeben, kann der lokale Gradient diese Änderung im Zeitschritt von t bis $t+dt$ nicht kennen.

Um auch größere Verschiebungen erkennen zu können, wird von Kourogi die Anwendung der Compensated Motion vorgeschlagen.

Um diese Methode verwenden zu können, wird eine (grobe) Schätzung $\begin{pmatrix} u_c \\ v_c \end{pmatrix}$ des Verschiebungsvektorfeldes benötigt, und an allen Stellen eingesetzt. Wenn nun in einem zweiten Schritt der verbleibende Rest $\begin{pmatrix} u_r \\ v_r \end{pmatrix}$ zum tatsächlichen Vektor $\begin{pmatrix} u \\ v \end{pmatrix}$ berechnet wird, muss jetzt nur noch eine kürzere Strecke geschätzt werden.

Um eine grobe Schätzung für die Compensated Motion zu bekommen, wird nun angenommen, dass das komplette Verschiebungsfeld z.B. aus der Klasse der

affinen Transformationen stammt. Wie weiter oben erwähnt wurde, handelt es sich hier bei dem affinen Geschwindigkeitsfeld um einen Ersatz des in der Literatur verwendeten Regularisierers. Somit kann aus der Pseudo Motion dann eine globale Compensated Motion geschätzt werden.

$$\begin{pmatrix} u_c \\ v_c \end{pmatrix} = \begin{pmatrix} a_1 x + a_2 y + a_3 \\ a_4 x + a_5 y + a_6 \end{pmatrix} \tag{2.11}$$

Da jetzt zum Zeitpunkt *t* nur die geschätzte Bewegung kompensiert werden kann, wird die Gleichung (2.12) für die partielle Ableitung nach der Zeit entsprechend verändert:

$$I_t^{(c)} = I\left(x + u_c, y + v_c, t\right) - I\left(x, y, t-1\right) \tag{2.12}$$

Die Gleichungen (2.13) für die Berechnung der Pseudo Motion werden ebenfalls abgeändert:

$$\begin{gathered} u_p = \left(-I_t^{(c)} / I_x\right) + u_c \\ \text{und} \\ v_p = \left(-I_t^{(c)} / I_y\right) + v_c \end{gathered} \tag{2.13}$$

Die neuen Vektoren müssen wieder nach Gleichung (2.10) getestet werden.
Die abwechselnden Berechnungen von (u_c,v_c) und (u_p,v_p) müssen nun iteriert werden, und führen zu einer besseren Schätzung des Verschiebungsvektorfeldes:

T=5		wahre Translation		Pseudo Motion		
		u	v	$\bar{u}_p$	$\bar{v}_p$	% accept
iter	1	-10.5	7.6	-1.5 ± 10.0	2.2 ± 12.3	14%
	2	-10.5	7.6	-3.1 ± 8.6	3.8 ± 10.3	16%
	5	-10.5	7.6	-7.4 ± 5.3	7.0 ± 6.8	23%
	10	-10.5	7.6	-10.47 ± 1.2	7.61 ± 1.27	55%

Tabelle 2.2: Ergebnisse der Pseudo Motion unter Zuhilfenahme der Compensated Motion. Quelle:[Kon06]

Um nun für das Pseudo Motion Feld (u_p,v_p) die affinen Parameter (Gleichungen (2.14)), die das Feld bestmöglich beschreiben sollen, zu errechnen, muss das

$$a_1 x_i + a_2 y_i + a_3 = u_{pi}$$
$$\text{und} \tag{2.14}$$
$$a_4 x_i + a_5 y_i + a_6 = v_{pi}$$

folgende Gleichungssystem (2.15) nach der Least-Square[11] Methode gelöst werden. Es ist zu beachten, dass hier nur mit den Bildpunkten gerechnet wird, die den Test in Gleichung (2.10) passiert haben.

$$\left(A \cdot a - b\right)^2 = Min$$
$$\text{wobei}$$
$$A = \begin{pmatrix} x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots \\ x_N & y_N & 1 \end{pmatrix}, \quad a = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \quad und \quad b = \begin{pmatrix} u_{p1} \\ \vdots \\ u_{pN} \end{pmatrix} \tag{2.15}$$

Um nicht, wie in dieser Methode, mit so großen Matrizen rechnen zu müssen (nicht performant), bietet sich ein weiteres Vorgehen an. Hierzu müssen in $Aa=b$ beide Seiten von links mit A^T durchmultipliziert werden:

$$\left(M \cdot a - c\right)^2 = Min$$
$$\text{wobei}$$
$$M = A^T A = \begin{pmatrix} \sum x_i x_i & \sum x_i y_i & \sum x_i \\ \sum x_i y_i & \sum y_i y_i & \sum y_i \\ \sum x_i & \sum y_i & \sum 1 \end{pmatrix} \; und \; c = A^T b = \begin{pmatrix} \sum x_i u_{pi} \\ \sum y_i u_{pi} \\ \sum u_{pi} \end{pmatrix} \tag{2.16}$$

[11] Die Methode der kleinsten Quadrate ist das mathematische Standardverfahren zur Ausgleichungsrechnung (siehe [Jäh05])

Kapitel 3

Das ImageJ-Plugin

In dem nachfolgenden Kapitel wird das Plugin schrittweise beschrieben. Nach einer Übersicht über die vorhandenen Klassen, wird anhand eines Flussdiagramms der Ablauf der einzelnen Bearbeitungsschritte verdeutlicht. In den darauf folgenden Methodenbeschreibungen wird sehr genau auf die Algorithmen eingegangen, um die exakte Vorgehensweise der Methoden erkennen zu können. Schließlich endet das Kapitel mit einer Beschreibung der grafischen Oberfläche des Plugins.

3.1 ImageJ

Bei dem von Wayne Rasband am U.S. National Institutes of Health (NIH) entwickelten ImageJ (Nachfolgeprojekt des älteren NIH-Image), handelt es sich um eine Java-basierte Software zur Bildverarbeitung. Häufigen Einsatz findet ImageJ in der medizinischen und biologischen Bildverarbeitung, sowie in der wissenschaftlichen Bildanalyse. Die Software bietet bereits fertige Werkzeuge zur Darstellung und Manipulation von Bildern, andererseits lässt es sich wegen seines Plugin-Konzepts (siehe [Bai03], [BB06]) sehr einfach durch eigene, oder bereits existierende, Softwarekomponenten erweitern.

3.2 Funktionalität der graphischen Oberfläche

Die graphische Oberfläche des Plugins bietet, neben der Anzeige von Bilddaten, einige Einstellungsmöglichkeiten für den Benutzer, auf die hier kurz eingegangen werden soll.
Wie in Abbildung 3.1 zu sehen ist, kann durch die Combobox (1) eine von vier vordefinierten Bildfolgen ausgewählt werden. Der Button (2) startet den Bearbeitungsvorgang. Über die übrigen Comboboxen (3) ist es möglich die Eingabeparameter vorab zu verändern, um somit Einfluss auf die späteren Berechnungen zu nehmen.

Wird die vierte Bildsequenz (*PDVD_*) gewählt, kann über Checkbox 4 auf die Ausgabe eines farbigen Panoramas umgeschaltet werden.
Zusätzlich können zur Ausgabe des Panoramas, weitere Bildinformationen angezeigt werden (Checkboxen 5 und 6), wobei hier jeweils nur eine Auswahl zur gleichen Zeit möglich ist. Das Differenzbild wird außerdem nur angezeigt, wenn die Anzeige der farbigen Ausgabe nicht aktiviert ist.
Wird Checkbox 7 aktiviert, vergrößert sich das Ausgabebild automatisch, wenn das Panorama in die Nähe eines Bildrandes kommt.
Zuletzt sorgt die Betätigung von Checkbox 8, für eine Verzögerung von 500ms (Millisekunden) bei der Ausgabe, um den Vorgang der Panoramaerweiterung besser verfolgen zu können.

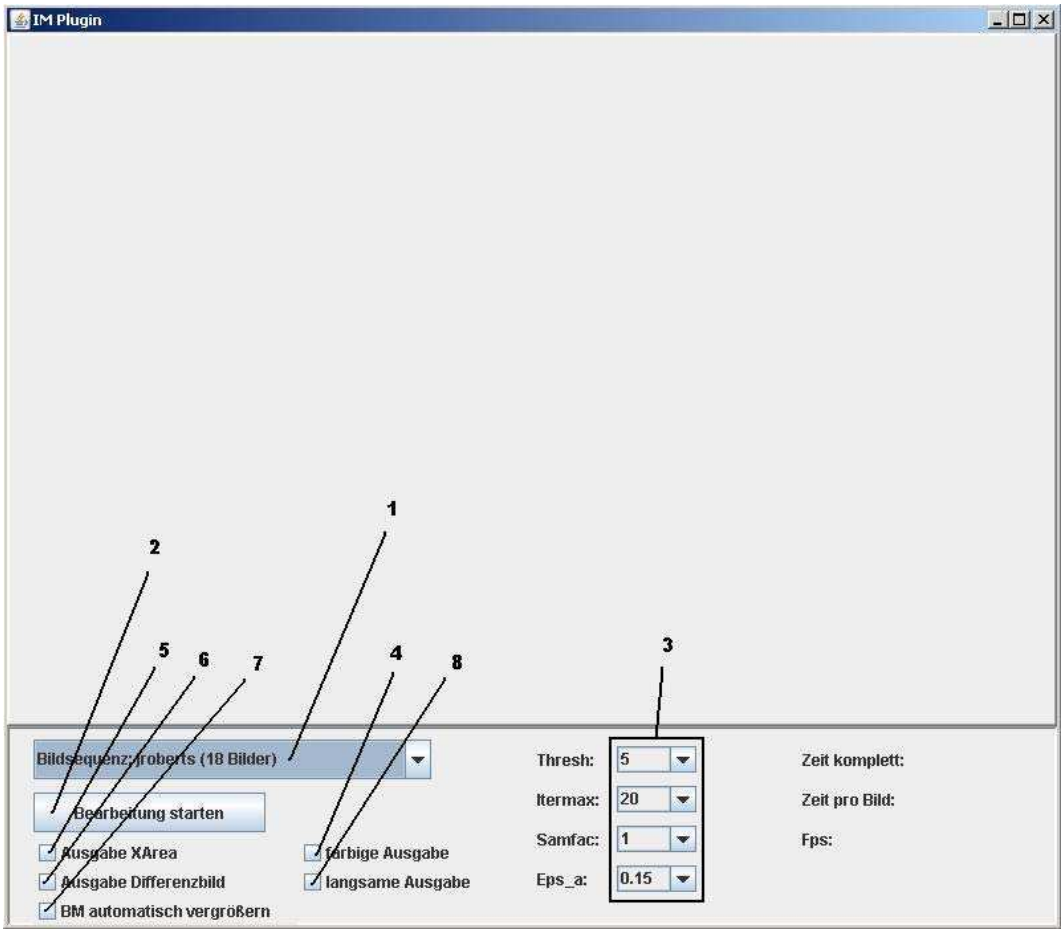


Abbildung 3.1: Die graphische Oberfläche des Plugins

3.3 Die Klassen des Plugins

Bei der Festlegung des Klassendesigns, wurde darauf geachtet das MVC Prinzip umzusetzen. Hierdurch kann ein flexibles Klassendesign realisiert werden, um eine spätere Erweiterung des Plugins zu erleichtern, und die Wiederverwendbarkeit der einzelnen Klassen zu ermöglichen.
Das MVC Prinzip unterteilt die einzelnen Klassen in drei verschiedene Einheiten.
Das Modell (model) enthält alle Daten die zur Darstellung benötigt werden. Es kennt weder die Präsentation (view) noch die Steuerung (controller).
Die Präsentation ist für die Visualisierung der Daten zuständig. Des Weiteren leitet sie die Benutzereingaben an die Steuerung weiter.
Die Aufgabe der Steuerung ist es, die Zusammenarbeit von Modell und Präsentation zu koordinieren. Sie verarbeitet die Benutzereingaben, und reagiert entsprechend auf sie.
Im Folgenden werden die Klassen in das MVC Muster eingeordnet. Gleichzeitig werden ihre jeweiligen Aufgaben geklärt, und anschließend in einem Klassendiagramm (Abbildung 3.2) dargestellt.

- GUI (view):
  Der Konstruktor dieser Klasse wird durch *Control* aufgerufen, und erzeugt die Oberfläche des Plugins. Neben weiteren Methoden (beispielsweise zum Anzeigen des Panoramas), ist in dieser Klasse eine *ActionListene*-Klasse in Form einer inneren Klasse vorhanden, die auf die Eingaben des Benutzers reagiert, und entsprechend Methoden aus *Control* aufrufen kann.

- ImgCanvas (view):
  Diese Klasse wurde von *JPanel* abgeleitet, und um einige Methoden erweitert. Ihre Aufgabe ist es, die durch dieses Plugin erzeugten Bilddaten entsprechend darzustellen (nicht im Klassendiagramm enthalten).

- Control (controller):
  Sinn und Zweck dieser Klasse ist es, die anderen Klassen zu steuern, und den Ablauf der einzelnen Schritte zu koordinieren. Auch die Interaktion mit dem Benutzer findet nur über diese Klasse statt (z.B.: Erfassen der Benutzereingaben und Weiterleitung an die entsprechenden Klassen).

- Var (model):
  Die Klasse Var enthält alle Parameter zur Manipulation der Berechnungen in der Klasse *Proc*. In der Klasse *Control* wird aus ihr ein Objekt instanziert, und mit den jeweiligen Werten belegt.

- PreProc (model):
  Ihre beiden Methoden bearbeiten rechteckige und nicht rechteckige Masken. Als Ergebnis erhält man zwei Maskenarrays, die u.a. von den Methoden *Proc.motion* und *Extend.ext* verwendet werden.

- Proc (model):
  Diese Klasse stellt Methoden zur Verfügung, mit deren Hilfe die affinen Parameter der jeweiligen Bildpaare errechnen werden können. Der Aufruf der public-Methode *Proc.motion* erfolgt aus der Klasse *Control*.

- PanoAffine (model):
  Objekte dieser Klasse beinhalten alle Variablen und Felder, die zur Erzeugung eines Panoramas nötig sind. Das Panorama-Objekt wird in der Klasse *Control* instanziert

- Extend (model):
  Durch diese Klasse kann aus den jeweiligen Bildern ein Panorama zusammengesetzt werden. Auch eine Methode zur automatischen Vergrößerung des Panoramas ist hier zu finden.

- Util (model):
  Eine Hilfsklasse, deren Methoden von den Klassen *Control*, *PreProc*, *Proc*, *Extend* und *PanoAffine* verwendet werden. Die wichtigste Methode (*getXmatYmat*) dient zur Erzeugung von Koordinatenmatrizen, welche an die jeweiligen Bildgrößen angepasst sind.
  Durch Aufruf von *bb* kann eine Bounding Box, um eine gegebene Bildmaske, errechnet werden. Neben der Methode zur Matrizen-multiplikation (*arrTimes*), kann durch Aufruf von *sum* die Anzahl der Pixel in einer Bildmaske ermittelt werden.

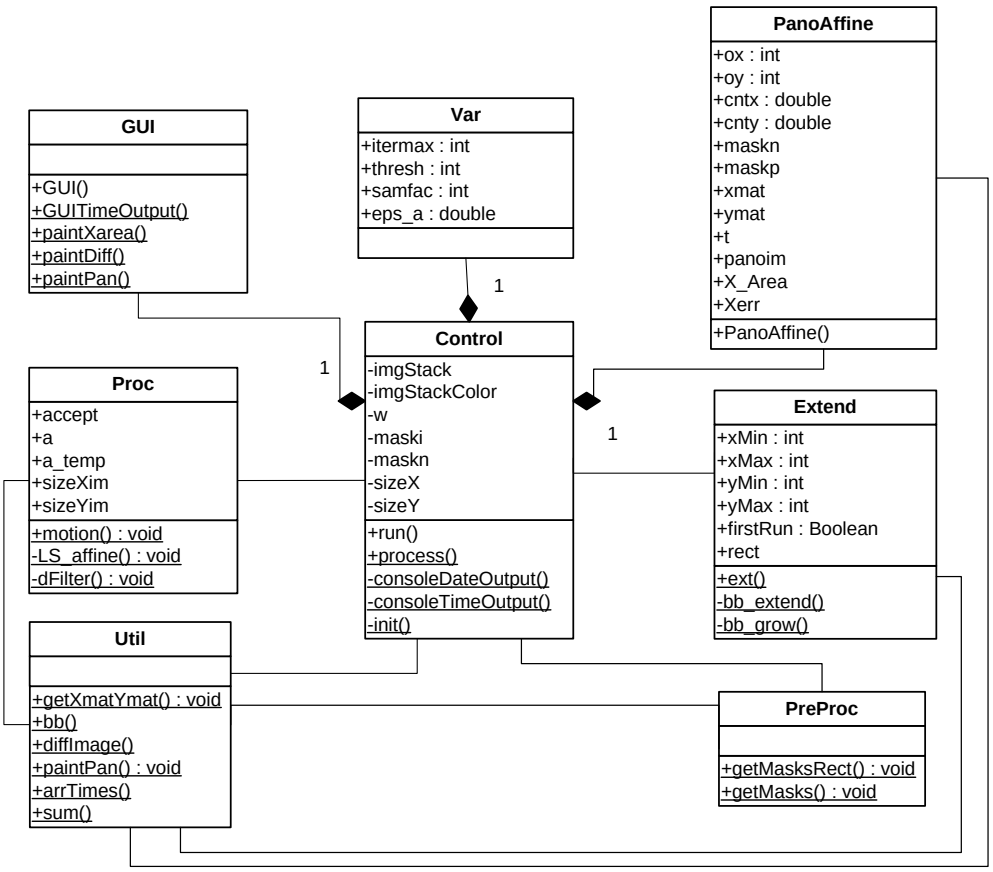


Abbildung 3.2: Klassendiagramm

3.4 Vorgehensweise des Plugins

Um ein Verständnis für die Vorgehensweise des ImageJ-Plugins zu bekommen, soll an dieser Stelle der allgemeine Ablauf genauer erläutert werden, wobei auf die iterativ aufgerufenen Methoden erst im nächsten Kapitel eingegangen wird. Das Flussdiagramm in Abbildung 3.3 enthält alle Stationen des Programms, nach dem Start durch den Benutzer.

- Zu Beginn werden die im Vorfeld eingegebenen Daten des Benutzers erfasst, und in dem dazugehörigen Parameterobjekt gespeichert. Diese Parameter haben großen Einfluss auf Geschwindigkeit und Genauigkeit der späteren Berechnungen.

- Zu den Eingaben des Benutzers, gehört auch die Auswahl der Bildfolge zur Bildung des Panoramas. Diese wird nun in einen, auf die Größe der Bildfolge zugeschnittenen, Image-Stack verschoben. Die einzelnen Bilder werden zuvor, falls es sich bei ihnen um Farbbilder handelt, in Grauwertbilder umgerechnet. Nur Bilder aus diesem Stack werden zur Berechnung der Bewegungsfelder verwendet.

- Hat der Benutzer als gewünschte Ausgabeart ein farbiges Panorama gewählt, wird nun zusätzlich zum Grauwert-Stack ein extra Farb-Stack angelegt, in dem die farbige Bildfolge abgelegt wird. Dieser Stack wird nun zur Panoramaerzeugung verwendet.

- Je nach Ausgabeart erfolgt jetzt die Erzeugung des Panoramaobjekts durch ein Grau- bzw. Farbbild.

- Vor der Berechnung des gesamten Image-Stacks, müssen nun noch diverse Variablen und Felder der Klasse *Proc* angelegt und initialisiert werden.

- Nach dem obigen Schritt, werden in einer Schleife jeweils aufeinanderfolgende Bildpaare an die Methode *Proc.motion*, zur Berechnung der affinen Parameter, übergeben.

- Anschließend wird die Methode *Extend.ext* zur Konstruktion des Panoramas aufgerufen. Wie weiter oben erwähnt, werden in diesem Methodenaufruf entweder graue oder farbige Bilder übergeben.

- Um das Panorama auch auf der Oberfläche des Plugins anzuzeigen, wird nun der Methode *GUI.paintPan* das Panoramaobjekt übergeben. Je nach Benutzereingabe erfolgen noch zusätzlich Methodenaufrufe von *GUI.paintXarea* oder *GUI.paintDiff*.

- Ist der komplette Image-Stack abgearbeitet, können die Zeitmessungen, die nebenher ermittelt wurden, umgerechnet und angezeigt werden.

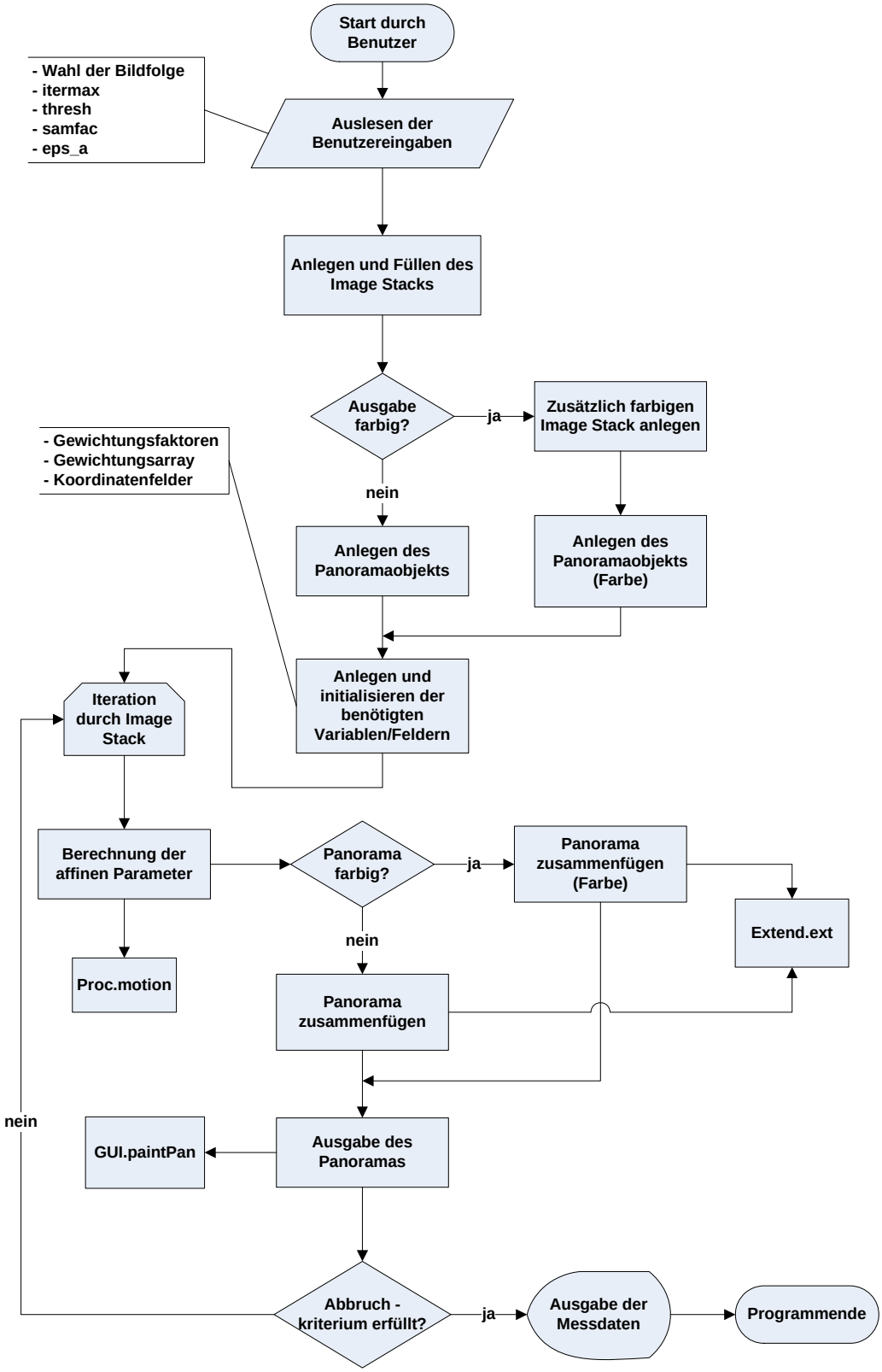


Abbildung 3.3: Flussdiagramm zum Programmablauf

3.5 Methodenbeschreibungen

Die Beschreibung der relevanten Methoden, ist bewusst sehr ausführlich gehalten worden, um die Einarbeitung für zukünftige Erweiterungen des Plugins zu erleichtern.

3.5.1 Proc.motion

Die Methode *motion* wird zur Berechnung des Bewegungsfeldes, in Form eines Parametervektors *a*, zwischen zwei aufeinanderfolgenden Bildern, aufgerufen.
Bei jedem Methodenaufruf werden zwei Bildprozessoren (*imRef* und *imCurr*), und eine Bounding Box als Parameter übergeben.
Nachdem alle, für diese Methode benötigten Variablen, angelegt und initialisiert wurden, wird bei jedem Methodenaufruf geprüft, ob sich die Größe der zu bearbeitenden Bilder geändert hat (Listing 3.1). Nur wenn das zutrifft, werden über die Methode *Util.getXmatYmat* angepasste Koordinatenfelder erzeugt. Zusätzlich werden die Gewichtungsfaktoren *wx* und *wy* neu errechnet (Zeilen 12 bis 19), die zur Berechnung einer Abbruchbedingung nötig sind. Anschließend werden die neuen Gewichtungsfaktoren an das Array *weigth_a* übergeben, und die Klassenvariablen *sizeXim* und *sizeYim* erhalten die neue Bildgröße.

```
01   if ((sizeXim!=imCurr.getWidth())||(sizeYim!=imCurr.getHeight())){
02      double[] cntXY = new double[2];
03      cntXY = Util.getXmatYmat(maskn, xmat, ymat);
04      cntx = cntXY[0];
05      cnty = cntXY[1];
06      for (y=0;y<sizeYim;y++){
07         for (x=0;x<sizeXim;x++){
08           xmat[y][x] -= cntx;
09           ymat[y][x] -= cnty;
10         }
11      }
12      for (y=0;y<sizeYim;y++){
13         for (x=0;x<sizeXim;x++){
14            wx += xmat[y][x] * xmat[y][x] * maskn[y][x];
15            wy += ymat[y][x] * ymat[y][x] * maskn[y][x];
16          }
17       }
18      wx = wx/sum_maskn;
19      wy = wy/sum_maskn;
20      weight_a[0][0] = wx;
21      weight_a[0][1] = wy;
22      weight_a[0][2] = 1;
23      weight_a[0][3] = wx;
24      weight_a[0][4] = wy;
25      weight_a[0][5] = 1;
26      sizeXim = imRef.getWidth();
27      sizeYim = imRef.getHeight();
28   }
```

Listing 3.1: Prüfung auf Änderung der Bildgröße

In Listing 3.2 werden die entsprechenden *x*- und *y*-Werte der Bounding Box *rect* ermittelt. Sie dienen dazu, die nachfolgenden Doppelschleifen, dieser und anderer Methoden, durch die Bounding Box der Bildmaske einzugrenzen.

```
01          xMin = (int)rect.getMinX();
02          xMax = (int)rect.getMinX()+rect.width;
03          yMin = (int)rect.getMinY();
04          yMax = (int)rect.getMinY()+rect.height;
```

Listing 3.2: Belegung der Bounding Box Variablen

Nach Ermittlung der Werte für die Bounding Box, wird die Methode *dFilter* aufgerufen. Diese errechnet aus dem Referenzbild die Werte für die zweidimensionalen Felder *I_x* und *I_y*, welche den partiellen Ableitungen nach den Gleichungen (2.9) entsprechen.
Die globale Compensated Motion, dargestellt durch die zweidimensionalen Felder *uc* und *vc*, wird einmal pro Methodenaufruf durch den affinen Parametervektor *a* initial berechnet (Listing 3.3, Zeilen 9 und 10), wobei der Vektor von einem vorherigen Bildpaar stammt. Handelt es sich bei dem aktuellen Methodenaufruf um das erste Bildpaar der Folge, werden die Felder *uc* und *vc* natürlich mit Null belegt (da Vektor *a* auch Null ist).

```
01          double a0 = a[0][0];
02          double a1 = a[0][1];
03          double a2 = a[0][2];
04          double a3 = a[0][3];
05          double a4 = a[0][4];
06          double a5 = a[0][5];
07          for (y=yMin;y<yMax;y=y+SAMFAC){
08              for (x=xMin;x<xMax;x=x+SAMFAC){
09                  uc[y][x] = (xmat[y][x]*a0) + (ymat[y][x]*a1) + a2;
10                  vc[y][x] = (xmat[y][x]*a3) + (ymat[y][x]*a4) + a5;
11              }
12          }
```

Listing 3.3: Die Berechnung von *uc* und *vc*

Der Rest der Methode wird nun durch die for-Schleife in Zeile 1 (Listing 3.4 solange iteriert, bis die Abbruchbedingung *delta_a<EPS_A* erfüllt ist, oder die maximale Anzahl an Iterationen (festgelegt durch die Variable *ITERMAX*) erreicht wurde.
In den Zeilen 2 bis 25 werden die Werte für das Feld *I_t*, an den durch *uc* und *vc* vorgegebenen Stellen (dargestellt durch *rpx* und *rpy*), kalkuliert. Die Abfrage in Zeile 4 stellt sicher, dass nur für die Stellen kalkuliert wird, an denen die Felder *I_x*, *I_y* und *maski* nicht mit Null belegt sind.
Die bilineare Interpolation (Zeile 17), die zur Errechnung der Werte für *I_t* in Zeile 18 angewendet wird, benötigt vier benachbarte Grauwerte (*val1* bis *val4*),

wobei diese natürlich nicht außerhalb des Bildbereiches bzw. außerhalb von *maskn* liegen dürfen, was durch die Abfrage in den Zeilen 9 und 10 sichergestellt ist. Um die Grauwerte des Bildes zu bekommen, ist eine bitweise Maskierung „*0xFF & newImPixels[]*“ nötig, um den Bytewert des Pixels, ohne Vorzeichen im Bereich 0…255 zu erhalten.
Bei *newImPixels* handelt es sich um eine Referenz auf das eindimensionale Pixelarray des Byteprozessors. Für jeden Zugriff auf ein Pixel an der Position (*floorX*, *floorY*), muss somit der eindimensionale Index innerhalb des Arrays berechnet werden, also: *newImPixels[floorY*sizeXim+floorX]*. Durch Verwendung eines eindimensionalen Arrays erfolgt der Zugriff auf die Werte sehr schnell.
In den Zeilen 21 und 22 wird die Pseudo-Motion *up* und *vp* gemäß Gleichung (2.13) errechnet. Anschließend muss das zweidimensionale Feld *accept* auf Null zurückgesetzt werden

```
01   for (int cnt=1; cnt<=ITERMAX; cnt++){
02      for (y=yMin;y<yMax;y=y+SAMFAC){
03          for (x=xMin;x<xMax;x=x+SAMFAC){
04             if (I_x[y][x]!=0 && I_y[y][x]!=0 && maski[y][x]!=0){
05                rpx = x+1 + uc[y][x];
06                rpy = y+1 + vc[y][x];
07                floorX = (int)rpx;
08                floorY = (int)rpy;
09                if ((!(floorX<0 || floorX>=sizeXim))&&(!(floorY<0 ||
10                   floorY>=sizeYim))&&(!(maskn[floorY][floorX]== 0))){
11                   dx = rpx - floorX;
12                   dy = rpy - floorY;
13                   val1 = 0xFF & newImPixels[(floorY-1)*sizeXim+floorX+1-1];
14                   val2 = 0xFF & newImPixels[(floorY-1)*sizeXim+floorX-1];
15                   val3 = 0xFF & newImPixels[(floorY+1-1)*sizeXim+floorX+1-1];
16                   val4 = 0xFF & newImPixels[(floorY+1-1)*sizeXim+floorX-1];
17                   Ip = (dx*val1+(1-dx)*val2)*(1-dy)+(dx*val3+(1-dx)*val4)*dy;
18                   I_t[y][x] = Ip - (0xFF & imPixels[y*sizeXim+x]);
19                }
20             }
21             up[y][x] = ((I_t[y][x]*-1)/(I_x[y][x]+1E-6))+uc[y][x];
22             vp[y][x] = ((I_t[y][x]*-1)/(I_y[y][x]+1E-6))+vc[y][x];
23             accept[y][x] = 0;
24          } // end for(x
25      } // end for(y
```

Listing 3.4: Die Berechnung von *I_t*, *up*, und *vp*

Auch in Listing 3.5 muss der Wert *Ip* mit Subpixel-Genauigkeit ermittelt werden. Hierfür wird ebenfalls die bilineare Interpolation verwendet, wobei diesmal die gerade errechneten Werte von *up* und *vp* für die Berechnung von *rpx* und *rpy* benötigt werden. Die Differenz aus dem interpolierten Punkt *Ip* des aktuellen Bildes (*imCurr*) und dem entsprechenden Punkt des vorherigen Bildes (*imRef*) *delta*, wird für den Test gemäß (Un-) Gleichung (2.10) in Zeile 2 benötigt, wobei es sich bei der Konstante *THRESH* um einen gewählten Schwellenwert handelt.

Alle Bildpunkte, die diesen Test bestehen, werden nun über das *accept*-Feld mit 1 markiert.

```
01          delta = Ip - (0xFF & imPixels[y*sizeXim+x]);
02          if (Math.abs(delta)<THRESH){
03             accept[y][x] = 1;
04          }
```

Listing 3.5: Der Test nach Gleichung (2.10)

Bevor in Listing 3.6, Zeile 4 ein Aufruf der Methode *lsAffine* zur Errechnung der affinen Parameter stattfindet, werden die schon aus einem vorherigen Bildpaar berechneten Parameter, (im Feld *a*) in ein temporäres Feld *a_prev* übergeben.
Nachdem die neuen Parameter durch *lsAffine* errechnet wurden, wird in Zeile 6 die Differenz zwischen den aktuellen und den alten Parametern gebildet, und in einem weiteren, eigens dafür angelegten Feld *a_temp* gespeichert.
Um den Wert für *delta_a* ermitteln zu können, müssen alle Elemente in *a_temp* mit 2 potenziert werden (Zeile 9). Anschließend wird die Summe aller Elemente des Matrizenproduktes von *weight_a* und *a_temp*, durch Aufruf der Methode *Util.sum* ermittelt. Zusätzlich wird in der gleichen Zeile (11) noch die Wurzel aus der errechneten Summe gezogen.

```
01          for (int i=0; i<=5; i++){
02              a_prev[0][i] = a[0][i];
03          }
04          lsAffine(SAMFAC, xmat, accept, up, vp, a, xMin, xMax, yMin, yMax);
05          for (int i=0; i<=5; i++){
06              a_temp[i][0] = a[0][i]-a_prev[0][i];
07          }
08          for (int i=0; i<=5; i++){
09              a_temp[i][0] = a_temp[i][0]*a_temp[i][0];
10          }
11          delta_a = Math.sqrt(Util.sum(Util.arrTimes(weight_a, a_temp)));
```

Listing 3.6: Berechnung von *delta_a*

Schließlich werden die Felder *uc* und *vc* aus den aktuellen affinen Parametern neu errechnet, um wie in Listing 3.4, als Grundlage für die Berechnung von *up* und *vp* zu dienen (oder als Initialschätzung der Bewegung für das nächste Bildpaar).

```
01          for (y=yMin;y<yMax;y=y+SAMFAC){
02              for (x=xMin;x<xMax;x=x+SAMFAC){
03                  uc[y][x] = (xmat[y][x]*a0) + (ymat[y][x]*a1) + a2;
04                  vc[y][x] = (xmat[y][x]*a3) + (ymat[y][x]*a4) + a5;
05              }
06          }
07          if (delta_a<EPS_A){
08              break;
09          }
```

Listing 3.7: Berechnung von *uc*, *vc* und Abbruchbedingung

In den Zeilen 7 bis 9 wird hingegen geprüft, ob die kompensierte Bewegung von der letzten Iteration zu dieser nur gering ($< EPS\_A$) ist. Sollte das der Fall sein, ist die Berechnung der Bewegung für dieses Bildpaar abgeschlossen.

3.5.2 Util.getXmatYmat

Das nachfolgende Listing beinhaltet den kompletten Quellcode der Methode *Util.getXmatYmat*. Ziel dieser Methode ist es aus einer gegebenen Maske *maskn*, zwei zweidimensionale Koordinaten-Arrays (*xmat*, *ymat*), und die dazugehörigen Mittelpunkte der Maske zu berechnen. Somit kann jedem Bildpunkt ein Koordinatenpaar zugeordnet werden.

```
01     public static double[] getXmatYmat(double[][] maskn, double[][] xmat,
02                                        double[][] ymat){
03         int sizeY = maskn.length;
04         int sizeX = maskn[0].length;
05         int sumMaskn;
06         double cntx, cnty;
07         double countsumXmat = 0.0;
08         double countsumYmat = 0.0;
09         for (int y=0;y<sizeY;y++){
10             for (int x=0;x<sizeX;x++){
11                 xmat[y][x] = x;
12                 ymat[y][x] = y;
13                 countsumXmat += xmat[y][x] * maskn[y][x];
14                 countsumYmat += ymat[y][x] * maskn[y][x];
15             }
16         }
17         sumMaskn = (int)sum(maskn);
18         cntx = countsumXmat/sumMaskn;
19         cnty = countsumYmat/sumMaskn;
20         double[] cntXY = new double[2];
21         cntXY[0] = cntx;
22         cntXY[1] = cnty;
23         return cntXY;
24     }
```

Listing 3.8: Die Methode *getXmatYmat*

Um die Koordinaten-Arrays anzulegen, werden in einer Doppelschleife (Zeile 9-16) beide Arrays zeilenweise mit den entsprechenden Wertigkeiten von *x* bzw. *y* belegt. Gleichzeitig werden in den Zeilen 13 und 14 alle gerade belegten, und in der Maske befindlichen Werte, aufsummiert.

```
-3.5 -2.5 -1.5 -0.5 0.5 1.5 2.5 3.5
-3.5 -2.5 -1.5 -0.5 0.5 1.5 2.5 3.5
-3.5 -2.5 -1.5 -0.5 0.5 1.5 2.5 3.5
-3.5 -2.5 -1.5 -0.5 0.5 1.5 2.5 3.5
-3.5 -2.5 -1.5 -0.5 0.5 1.5 2.5 3.5
-3.5 -2.5 -1.5 -0.5 0.5 1.5 2.5 3.5
-3.5 -2.5 -1.5 -0.5 0.5 1.5 2.5 3.5
-3.5 -2.5 -1.5 -0.5 0.5 1.5 2.5 3.5

-3.5 -3.5 -3.5 -3.5 -3.5 -3.5 -3.5 -3.5
-2.5 -2.5 -2.5 -2.5 -2.5 -2.5 -2.5 -2.5
-1.5 -1.5 -1.5 -1.5 -1.5 -1.5 -1.5 -1.5
-0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5 -0.5
0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
1.5 1.5 1.5 1.5 1.5 1.5 1.5 1.5 1.5 1.5
2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5
3.5 3.5 3.5 3.5 3.5 3.5 3.5 3.5 3.5 3.5
```

Abbildung 3.4: Die Werte von xmat (oben) und ymat (unten) um die jeweiligen Mittelpunkte (nach Subtraktion von cntx und cnty)

Nachdem die Anzahl der Bildpunkte, die sich in der Maske befinden, ermittelt wurde, können durch Division (Zeile 18 und 19) die Werte für die Mittelpunkte der Maske errechnet werden.
In einem letzten Schritt, werden beide Werte in das Array *cntXY* gelegt, und schließlich zurückgegeben.

3.5.3 Proc.dFilter

Die folgende Methode wird von *Proc.motion* aus einmal pro Bildpaar aufgerufen. Sie berechnet die partiellen Ableitungen, in der übergebenen Bounding Box, nach *x* und *y*, die zur Ermittlung der Pseudo-Motion benötigt werden.
Der Zugriff auf die Grauwerte des eindimensionalen Bytearrays *imPixels*, erfolgt auf die gleiche Weise, die schon in der Methode *motion* beschrieben wurde.

```
01   private static void dFilter(byte[] imPixels, double[][] I_x, double[][] I_y,
                                 int xMin, int xMax, int yMin, int yMax, int
                                 sizeX){
02     int x, y, pos;
03     for(y=yMin; y<=yMax; y++){
04        pos = y*sizeX;
05        for(x=xMin; x<=xMax; x++){
06           I_x[y][x] = 0.5*((0xFF & imPixels[pos+x+1])-
07                           (0xFF & imPixels[pos+x-1]));
08           I_y[y][x] = 0.5*((0xFF & imPixels[(sizeX+pos+x)])-
09                           (0xFF & imPixels[pos+x-sizeX]));
10        }
11     }
12   }
```

Listing 3.9: Die Methode *dFilter*

Für jeden Bildpunkt werden in den Zeilen 6 und 8 die jeweiligen Nachbarwerte ermittelt, wobei der linke (bei der Ermittlung nach *x*) bzw. obere (bei Ermittlung nach *y*) von dem anderen Nachbarpunkt subtrahiert wird. Nach Addition beider Werte, wird das Ergebnis halbiert.

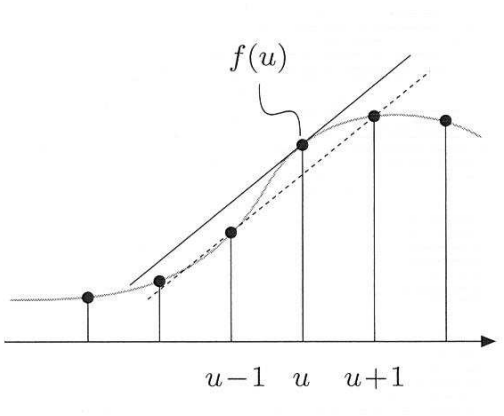


Abbildung 3.5: Schätzung der ersten Ableitung

Zur Verdeutlichung der Idee soll Abbildung 3.5 dienen, die, für den eindimensionalen Fall, die Schätzung der ersten Ableitung einer Funktion darstellt.
Da es sich hierbei um eine diskrete Funktion handelt, kann der Anstieg der Tangente an der Stelle *u* einfach dadurch geschätzt werden, dass eine Gerade durch die Abtastwerte der beiden Nachbarpunkte an den Stellen *u-1* und *u+1* gezogen wird. Die Berechnung des Anstiegs gleicht exakt der Vorgehensweise im Quellcode:

$$\frac{df}{du}(u) = 0.5\big(f(u+1) - f(u-1)\big)$$

3.5.4 Proc.lsAffine

In dieser Methode werden mittels Least Square-Schätzung die affinen Parameter durch Lösen eines Gleichungssystems errechnet.
Hierfür wird das Jama[12] Matrix-Paket verwendet, welches für Matrizen eine Methode *solve* zur Verfügung stellt.
Um das Gleichungssystem zu lösen, und die Matrizen nach Gleichung (2.16) zu füllen, müssen zuerst die entsprechenden Summen der einzelnen Pixel ermittelt werden.

[12] Jama - (Java Matrix Package): stellt fundamentale Operationen der linearen Algebra, wie Addition, Multiplikation, Berechnung der Determinante, Inversen und verschiedenen Normen einer Matrix zur Verfügung

Dies geschieht in den Zeilen 7 bis 27. Durch die temporären Variablen *acc*, *xacc*, *yacc*, *upval* und *vpval* wird ein unnötiger, mehrmaliger Zugriff auf die entsprechenden Felder vermieden.

```
01  private static void lsAffine(int samfac, double[][] xmat, double[][] ymat,
02                          double[][] accept, double[][] up, double[][]vp,
03                          double[][] a, int xMin, int xMax, int yMin, int
                             yMax){
04     (…)
05     /*Initialisierung der Variablen, und Anlegen der Matrizen*/
06     (…)
07      s  = Util.sum(accept);
08      for (y=yMin;y<yMax;y=y+samfac){
09         for (x=xMin;x<xMax;x=x+samfac){
10             acc = accept[y][x];
11             xacc = xmat[y][x] * acc;
12             yacc = ymat[y][x] * acc;
13             upval = up[y][x];
14             vpval = vp[y][x];
15             xx += xacc * xmat[y][x];
16             xy += xacc * ymat[y][x];
17             yy += yacc * ymat[y][x];
18             x1 += xacc;
19             y1 += yacc;
20             xu += xacc * upval;
21             yu += yacc * upval;
22             u  += acc * upval;
23             xv += xacc * vpval;
24             yv += yacc * vpval;
25             v  += acc * vpval;
26         }
27      }
28      rhs_u.set(0, 0, xu);
29      rhs_u.set(1, 0, yu);
30      rhs_u.set(2, 0, u);
31      rhs_v.set(0, 0, xv);
32      rhs_v.set(1, 0, yv);
33      rhs_v.set(2, 0, v);
34      M.set(0, 0, xx);
35      M.set(1, 0, xy);
36      M.set(2, 0, x1);
37      M.set(0, 1, xy);
38      M.set(1, 1, yy);
39      M.set(2, 1, y1);
40      M.set(0, 2, x1);
41      M.set(1, 2, y1);
42      M.set(2, 2, s);
43      a_u = M.solve(rhs_u);
44      a_v = M.solve(rhs_v);
45      for (int i=0; i<=2; i++){
46         a[0][i] = a_u.get(i, 0);
47         a[0][i+3] = a_v.get(i, 0);
48      }
49  }
```

Listing 3.10: Die Methode *lsAffine*

Nun werden die Matrizen mit den jeweiligen Summen gefüllt, was in den Zeilen 28 bis 42 geschieht. Durch Anwendung der Methode *solve* auf die entsprechenden Matrizen, erhält man nun die geschätzten Parameter die in den Zeilen 46 und 47 in den Parametervektor *a* geschrieben werden.

3.5.5 Extend.ext

Um die einzelnen Bilder der aktuellen Bildfolge zu einem Panorama zusammenfassen zu können, wird für jedes Bild, das in das Panorama integriert werden soll, die Methode *ext* aufgerufen. Bei diesem Aufruf werden als Methodenparameter ein Objekt der Klasse *PanoAffine*, der Bildprozessor des zu addierenden Bildes, der Parametervektor *a* und drei Variablen, nämlich *outputRGB, diffImg* und *doExt*, übergeben.
Anfangs werden alle benötigten Felder und Matrizen erzeugt, falls es sich um den ersten Aufruf der Methode handelt.

```
01      A.set(0, 0, a[0][0]);
02      A.set(0, 1, a[0][1]);
03      A.set(0, 2, a[0][2]);
04      A.set(1, 0, a[0][3]);
05      A.set(1, 1, a[0][4]);
06      A.set(1, 2, a[0][5]);
07      A.set(2, 0, 0);
08      A.set(2, 1, 0);
09      A.set(2, 2, 0);
10      w.t = id.plus(A).times(w.t);
11      w.tA = w.t.getArray();
```

Listing 3.11: Berechnung der Transformationsmatrix

In den Zeilen 1 bis 9 des obigen Listings wird Matrix *A* mit den bereits errechneten Freiheitsgraden der affinen Abbildung gefüllt. Anschließend wird diese Matrix mit der Einheitsmatrix *id* addiert, und das Ergebnis mit der Transformationsmatrix *t* des Panorama-Objektes *w* multipliziert. In Zeile 11 speichert die Feldvariable *tA* die Referenz des internen Array von *t*, für spätere Berechnungen.
Im nächsten Listing (3.12), zwischen den Zeilen 3 und 17, wird iterativ ein 20 Pixel breiter Bereich (die *w.X_Area* bzw. *xAreaPixels*) um die aktuelle Maske *w.maskp* (bzw. *maskpPixels*) gebildet. Dabei wird die Bounding Box, die in Zeile 18 gesetzt wird, gleich den neuen Dimensionen von *X_Area* angepasst (11-14). Zu diesem Zeitpunkt enthält *X_Area* allerdings noch alle Punkte die auch Maske *w.maskp* enthält.

Abbildung 3.6: Ein Panorama und seine dazugehörige *X_Area*

Um das zu ändern, werden in einem nächsten Schritt (19-26) alle Pixel, die sich im selben Bereich wie *w.maskp* befinden, auf Null gesetzt. Das Ergebnis dieses Schrittes ist in Abbildung 3.6 zu sehen.

```
01   int pos;
02   int mx20 = mx*20;
03   for (int y=yMin;y<yMax;y++){
04      pos = y*mx;
05      for (int x=xMin;x<xMax;x++){
06         if (maskpPixels[x+pos]==1){
07             if (!(x-20<0)&&!(y-20<0)){xAreaPixels[(x-20)+pos-mx20]= 1;}
08             if (!(x+20>=mx)&&!(y+20>=my)){xAreaPixels[(x+20)+pos+mx20]= 1;}
09             if (!(y-20<0)&&!(x+20>=mx)){xAreaPixels[(x+20)+pos-mx20]= 1;}
10             if (!(x-20<0)&&!(y+20>=my)){xAreaPixels[(x-20)+pos+mx20]= 1;}
11             if ((x-20<xTempLeft)&&(x-20>0)){xTempLeft = x-20;}
12             if ((x+20>xTempRight)&&(x+20<mx)){xTempRight = x+20;}
13             if ((y-20<yTempTop)&&(y-20>0)){yTempTop = y-20;}
14             if ((y+20>yTempDown)&&(y+20<my)){yTempDown = y+20;}
15          }
16      }
17   }
18   rect.setBounds(xTempLeft, yTempTop, (xTempRight-xTempLeft)+1,
                    (yTempDown-yTempTop)+1);
19   for (int y=yMin;y<yMax;y++){
20      pos = y*mx;
21      for (int x=xMin;x<xMax;x++){
22          if (maskpPixels[x+pos]==1){
23              xAreaPixels[x+pos]=0;
24          }
25      }
26   }
27   int xArCount = rect.height*rect.width;
28   double[] xvec = new double[xArCount];
29   double[] yvec = new double[xArCount];
30   int count = 0;
31   for (int y=yMin;y<yMax;y++){
32          pos = y*mx;
33          for (int x=xMin;x<xMax;x++){
34                 if (xAreaPixels[x+pos]==1){
35                        xvec[count] = w.xmat[y][x];
36                        yvec[count] = w.ymat[y][x];
37                        count++;
38                 }
39          }
40   }
```

Listing 3.12: Auszug aus der Methode *Extend.ext* (1)

In den Zeilen 28 und 29 werden zwei eindimensionale Koordinatenfelder (*xvec*, *yvec*) angelegt, wobei sie jeweils nach der aktuellen Größe der Bounding Box dimensioniert werden. Danach werden sie in den Doppelschleifen (31-40) mit den jeweiligen Koordinatenwerten, der Koordinatenfelder *w.xmat* und *w.ymat*, belegt. Allerdings werden nur solche Koordinaten übergeben, die in dem Feld *X_Area* liegen.
Die Abfrage in Listing 3.13, Zeile 1 dient dazu, ein farbiges Panorama zu bilden. Hierbei wird die Variable *outputRGB* ausgewertet, und die Methode *interpolateRGB* aufgerufen. Diese Methode ähnelt sehr dem *else*-Zweig der Abfrage, allerdings wurde sie ausgelagert, um die Übersichtlichkeit des Codes zu erhöhen.
Ab Zeile 4 wird nun jedes Pixel des Feldes *X_Area*, entsprechend dem Transformationsarray *w.tA* (aus Performancegründen nicht *w.t*) geometrisch transformiert. Hierfür werden die Koordinaten des jeweiligen aktuellen Pixels (*x*, *y*) mit *w.tA* multipliziert (7-12). Die Variablen *rpx* und *rpy* erhalten durch Addition des Ergebnisses mit den Koordinaten des Ursprungs die endgültige Stelle des transformierten Pixels.

```
01   if (outputRGB) interpolateRGB(w, xvec, yvec, newIm, cntxMinox, cntxMinoy,
02                                 cntxTemp, cntyTemp, xArCount, mx, my);
03   else {
04      for (int k=0; k<xArCount; k++){
05      x = xvec[k]-cntxMinox;
06      y = yvec[k]-cntyMinoy;
07      rp1 = w.tA[0][0] * x;
08      rp1 += w.tA[0][1] * y;
09      rp1 += w.tA[0][2];
10      rp2 = w.tA[1][0] * x;
11      rp2 += w.tA[1][1] * y;
12      rp2 += w.tA[1][2];
13      rpx = rp1 + w.cntx;
14      rpy = rp2 + w.cnty;
14      floorX = (int) rpx;
16      floorY = (int) rpy;
17      if ((!(floorX<0 || floorX>=mx))&&(!(floorY<0 || floorY>=my))&&
           (!(w.maskn[floorY][floorX]== 0))){
18          dx = rpx - floorX;
19          dy = rpy - floorY;
            (…) bilineare Interpolation (…)
20          panoimPixels[(int)yvec[k]*mx+(int)xvec[k]] = (byte)(int)Ip;
21          maskpPixels[(int)yvec[k]*mx+(int)xvec[k]] =  1;
22            if (diffImg)
23              w.Xerr.putPixelValue((int)xvec[k], (int)yvec[k],
24              Math.abs(newIm.getPixelValue(floorX, floorY)-(int)Ip)+100);
25      }
26   }
```

Listing 3.13: Auszug aus der Methode *Extend.ext* (2)

Die nachfolgende bilineare Interpolation, die schon aus der Methode *Proc.motion* bekannt ist, ermittelt die Werte der transformierten Pixel noch mit Subpixel-Genauigkeit. Anschließend wird dem Panorama *w.panoim* (bzw. *panoimPixels*)

der Pixelwert übergeben, und an der entsprechenden Stelle die Panoramamaske *w.maskp* erweitert.
Wie bereits erwähnt, läuft die Ermittlung der Pixelwerte für farbige Pixel ähnlich ab. Der Unterschied besteht in der leicht veränderten Interpolation. Hier wird nicht für einen Grauwert interpoliert, sondern es werden aus den jeweiligen Pixeln die Rot-, Grün- und Blau-Werte, durch Bitoperationen in den Zeilen 2 bis 4 (Listing 3.14) extrahiert, und für jeden Farbwert einzeln (wie in Zeile 5 für den Rotwert) eine Interpolation durchgeführt. Der Zusammenbau des RGB-Pixels erfolgt in umgekehrter Weise, durch Verwendung der bitweisen ODER-Operation und Verschiebung nach links (Zeile 6).

```
01   val1 = newIm.getPixel(floorX, floorY-1);
02   rgb[0][0] = (val1 & 0xff0000) >> 16;
03   rgb[0][1] = (val1 & 0x00ff00) >> 8;
04   rgb[0][2] = (val1 & 0x0000ff);
     (…) gleich für val 2 bis 4 (…)
05   r = (int)(((dx*rgb[0][0])+((1-dx)*rgb[1][0])) * (1-dy)+
              ((dx*rgb[2][0])+(1-dx)*rgb[3][0]) * dy);
06   Ip = ((r & 0xff)<<16)|((g & 0xff)<<8)|b & 0xff;
07   w.panoim.putPixel((int)xvec[k], (int)yvec[k], (int)Ip);
```

Listing 3.14: Auszug aus *interpolateRGB*

Zum Thema Panoramaerzeugung ist noch zu anzumerken, dass hier drei verschiedene Strategien existieren. Bei der in diesem Plugin angewendeten Strategie, wird nur an den Stellen gemalt, an denen vorher noch keine Bilder um das Mosaik gesetzt wurden (*X_Area*).
Das heißt, dass das Referenzbild niemals übermalt wird. Diese Methode wurde gewählt, da sie die schnellste Erzeugung eines Panoramas gewährleistet.
Eine weitere Strategie ist es, das aktuelle Bild so komplett wie möglich zu malen. Somit ist das Panorama immer auf dem aktuellsten Stand was die in das Mosaik hinzugefügten Bilder angeht.
Die dritte Strategie wählt zwischen beiden Bildern jenes aus, welches den größten Flächenvergrößerungsfaktor besitzt. Diese Art der Panoramaerzeugung bietet also an jeder Stelle die beste verfügbare Auflösung.

Kapitel 4

Die Optimierung der Performance

Neben der Implementierung und Umsetzung des Matlab Codes, ist das Performance-Tuning des Plugins einer der zentralen Punkte dieser Arbeit. Kapitel 4 befasst sich hauptsächlich mit den einzelnen Schritten, in denen der Quellcode optimiert wurde. Bevor die erzielten Ergebnisse aber zusammengefasst, und graphisch dargestellt werden, behandelt ein weiteres Unterkapitel die verschiedenen Eingabeparameter, und ihre Auswirkungen auf Performance und Genauigkeit.

4.1 Methoden der Performancemessung

Die Optimierung des Codes erfolgte schrittweise während, sowie nach der Implementierung. Die Messung der jeweiligen aktuellen Performance wurde mittels der Methode *System.nanoTime* durchgeführt. Diese Methode gibt, zu einem Zeitpunkt 1, die Anzahl der vergangenen Zeit seit dem 01.01.1970 in Nanosekunden zurück. Wird sie zu einem späteren Zeitpunkt 2 nochmals angewendet, so kann durch Subtraktion der zweiten von der ersten Zeit, recht genau die vergangene Zeitspanne ermittelt werden.
Neben dieser Möglichkeit zur Messung der Performance, wurde noch der Profiler TPTP eingesetzt. Mit ihm ist es, in einem wesentlich größeren Umfang, möglich Messungen vorzunehmen. Allerdings handelt es sich bei den Messungen nicht um die tatsächliche Geschwindigkeit, da die Anwendung des Profiling an sich schon sehr viel Zeit bzw. Rechenleistung in Anspruch nimmt. Außerdem kann es vorkommen, dass das Ergebnis teilweise sehr überraschend ausfällt, und sich die Messung nicht proportional zu der Messung der oben erwähnten Methode verhält. Der Vorteil des Profilers liegt vielmehr darin, langsame Programmteile ausfindig zu machen, und sie dann schrittweise zu optimieren.

4.2 Optimierung des Quellcodes

Der folgende Abschnitt beschreibt die Optimierungsschritte in chronologischer Weise, wobei hier zwischen allgemeinen Optimierungen, die sich auf alle zeitkritischen Methoden auswirken, und speziellen Optimierungen, die sich nur auf die jeweilige Methode auswirken, unterschieden werden kann.
Zu Beginn wurden erst alle Codeteile optimiert, die für die Ermittlung der affinen Parameter von Bedeutung sind (also die Methoden *Proc.motion, Proc.lsAffine* und *Proc.dFilter*). Erst anschließend wurde die Klasse *Extend* zur Erzeugung des Panoramas ins Visier genommen.
Die Messungen werden in jeweils zwei Tabellen dargestellt. Die erste Tabelle erhält die Messergebnisse der Methode *System.nanoTime*, die zweite die des Profilers. Als Basis[13] für alle folgenden Messungen dient die Bildfolge *jrobertB*, welche aus 31 Grauwertbildern besteht, und eine Maskengröße von 8016 Pixel besitzt.

Im Laufe des Optimierungsprozesses wurden natürlich noch weitere Methoden des Plugins beschleunigt (z.B. *Util.getXmatYmat* oder *PreProc.getMasks*). Allerdings sind hier die Auswirkungen auf die Gesamtperformance, wenn überhaupt, nur sehr gering, da sie meist nicht häufiger als ein Mal aufgerufen werden.

4.2.1 Die Performance in unoptimiertem Zustand (motion, dFilter, lsAffine)

Die erste Messung der Performance, ohne jegliche Optimierung, ergab:

	Komplette Zeit im ms	**Zeit pro Bildpaar in ms**	**Bilder pro Sekunde**
Zeitmessung	18569	618,97	1,62

Tabelle 4.1: Ergebnis der Messung mit *System.nanoTime*

Der Profiler lieferte zu dieser Zeit die folgenden Werte:

Methode	**Durchschnittliche Dauer in Sekunden**
Proc.lsAffine	0,0287
Proc.motion	33,4310
Proc.dFilter	4,2065

Tabelle 4.2: Ergebnis des Profilers

[13] Die CPU des Messcomputers ist ein Pentium M mit 1,6 GHz

4.2.2 Optimierungen an den Methoden motion, dFilter und lsAffine

Schritt 1:

In einem ersten Schritt wurden alle *get*- und *set*- Methoden, der zu diesem Zeitpunkt noch verwendeten Jama-Matrizen, durch Zugriffe auf deren interne Arrays ersetzt. Diese Änderung betraf alle drei Methoden.

Vorher:

```
for (y=0;y<sizeYnew;y++){
        for (x=0;x<sizeXnew;x++){
                wx += xmat.get(y,x) * xmat.get(y,x) * maskn get(y,x);
                wy += ymat get(y,x) * ymat get(y,x) * maskn get(y,x);
        }
}
```

Nachher:

```
double[][] xmatA = xmat.getArray();
double[][] ymatA = ymat.getArray();
double[][] masknA = maskn.getArray();
for (y=0;y<sizeYnew;y++){
        for (x=0;x<sizeXnew;x++){
                wx += xmatA[y][x] * xmatA[y][x] * masknA[y][x];
                wy += ymatA[y][x] * ymatA[y][x] * masknA[y][x];
        }
}
```

Listing 4.1: Austausch der *get*- und *set*- Methoden

Es konnte hierbei ein Geschwindigkeitsgewinn von 14,81 % erzielt werden.

	Komplette Zeit im ms	**Zeit pro Bildpaar in ms**	**Bilder pro Sekunde**
Zeitmessung	16125	537,5	1,86

Tabelle 4.3: Ergebnis der Messung mit *System.nanoTime* (Schritt 1)

Methode	**Durchschnittliche Dauer in Sekunden**
Proc.lsAffine	0,0047
Proc.motion	29,0610
Proc.dFilter	2,2017

Tabelle 4.4: Ergebnis des Profilers (Schritt 1)

Schritt 2:

Bei der Ermittlung der Pixelwerte in den Methoden *Proc.motion* und *Proc.dFilter*, wurde bisher die Methode *getPixelValue* verwendet. In diesem Schritt wurde sie nun durch den Zugriff auf das wesentlich schnellere, eindimensionale Pixelarray des Bildprozessors ersetzt.

```
Vorher:
val1 = newIm.getPixelValue((int)floorY, (int)floorX+1);

Nachher:
val1 = 0xFF & newImPixels[(floorY-1)*sizeXim+floorX+1-1];
```

Listing 4.2: Austausch einer *getPixelValue*- Methoden

	Komplette Zeit im ms	**Zeit pro Bildpaar in ms**	**Bilder pro Sekunde**
Zeitmessung	14000	466,67	2,14

Tabelle 4.5: Ergebnis der Messung mit *System.nanoTime* (Schritt 2)

Methode	**Durchschnittliche Dauer in Sekunden**
Proc.lsAffine	0,0047
Proc.motion	23,2580
Proc.dFilter	0,2958

Tabelle 4.6: Ergebnis des Profilers (Schritt 2)

Laut den Messungen, wurde durch den Austausch der Zugriffsmethode, ein Performancegewinn von 15,05% erzielt.

Schritt 3:

Hier wurden die Doppelschleifen der bilinearen Interpolation in *Proc.motion*, durch Verwendung einer Bounding Box (*yMin, yMax und xMin, xMax*) eingeschränkt. Der Effekt dieser Maßnahme ist allerdings sehr gering, da die Anzahl der Aufrufe (der Berechnungen) ohnehin schon durch die einleitende *if*-Abfrage stark vermindert wird.

```
for (y=yMin;y<yMax;y=y+SAMFAC){
       for (x=xMin;x<xMax;x=x+SAMFAC){
              if (I_x[y][x]!=0 && I_y[y][x]!=0 && maski[y][x]!=0){
```

Listing 4.3: Einschränkung der Doppelschleife durch eine Bounding Box

Außerdem wird nun verhindert, dass die zweidimensionalen Felder (*I_x*, *I_y* etc.), zu Beginn der *Proc.motion*-Methode, bei jedem Methodenaufruf neu angelegt werden, wenn sie schon existieren.

```
if (I_x==null){I_x = new double[sizeYim][sizeXim];}
if (I_y==null){I_y = new double[sizeYim][sizeXim];}
if (uc==null){uc  = new double[sizeYim][sizeXim];}
if (vc==null){vc  = new double[sizeYim][sizeXim];}
(…)
```

Listing 4.4: Eine Abfrage, ob die jeweiligen Felder bereits angelegt wurden

Der Perfomancegewinn von 35,98% in diesem Schritt ist größtenteils dem zweiten Punkt zuzuschreiben.

	Komplette Zeit im ms	Zeit pro Bildpaar in ms	Bilder pro Sekunde
Zeitmessung	10280	342,67	2,91

Tabelle 4.7: Ergebnis der Messung mit *System.nanoTime* (Schritt 3)

Methode	Durchschnittliche Dauer in Sekunden
Proc.lsAffine	0,0047
Proc.motion	12,6068
Proc.dFilter	0,2958

Tabelle 4.8: Ergebnis des Profilers (Schritt 3)

Schritt 4:

Die verwendeten Jama-Methoden sind, besonders bei Verwendung großer Matrizen, sehr unperformant. Dies kommt vor allem durch die häufigen Exception-Tests und Bounds-Checks, die bei den einzelnen Zugriffen durchgeführt werden. Daher wurden sie komplett durch eigene, iterative Berechnungen ersetzt.

```
       Vorher:
       up = (I_tTemp1.times(-1).arrayRightDivide(I_xTemp.plus(notZero))).plus(uc)
;
       vp = (I_tTemp2.times(-1).arrayRightDivide(I_yTemp.plus(notZero))).plus(vc)
;

       Nachher:
       for (y=0;y<sizeYnew;y++){
              for (x=0;x<sizeXnew;x++){
                     up[y][x] = ((I_tA[y][x]*-1)/(I_xA[y][x]+1E-6))+uc[y][x];
                     vp[y][x] = ((I_tA[y][x]*-1)/(I_yA[y][x]+1E-6))+vc[y][x];
              }
       }
```

Listing 4.5: Austausch der Jama-Methoden

Auch bei den beiden Doppelschleifen der bilinearen Interpolation wurde eine kleine Änderung vorgenommen. Statt wie bisher das Vektorfeld *rp* zu benutzen, werden nun die zwei Variablen *rpx* und *rpy* verwendet, um die, in diesem Bereich, häufigen Arrayzugriffe zu unterbinden.
Außerdem wird nun vor jedem Aufruf der Methode *Util.getXmatYmat* geprüft, ob sich die Größe der übergebenen Bildprozessoren verändert hat. Nur in diesem Fall ist eine Neuberechnung der Koordinatenmatrizen nötig.

	Komplette Zeit im ms	Zeit pro Bildpaar in ms	Bilder pro Sekunde
Zeitmessung	2852	95,67	10,45

Tabelle 4.9: Ergebnis der Messung mit *System.nanoTime* (Schritt 4)

Der Gewinn an Performance beträgt nun 259,11% gegenüber dem vorherigen Schritt.

Methode	Durchschnittliche Dauer in Sekunden
Proc.lsAffine	0,0047
Proc.motion	3,3376
Proc.dFilter	0,2958

Tabelle 4.10: Ergebnis des Profilers (Schritt 4)

Schritt 5:

Der folgenden Änderung liegt die Idee zugrunde, dass ein einfacher int-Cast, in dieser Anwendung, das gleiche Resultat erzielt wie die verwendete Methode Math.floor.

Die Methode wurde also in den beiden Doppelschleifen der bilinearen Interpolation in *Proc.motion* ersetzt.

```
Vorher:
floorX = Math.floor(rpx);
floorY = Math.floor(rpy);

Nachher:
floorX = (int)rpx;
floorY = (int)rpy;
```

Listing 4.6: Austausch der Methode *Math.floor*

Ein weiterer Punkt in diesem Schritt, ist die *inMask-Methode* inline zu setzen. Hierbei wird der Rumpf der Methode an die Stelle des Methodenaufrufs kopiert. Da der Compiler für jeden einzelnen Pixel zu dieser Methode springen muss, ist es zeitsparender die Abfrage direkt vor Ort stattfinden zu lassen.

```
Vorher:
if (inMask(rp, maskn)){

Nachher:
if ((!(floorX<0 || floorX>=sizeXnew))&&(!(floorY<0 ||
       floorY>=sizeYnew))&&(!(maskn[floorY][floorX]== 0))){
```

Listing 4.7: Die Methode *inMask* wird inline gesetzt

In der Methode *Proc.dFilter* wurde aus dem Array *filter* der nicht benötigte Null-Wert gestrichen. Das Ergebnis der Berechnungen für *I_x* und *I_y* wird nun direkt übergeben, ohne in die entsprechenden Zwischenvariablen geschrieben zu werden. Der Quellcode aus beiden Schleifen wurde in eine Schleife verschoben.

```
• int[] filter = {-1, 1};

• for(i=0; i<=w; i++){
      o  (…)
      o  p = (0xFF & imPixels[y*sizeXim+(x+c)]);
      o  sumx += c*p;
      o  p = (0xFF & imPixels[(c*sizeXim)+(y*sizeXim+x)]);
      o  sumy += c*p;
  }

• I_x[y][x] = s*sumx;
  I_y[y][x] = s*sumy;
```

Listing 4.8: Änderungen in *Proc.dFilter*

Durch diese Änderungen steigt die Performance um 22,68 %.
Die Messungen durch *System.nanoTime* und dem Profiler sind den folgenden Tabellen zu entnehmen:

	Komplette Zeit im ms	**Zeit pro Bildpaar in ms**	**Bilder pro Sekunde**
Zeitmessung	2344	78,13	12,82

Tabelle 4.11: Ergebnis der Messung mit *System.nanoTime* (Schritt 5)

Methode	**Durchschnittliche Dauer in Sekunden**
Proc.lsAffine	0,0047
Proc.motion	0,2894
Proc.dFilter	0,0879

Tabelle 4.12: Ergebnis des Profilers (Schritt 5)

Schritt 6:
In diesem Schritt wurden die Schleifen in *Proc.dFilter*, *Proc.lsAffine*, und die zur Berechnung von *uc*, *vc* bzw. *up*, *vp* in *Proc.motion*, nun ebenfalls durch Einschränkung mittels einer Bounding Box, beschleunigt.

```
for (y=yMin;y<yMax;y=y+samfac){
      for (x=xMin;x<xMax;x=x+samfac){
            uc[y][x] = (xmat[y][x]*a0) + (ymat[y][x]*a1) + a2;
            vc[y][x] = (xmat[y][x]*a3) + (ymat[y][x]*a4) + a5;
      }
}
```

Listing 4.9: Einschränkung durch Bounding Box

Das Resultat ist eine große Steigerung der Performance um 168,96 %.

	Komplette Zeit im ms	**Zeit pro Bildpaar in ms**	**Bilder pro Sekunde**
Zeitmessung	896	29,87	34,48

Tabelle 4.13: Ergebnis der Messung mit *System.nanoTime* (Schritt 6)

Methode	Durchschnittliche Dauer in Sekunden
Proc.lsAffine	0,0018
Proc.motion	0,2547
Proc.dFilter	0,0154

Tabelle 4.14: Ergebnis des Profilers (Schritt 6)

Schritt 7:

Im nächsten Schritt wurden Änderungen hinsichtlich der Gewichtungsfaktoren *wx* und *wy* vorgenommen. Diese wurden bisher bei jedem Methodenaufruf neu berechnet, obwohl das bei gleich bleibender Bildgröße nicht nötig ist. Daher wurde die Berechnung in den gleichen Bereich wie *Util.getXmatYmat* verschoben, d.h. es wird nun erst geprüft, ob sich die Größe des Bildes geändert hat, bevor die Faktoren angepasst werden.

Einer weiteren Änderung wurde die Methode *Proc.lsAffine* unterzogen. Es handelt sich bei der Änderung um fünf Variablen, die zu Beginn jeder Iteration der inneren Schleife, mit entsprechenden Werten belegt, und bei den Berechnungen der Summen eingesetzt werden. Somit müssen die jeweiligen Multiplikationen nur einmal berechnet werden.

```
double xacc = 0.0;
double yacc = 0.0;
double upval = 0.0;
double vpval = 0.0;
double acc = 0.0;
for (y=yMin;y<yMax;y=y+samfac){
   for (x=xMin;x<xMax;x=x+samfac){
        acc = accept[y][x];
        xacc = xmat[y][x] * acc;
        yacc = ymat[y][x] * acc;
        upval = up[y][x];
        vpval = vp[y][x];
        xx += xacc * xmat[y][x];
        xy += xacc * ymat[y][x];
        yy += yacc * ymat[y][x];
        x1 += xacc;
        y1 += yacc;
        xu += xacc * upval;
        yu += yacc * upval;
        u  += acc * upval;
        xv += xacc * vpval;
        yv += yacc * vpval;
        v  += acc * vpval;
```

Listing 4.10: Änderungen in *Proc.lsAffine*

Um die Methode *Proc.dFilter* weiter zu beschleunigen, wurde sie weiter umstrukturiert, und gänzlich auf den Einsatz in diesem Plugin angepasst. Daher sind das *filter*-Array, die beiden Summenvariablen *sumx* und *sumy*, sowie die Variable *p* zur Zwischenspeicherung der Grauwerte herausgenommen worden.

Die jeweiligen Ableitungen werden nun in einer Zeile berechnet, und direkt an die Felder *I_x* bzw. *I_y* übergeben.

```
       int x, y, pos;
       for(y=yMin; y<=yMax; y++){
          pos = y*sizeX;
          for(x=xMin; x<=xMax; x++){
             I_x[y][x] = 0.5*((0xFF & imPixels[pos+x+1])-(0xFF & imPixels[pos+x-
1]));
             I_y[y][x] = 0.5*((0xFF & imPixels[(sizeX+pos+x)])-
                              (0xFF & imPixels[pos+x-sizeX]));
          }
       }
```

Listing 4.11: Umgestellte *dFilter*- Methode

Diese Optimierungen erzielen einen Performancegewinn von 3,57 %.

	Komplette Zeit im ms	**Zeit pro Bildpaar in ms**	**Bilder pro Sekunde**
Zeitmessung	841	28,03	35,71

Tabelle 4.15: Ergebnis der Messung mit *System.nanoTime* (Schritt 7)

Methode	**Durchschnittliche Dauer in Sekunden**
Proc.lsAffine	0,0016
Proc.motion	0,2519
Proc.dFilter	0,0004

Tabelle 4.16: Ergebnis des Profilers (Schritt 7)

4.2.3 Die Performance in unoptimiertem Zustand (ext)

Die ersten Messungen für die Methode *Extend.ext* ergaben die folgenden Ergebnisse:

	Komplette Zeit im ms	**Zeit pro Bildpaar in ms**	**Bilder pro Sekunde**
Zeitmessung	2443	81,43	12,28

Tabelle 4.17: Ergebnis der Messung mit *System.nanoTime*

Methode	**Durchschnittliche Dauer in Sekunden**
Extend.ext	6,6428

Tabelle 4.18: Ergebnis des Profilers

4.2.4 Optimierungen an der Methode ext

Schritt 1:
Zu Beginn wurden alle Schleifen, wie auch schon in den obigen Methoden, durch eine Bounding Box eingeschränkt.

Die Methode *copyBits*, zur Vervollständigung der *X_Area*, wurde außerdem durch eine schnellere, iterative Variante ersetzt. Den größten Performancegewinn erzielte jedoch der Austausch der (Jama-) Matrizenmethode *times* durch eine inline-gesetzte Multiplikationsschleife, die komplett ohne Objekte der Matrix-Klasse realisiert wurde.

Vorher:
```
w.X_Area.copyBits(w.maskp, 0, 0, Blitter.XOR);
```

Nachher:
```
for (int y=yMin;y<yMax;y++){
   for (int x=xMin;x<xMax;x++){
      if (maskpPixels[x+y*mx]==1){
         xAreaPixels[x+y*mx]=0;
      }
    }
 }
```

Listing 4.12: Austausch der *copyBits*- Methode

Vorher:
```
rp = w.t.times(temp);
```

Nachher:
```
for (int i = 0; i < 3; i++) {
   for (int j = 0; j < 1; j++) {
      rp[i][j] = 0.0;
      for (int z = 0; z < 3; z++) {
         rp[i][j] += w.tA[i][z] * temp[z][j];
      }
   }
}
```

Listing 4.13: Austausch der *times*- Methode

In der Schleife zur bilinearen Schätzung wurde, wie in *Proc.motion*, ebenfalls *Math.floor* durch einen int-cast ersetzt. Auch wurde die *inMask*-Methode inline gesetzt.

Durch diese Maßnahmen wurde ein großer Geschwindigkeitsgewinn von 333,88 % erzielt, was auch durch das Ergebnis des Profilers bescheinigt wird.

	Komplette Zeit im ms	**Zeit pro Bildpaar in ms**	**Bilder pro Sekunde**
Zeitmessung	563	18,77	53,28

Tabelle 4.19: Ergebnis der Messung mit *System.nanoTime* (Schritt 1)

Methode	**Durchschnittliche Dauer in Sekunden**
Extend.ext	0,2248

Tabelle 4.20: Ergebnis des Profilers (Schritt 1)

Schritt 2:
In einem zweiten Schritt wurden nun alle noch verwendeten Matrix-Objekte durch schnellere Arrays ersetzt.

Vorher:
```
if (xArea.get(x, y)==1){
   xvec[count] = w.xmat.get(y, x);
   yvec[count] = w.ymat.get(y, x);
   count++;
}
```

Nachher:
```
if (xAreaPixels[x+pos]==1){
   xvec[count] = w.xmat[y][x];
   yvec[count] = w.ymat[y][x];
   count++;
}
```

Listing 4.14: Austausch der Jama-Matrizen durch Arrays

Das Ergebnis dieser Änderung ist ein Gewinn von 11,26%.

	Komplette Zeit im ms	Zeit pro Bildpaar in ms	Bilder pro Sekunde
Zeitmessung	506	16,87	59,28

Tabelle 4.21: Ergebnis der Messung mit *System.nanoTime* (Schritt 2)

Methode	Durchschnittliche Dauer in Sekunden
Extend.ext	0,0579

Tabelle 4.22: Ergebnis des Profilers (Schritt 2)

Schritt 3:
Die nächsten Änderungen betrafen die hier noch verwendeten *getPixelValue*- und *putPixelValue*-Methoden der Bildprozessoren. Sie wurden durch den Zugriff auf das eindimensionale Pixelarray ersetzt. Betroffen sind hiervon die Bildprozessoren von *X_Area*, der Maske *w.maskp*, des Bildes *newIm* und des Panoramabildes *w.panoim*.

Vorher:
```
w.maskp.putPixelValue((int)yvec[k], (int)xvec[k], 1);
```

Nachher:
```
maskpPixels[(int)yvec[k]*mx+(int)xvec[k]] =  1;
```

Listing 4.15: Austausch der Pixel - Zugriffsmethoden

Diese Änderungen ergaben laut Messung +10,26 % Geschwindigkeitszuwachs.

	Komplette Zeit im ms	Zeit pro Bildpaar in ms	Bilder pro Sekunde
Zeitmessung	459	15,30	65,36

Tabelle 4.23: Ergebnis der Messung mit *System.nanoTime* (Schritt 3)

Methode	Durchschnittliche Dauer in Sekunden
Extend.ext	0,0232

Tabelle 4.24: Ergebnis des Profilers (Schritt 3)

Schritt 4:

In Schritt 4 wurde durch die Variablen *cntxMinox* bzw. *cntyMinoy* versucht, die sehr häufig (einmal pro darzustellendem Pixel) auftretenden Subraktionen des Offsets zu unterbinden.

```
Vorher:
double x = (double)(xvec[k]-w.cntx-w.ox);
double y = (double)(yvec[k]-w.cnty-w.oy);

Nachher:
double cntxMinox = w.cntx-w.ox;
double cntyMinoy = w.cnty-w.oy;
(…)
x = (double)(xvec[k]-cntxMinox);
y = (double)(yvec[k]-cntyMinoy);
(…)
```

Listing 4.16: Einsatz von *cntxMinox* und *cntyMinoy*

Der Performancegewinn fiel bei dieser Änderung allerdings nur sehr gering aus (2%).

	Komplette Zeit im ms	Zeit pro Bildpaar in ms	Bilder pro Sekunde
Zeitmessung	450	15,00	66,67

Tabelle 4.25: Ergebnis der Messung mit *System.nanoTime* (Schritt 4)

Methode	Durchschnittliche Dauer in Sekunden
Extend.ext	0,0183

Tabelle 4.26: Ergebnis des Profilers (Schritt 4)

Schritt 5:

Im letzten Schritt wurde die, vorher bereits inline-gesetzte Methode zur Matrizenmultiplikation(Listing 4.13), nun noch einmal verändert. Die Werte für *rp1* und *rp2* werden nun komplett ohne Schleifen berechnet. Zusätzlich werden bei dieser Variante noch Arrayzugriffe eingespart.

```
rp1 = w.tA[0][0] * x;
rp1 += w.tA[0][1] * y;
rp1 += w.tA[0][2];
rp2 = w.tA[1][0] * x;
rp2 += w.tA[1][1] * y;
rp2 += w.tA[1][2];
```

Listing 4.17: Ersatz der Methode zur Matrizenmultiplikation

Die Performance konnte durch diesen Schritt um weitere 42,45% gesteigert werden.

	Komplette Zeit im ms	Zeit pro Bildpaar in ms	Bilder pro Sekunde
Zeitmessung	316	10,53	94,97

Tabelle 4.27: Ergebnis der Messung mit *System.nanoTime* (Schritt 5)

Methode	Durchschnittliche Dauer in Sekunden
Extend.ext	0,0141

Tabelle 4.28: Ergebnis des Profilers (Schritt 5)

4.3 Exkurs: Auswirkungen der Eingabeparameter auf Performance und Genauigkeit

Nachdem der Quellcode nun optimiert wurde, soll in diesem Unterkapitel kurz auf die verschiedenen Eingabeparameter, und ihr Einfluss auf Performance und Genauigkeit des Ergebnisses, eingegangen werden. Hierfür werden die Parameterwerte schrittweise verändert, und die Ergebnisse anschließend in Diagrammen dargestellt. Nach diesen Betrachtungen schließt ein kleines Fazit das Kapitel ab.
Hierbei werden als default-Werte SAMFAC = 1, ITERMAX = 20 und EPS_A = 0,15 verwendet.

4.3.1 Der Parameter SAMFAC

Mit diesem Parameter wird festgelegt, wie viele Pixel zur Berechnung des globalen Bewegungsfeldes verwendet werden sollen. Ein Wert von 1 bedeutet, dass jeder Pixel in der Bildmaske zur Berechnung herangezogen wird. Ist der Wert entsprechend höher, wird nur jeder n-te Pixel in x und y-Richtung verwendet. Das bedeutet, dass die ursprünglich zur Verfügung stehende Anzahl an Pixel um den Faktor n^{2} abnimmt.
Wie in Abbildung 4.1 zu sehen ist, hat die Reduzierung der Pixelanzahl bis zu einem SAMFAC-Wert von 3 nur sehr geringe Auswirkungen auf die erreichte Genauigkeit. Erst ab einem Wert von 4 ist eine deutliche Abnahme zu verzeichnen.

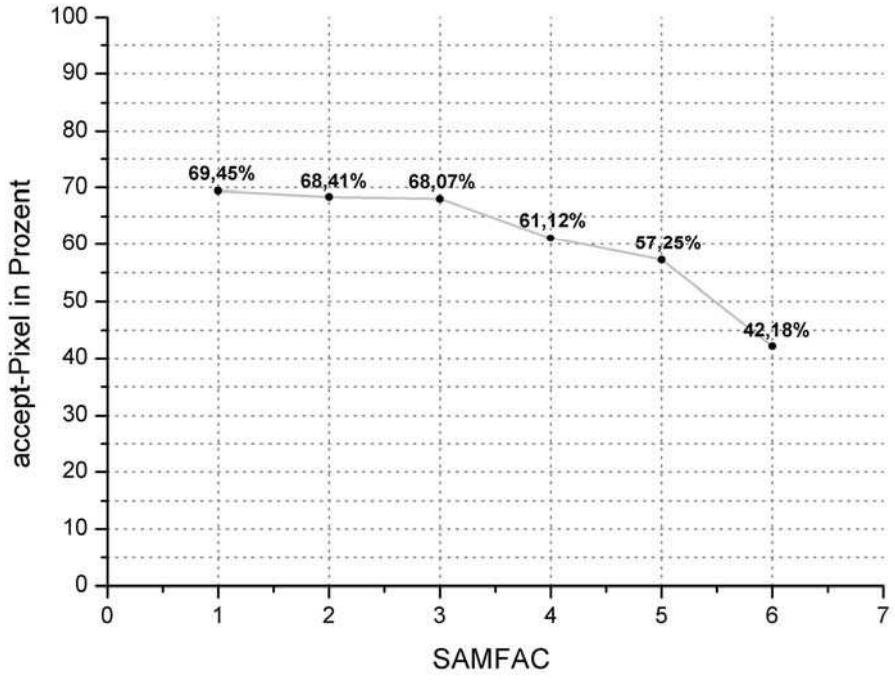


Abbildung 4.1: Die Genauigkeit bei verschiedenen SAMFAC-Werten

Auf die Performance hat der SAMFAC-Wert ebenfalls starken Einfluss. So steigt beispielsweise die Geschwindigkeit bei SAMFAC = 3 um 321,46%.

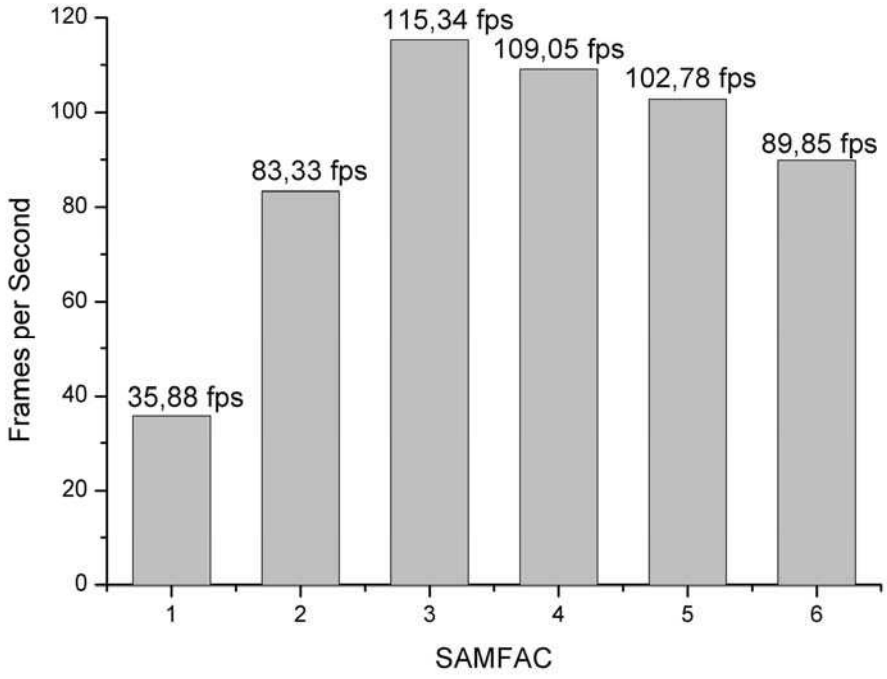


Abbildung 4.2: Die erreichte Performance bei verschiedenen SAMFAC-Werten

Ab einem gewissen Punkt (hier: SAMFAC = 4) verringert sich der Geschwindigkeits-zuwachs allerdings wieder. Dies liegt an der, oben gezeigten, abnehmenden Genauigkeit bei steigendem SAMFAC-Wert, da der Algorithmus nun mehrere Iterationen benötigt, um das Bewegungsfeld zu schätzen. Die Abbruchbedinung *delta_a<EPS_A* wird erst, wenn überhaupt, sehr viel später erfüllt. Die Summe der benötigten Iterationen für die gesamte Bildfolge, steigt bei einem SAMFAC-Wert von 6 beispielsweise auf 425, statt der ursprünglichen 172.

4.3.2 Der Parameter ITERMAX

ITERMAX ist ein weiterer Parameter, der Genauigkeit und Performance beeinflusst. Über diesen Wert wird, wie in Kapitel 3 bereits erwähnt, die maximale Anzahl der Iterationen zur Berechnung des Bewegungsfeldes festgelegt.
Wie aus dem folgenden Diagramm zu erkennen ist, sinkt die Genauigkeit (angegeben durch die durchschnittliche Anzahl der accept-Pixel) wenn als Maximalwert weniger als 20 Iterationen verwendet werden.
Wird die Anzahl der maximalen Iterationen auf über 20 erhöht, ändert sich das Ergebnis hingegen nicht. Der Algorithmus benötigt also, bei der hier verwendeten Bildfolge, zu keinem Zeitpunkt mehr als 20 Wiederholungen.

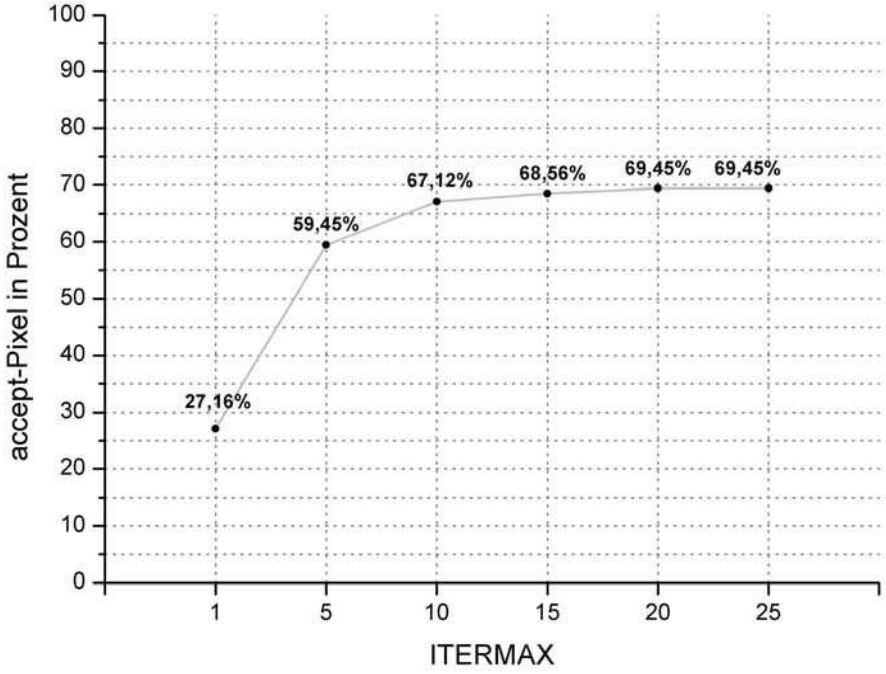


Abbildung 4.3: Die Genauigkeit bei verschiedenen Werten von ITERMAX

Da der Algorithmus für jedes Bildpaar durchschnittlich nur 5,7 Wiederholungen benötigt, wirkt sich eine Halbierung des Wertes von 20 auf 10 nur gering auf die erlangte Genauigkeit aus.

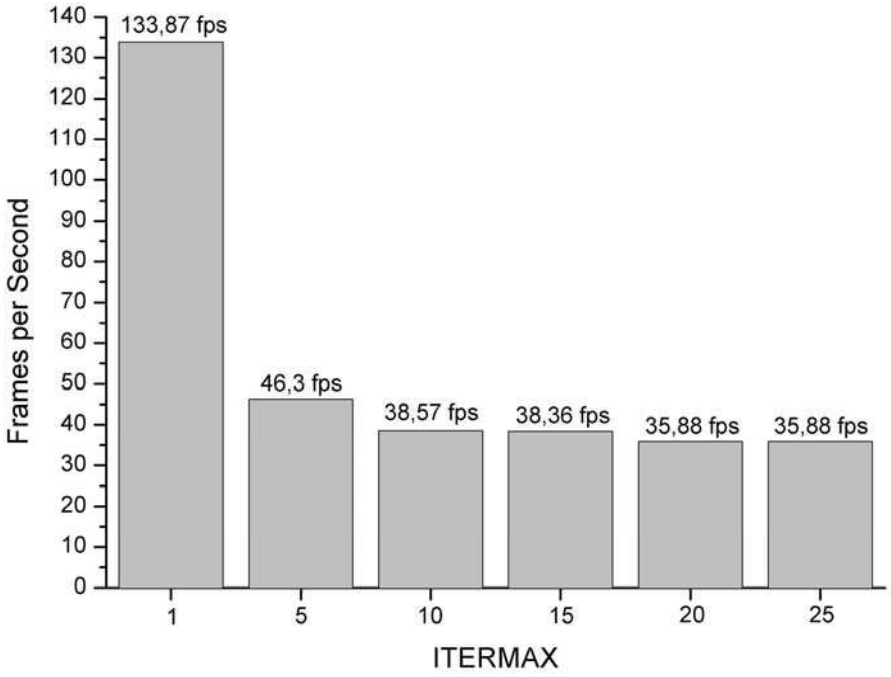


Abbildung 4.4: Die erreichte Performance bei verschiedenen Werten von ITERMAX

Eine Halbierung der Iterationen, von 20 auf 10, hat somit auch auf die Performance nur geringe Auswirkungen. Erst wenn die Anzahl der Wiederholungen unter den durchschnittlichen Wert gesetzt werden, ist bei beiden Diagrammen eine starke Änderung zu verzeichnen.

4.3.3 Der Parameter EPS_A

Dieser Parameter gibt an, wie groß die Änderung in der globalen Motion, von einer Iteration zur nächsten, sein darf, um die Abbruchbedingung *delta_a<EPS_A* zu erfüllen. Hierbei führt ein größerer Wert zu einem früheren Abbruch, und umgekehrt.
Wie auch in den Unterkapiteln zuvor, sollen durch die Diagramme die erreichte Genauigkeit und Performance bei verschiedenen Parameterwerten aufgezeigt werden.

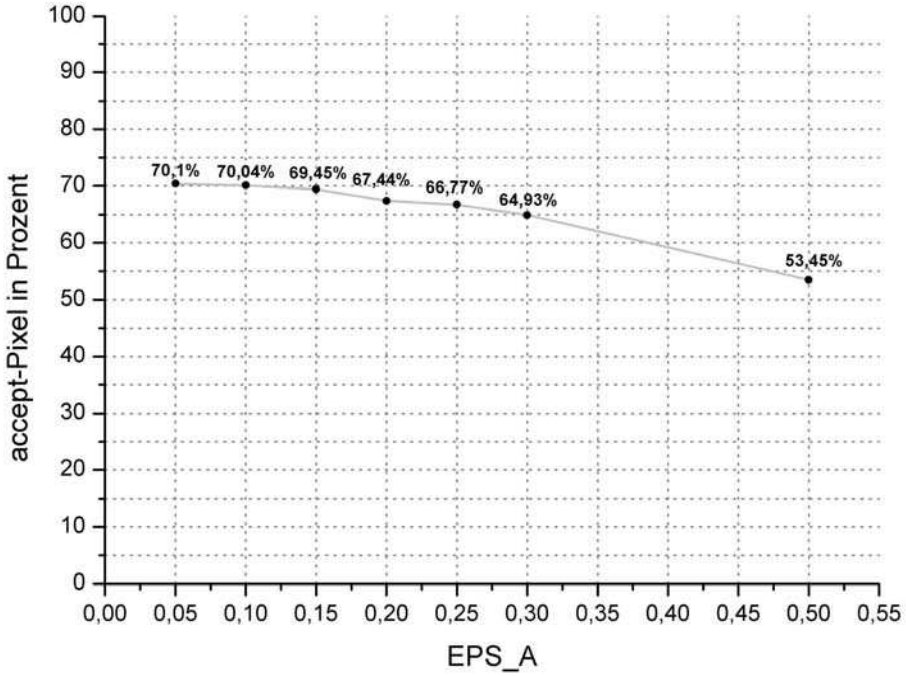


Abbildung 4.5: Die erreichte Genauigkeit bei verschiedenen EPS_A-Werten

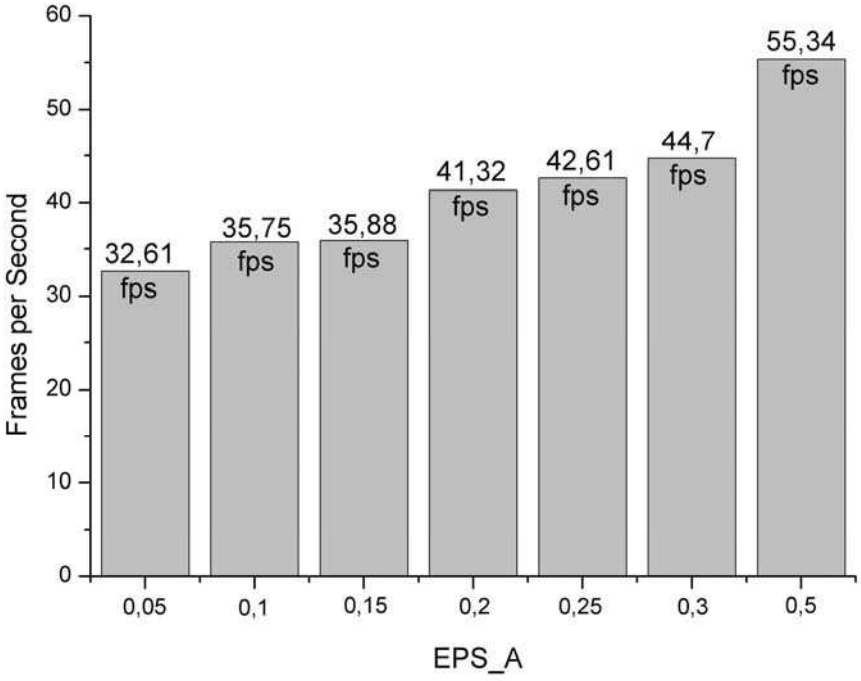


Abbildung 4.6: Die erreichte Performance bei verschiedenen EPS_A-Werten

4.3.4 Fazit

Die Betrachtungen zeigen noch einmal deutlich was im Grunde zu erwarten war: Parameterwerte im Nahbereich der default-Werte beeinflussen die Genauigkeit nur wenig. Somit kann durch ein Feintuning aller Parameter, abgestimmt auf die jeweilige Charakteristik der Bildfolge, die Performance weiter erhöht werden, ohne dabei das sichtbare Ergebnis zu verschlechtern.

Abbildung 4.7: Eine ungenaue Stelle im Panorama

Allerdings können auch solche geringen Änderungen der Parameter das Ergebnis an schwierigen Stellen (also an Stellen, an denen der Algorithmus mehr Iterationen benötigt) verschlechtern, was stark von der jeweiligen Bildfolge abhängt. In Abbildung 4.7 ist so eine Stelle in dem zusammengesetzten Panoramabild zu sehen. Der Algorithmus benötigt hier deutlich mehr Wiederholungen, da auf eine Skalierung eine Translation folgt, und somit die grobe Schätzung aus dem vorherigen Bildpaar unpassend ist (siehe Kapitel 2.4).

4.4 Ergebnis der Optimierung

Die Optimierungsschritte an der Methode *Proc.motion* ergeben insgesamt eine Beschleunigung um den Faktor 22,04, bzw. eine Steigerung der Performance um 2204,32 %.
Aus Abbildung 4.8 ist zu entnehmen, dass vor allem die Schritte 4 und 6 für diese Steigerung verantwortlich sind, wobei hier insbesondere der Austausch der Jama-Methoden in Schritt 4, und der erweiterte Einsatz der Bounding Box in Schritt 6 zu nennen ist.

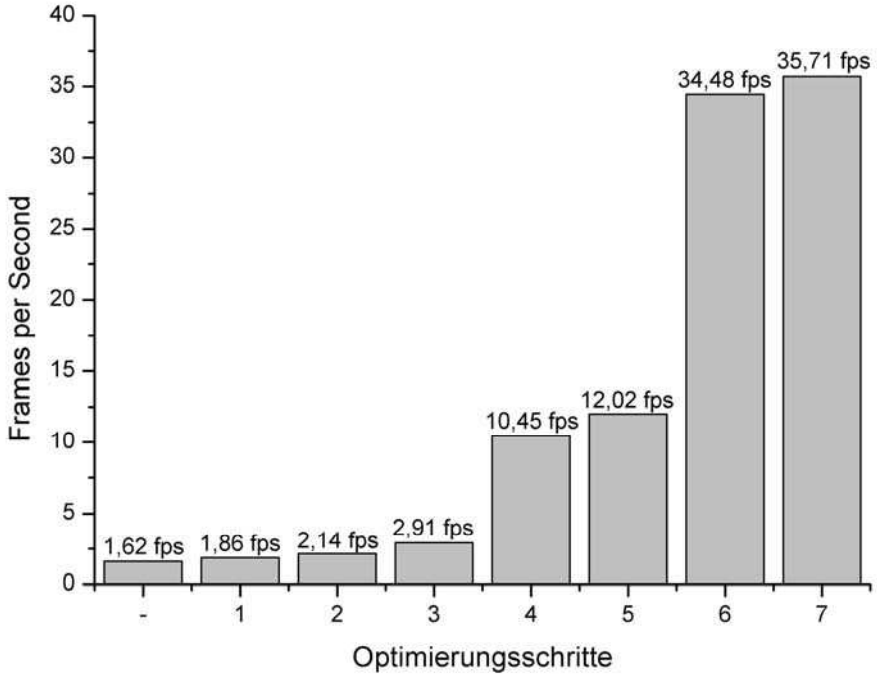


Abbildung 4.8: Auflistung der Performance nach jedem Optimierungsschritt (*Proc.motion*)

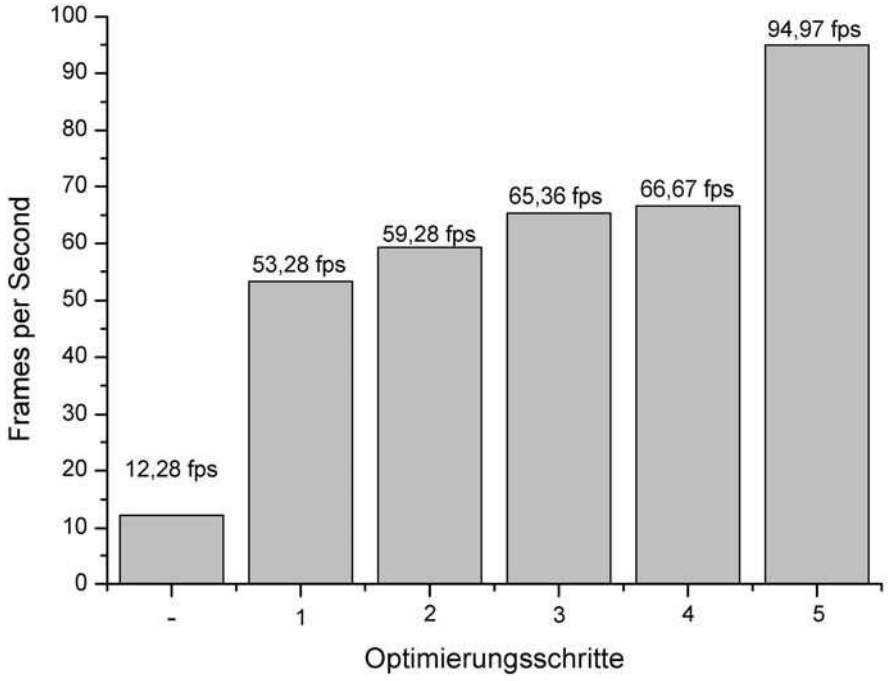


Abbildung 4.9: Auflistung der Performance nach jedem Optimierungsschritt (*Extend.ext*)

Bei Optimierung der Methode *Extend.ext* ist es vor allem der erste Schritt, der einen großen Performancesprung verursacht. Hier ist unter anderem die inline-gesetzte, eigene Methode zur Matrizenmultiplikation zu erwähnen, die für einen Großteil des Gewinnes verantwortlich ist. Aber auch durch die weiteren Optimierungen in Schritt 6 ist ein großer Anstieg der Geschwindigkeit zu verzeichnen.
Insgesamt beläuft sich hier der Zuwachs an Geschwindigkeit, durch alle Optimierungsschritte, auf 673,37 %.

Alle Messungen wurden mit Version 1.5 des JDK durchgeführt. Um abschließend, nach den Messungen, noch zu sehen, wie stark sich ein Wechsel auf die, derzeit, aktuelle Version 1.6 des JDK in der Performance auswirkt, wurde hierfür noch eine einzelne Messung mit *System.nanoTime* vorgenommen.

	Komplette Zeit im ms	**Zeit pro Bildpaar in ms**	**Bilder pro Sekunde**
Zeitmessung	592	19,74	50,66

Tabelle 4.29: Ergebnis der Messung mit *System.nanoTime*, unter Verwendung von Vers. 1.6 des JDK

Bei einem Vergleich der Tabellen 4.15 und 4.29 zeigt sich, dass durch einen Wechsel auf die neue Version des JDK, ein Performancegewinn von 41,89% zu erzielen ist.

Kapitel 5

Fazit

6.1 Zusammenfassung

Im Zuge dieser Arbeit wurde aus einer bestehenden, nicht echtzeitfähigen Matlab-Implementierung, ein echtzeitfähiges, ausbaubares ImageJ-Plugin realisiert. Im Vordergrund stand, neben der reinen Implementierung in Java, die Optimierung der Performance. Hierdurch wurde, im Vergleich zur ersten, unoptimierten Version, eine Beschleunigung um den Faktor 22 erzielt. Die Optimierungen wurden in mehreren Schritten während, sowie nach der Implementierungsphase vollzogen. Die ausführliche Auflistung der einzelnen Optimierungsschritte, und die Gegenüberstellungen der alten und neuen Codeteile ermöglichen es, jeden dieser Schritte genau nachzuvollziehen.
Um die Einarbeitung für zukünftige Arbeiten, die auf dieser aufbauen, zu erleichtern, sind die wichtigsten Methoden dieses Plugins, anhand der jeweiligen Codelistings im Text, sehr detailliert beschrieben und erklärt worden.
Zusätzlich wurde eine Methode zur Ausgabe eines farbigen Panoramabildes hinzugefügt. Somit ist es möglich, zwischen der Ausgabe als Grau- oder Farbbild zu wählen. [Auch die Anzeige eines Differenzbildes im jeweiligen Überlappbereich mit dem Panorama, ist als weiterer Punkt zu nennen.]

6.2 Ausblick

Die Aufgabenstellung dieser Bachelorarbeit wurde mit diesem Plugin gelöst, es gibt allerdings durchaus noch einige Erweiterungs- und Verbesserungsmöglichkeiten, die das Plugin leistungsfähiger, performanter und bedienbarer machen würden:

- Der Einsatz von Multithreading ermöglicht es, die rechenintensiven Methoden zur Berechnungen des globalen Bewegungsfeldes, und zur Bildung des Panoramas in einem eigenen Thread laufen zu lassen. Durch diese Abkopplung vom Programmthread wäre es möglich, beispielsweise die GUI des Plugins auch noch während des Rechenvorgangs zu bedienen.

  Ebenso kann die Priorität des Rechenthreads erhöht werden, um sicherzustellen dass die verfügbare Rechenleistung nicht durch andere Programme reduziert wird.

- Als weiterer Schritt wäre es denkbar, die Threads auf mehrere Prozessoren zu verteilen, und somit die Vorteile von heutigen Multicore-Prozessoren zu erlangen.

- Die Erweiterung, von der hier verwendeten affinen, auf die projektive Transformation ermöglicht es, mehrere Kamerabewegungen abzubilden.

- Der Einsatz von Algorithmen, die robust auf Beleuchtungsänderungen reagieren. So können beispielsweise sogenannte Glanzlichter, durch Reflexionen des Endoskoplichts auf feuchten Organoberflächen, entstehen, und die Qualität des Bildes verschlechtern.

- Die Vermeidung von Bildverzeichnungen im Panoramabild, durch Verzeichnungskorrektur, wobei die Verzeichnungen durch die eingesetzte Kameraoptik entstehen.

- Um zukünftig auch aus ganzen Filmen ein Panorama bilden zu können, müsste das Plugin noch um eine Funktionalität erweitert werden, welche aus einem kompletten Film die einzelnen Bilder, zur weiteren Bearbeitung, in einen Image Stack lädt. Auch die Möglichkeit, den Vorgang der Mosaikbildung als einen Film speichern zu können, würde hier für eine Erweiterung der Funktionalität sprechen. Alternativ kann sich hier natürlich auch das Plugin-Konzept von ImageJ zunutze gemacht werden, indem bereits bestehende Plugins (z.B. AviReader bzw. AviWriter) für diese Aufgaben verwendet werden.

- Für den späteren Einsatz, während einer Endoskopie, ist es wichtig, dass die Software in der Lage ist, Schnitte im aktuellen Videostream automatisch zu erkennen und entsprechend zu reagieren. Wird ein solcher Schnitt erkannt, muss folglich der Aufbau eines neuen Bildmosaiks begonnen werden.

Schließlich ist noch davon auszugehen, dass bei der Performance, durch Einsatz der kommenden, neuen Version 7 des JDK, eine Steigerung, wie auch bei dem Wechsel von Version 5 auf 6 in dieser Arbeit, zu erreichen ist.

Literaturverzeichnis

[Bai03] *Bailer, Werner:* Writing ImageJ Plugins - A Tutorial, mtd.fhhagenberg.at/depot/imaging/imagej/ijtutorial.pdf

[BB06] *Burger, W. / Burge, M. J.:* Digitale Bildverarbeitung Eine Einführung mit Java und ImageJ, X media press, Springer Verlag, Berlin, 2006.

[BKS07] *Breiderhoff B., Konen W., Scholz M., Ein automatisiertes Verfahren zum Image-Mosaicing bei endoskopischen Videoaufnahmen*, Technical Report, Inst. for Informatics, FH Cologne, 2007

[Chr04] *Christadler I.*. Mehrgitterverfahren für die Berechnung des Optischen Flusses mit Nicht-Standardregularisierungen. Diploma thesis, Friedrich-Alexander-Universit¨at, 2004.

[FVL04] *Fischer B, Vaessen B, Lehmann TM, Spitzer K* Bildverbesserung endoskopischer Videosequenzen in Echtzeit, 2004

[HS81] *B.K.P. Horn and B.G. Schunk.* Determining Optical Flow. A.I. Memo 572, Masachusetts Institute of Technology, April 1981.

[Jac03] *Jacobs A.*, Mosaicing auf Szenen mit bewegten Objekten, Diplomarbeit, 2003

[Jäh05] Jähne, B.: Digitale Bildverarbeitung, Springer Verlag, Berlin, 2005.

[KK04] *Klein, Hans-Ulrich / Koop, Michael:* Image Mosaicing, Seminararbeit,Westfälische Wilhelms-Universität Münster, Institut für Informatik, Lehrstuhl Prof. Dr. Xiaoyi Jiang, 2004

[Kon06] *Konen W*, Optischer Fluss und Echtzeitvideobearbeitung, Technical Report, Inst. For Informatics, FH Cologne, 2006

[Kou99] *Kourogi M, Kurata T, Hoshino J, et al*.: Real-time image mosaicing from a video sequence. Procs ICIP99, vol. 4, 133-137, 1999.

[Neu05] *Neumann, B.*: Bildverarbeitung für Einsteiger, Springer Verlag, Berlin, 2005

[Rob03] *Robinson JA*: A Simplex-Based Projective Transform Estimator. Procs Visual Information Engineering (VIE), Guildford, 290-293, July, 2003.

[SLH06] *Seshamani S, Lau WW, Hager GD*: Real-Time Endoscopic Mosaicking. MICCAI'2006, Lecture Notes in Computer Science, vol. 4190, Springer, 355-363 2006.

[Sze94] *Szeliski R*: Image Mosaicing for Tele-Reality Applications. TR 94/2, Digital Equipment Corporation, Cambridge Research Lab, June 1994.

A Anhang

Im Folgenden wird der Quellcode, aller relevanten Klassen, des Plugins aufgelistet.
Die CD-ROM, die dieser Arbeit beigelegt wurde, enthält die kompletten Klassen, sowie die generierten Javadoc Dateien.

A.1 Quellcode der Klasse Control

```
import ij.plugin.*;
import ij.io.*;
import ij.ImagePlus;
import ij.ImageStack;
import java.awt.*;
import java.text.*;
import ij.process.*;

public class Control_ implements PlugIn{
   private static ImageStack imgStack, imgStackColor;
   private static PanoAffine w;
   private static double[][] maski, maskn;
   private static int sizeX, sizeY;
/*-------------------------------------------------------------------------------------------*/

   public void run(String args){
      GUI wnd = new GUI();
      wnd.setLocation(200,200);
      wnd.setSize(800, 700);
      wnd.setVisible(true);
   }
/*-------------------------------------------------------------------------------------------*/
   /*
    * this method contains all important method invocations, to build
    * the panorama
    * IN:  THRESH, SAMFAC, ITERMAX, EPS_A, imgSeq, outputRGB, diffImg, xArea, doExt,
                delay
    * OUT: nothing
    * @param THRESH parameter for Proc.motion
    * @param SAMFAC parameter for Proc.motion
    * @param ITERMAX parameter for Proc.motion
    * @param EPS_A parameter for Proc.motion
    * @param imgSeq which image sequence was chosen?
    * @param outputRGB create a colored panorama yes/no
    * @param diffImg create a difference Image yes/no
    * @param xArea draw the XArea next to the panorama
    * @param doExt extend the panorama automatically yes/no
    * @param delay delay the graphical output yes/no
    */
   public static void process(int THRESH, int SAMFAC, int ITERMAX, double EPS_A, int
                              imgSeq,boolean outputRGB, boolean diffImg, boolean xArea,
                              boolean doExt, boolean delay){
      ImagePlus img = new ImagePlus();
      int imgCount = 0;
      String imgName = "";
      String imgFmt = "";
      if (imgSeq==1){
         imgCount = 18;
         imgName = "jroberts";
         imgFmt = ".TIF";
      } else
      if (imgSeq==2){
         imgCount = 31;
         imgName = "jrobertB";
```

```
   imgFmt = ".bmp";
} else
if (imgSeq==3){
   imgCount = 31;
   imgName = "3750-6665-2110";
   imgFmt = ".bmp";
}
if (imgSeq==4){
   imgCount = 6;
   imgName = "PDVD_";
   imgFmt = ".TIF";
}
Opener imgLoader = new Opener();
ImagePlus maskd = imgLoader.openImage("images\\"+imgName+"-maskp"+imgFmt);
sizeY = maskd.getHeight();
sizeX = maskd.getWidth();
maski = new double[sizeY][sizeX];
maskn = new double[sizeY][sizeX];
PreProc.getMasks(maskd.getProcessor(), maski, maskn);
imgStack = new ImageStack(maskd.getWidth(), maskd.getHeight());
img = imgLoader.openImage("images\\"+imgName+"000"+imgFmt);
   imgStackColor = new ImageStack(maskd.getWidth(), maskd.getHeight());
for (int i=0;i<imgCount;i++){
   if (i<10)img = imgLoader.openImage("images\\"+imgName+"00"+i+imgFmt);
      else img = imgLoader.openImage("images\\"+imgName+"0"+i+imgFmt);
   if (outputRGB) imgStackColor.addSlice("Image"+i, img.getProcessor());
   imgStack.addSlice("Image"+i,img.getProcessor().convertToByte(false));
}

Var par = new Var();
par.thresh = THRESH;
par.itermax = ITERMAX;
par.samfac = SAMFAC;
par.eps_a = EPS_A;

Proc.sizeXim = maskd.getWidth();
Proc.sizeYim = maskd.getHeight();
Rectangle rect = Util.bb(maski);
init(Proc.sizeXim, Proc.sizeYim, rect);

if (outputRGB) w = new PanoAffine(imgStackColor.getProcessor(1), maskn, rect);
   else w = new PanoAffine(imgStack.getProcessor(1), maskn, rect);
Proc.xmat = new double[sizeY][sizeX];
Proc.ymat = new double[sizeY][sizeX];
double[] cntXY = new double[2];
cntXY = Util.getXmatYmat(maskn, Proc.xmat, Proc.ymat);
Proc.cntx = cntXY[0];
Proc.cnty = cntXY[1];
for (int y=0;y<sizeY;y++){
   for (int x=0;x<sizeX;x++){
      Proc.xmat[y][x] -= Proc.cntx;
      Proc.ymat[y][x] -= Proc.cnty;
   }
}
for (int y=0;y<maskd.getHeight();y++){
   for (int x=0;x<maskd.getWidth();x++){
      Proc.wx = Proc.wx + Proc.xmat[y][x] * Proc.xmat[y][x] * maskn[y][x];
      Proc.wy = Proc.wy + Proc.ymat[y][x] * Proc.ymat[y][x] * maskn[y][x];
   }
}
Proc.wx = Proc.wx/Util.sum(maskn);
Proc.wy = Proc.wy/Util.sum(maskn);
Proc.weight_a = new double[1][6];
Proc.weight_a[0][0] = Proc.wx;
Proc.weight_a[0][1] = Proc.wy;
Proc.weight_a[0][2] = 1;
Proc.weight_a[0][3] = Proc.wx;
Proc.weight_a[0][4] = Proc.wy;
Proc.weight_a[0][5] = 1;
long timeComplete = 0;
timeComplete = System.nanoTime();
for (int x = 0; x<imgStack.getSize()-1; x++){
   Proc.motion(imgStack.getProcessor(x+1), imgStack.getProcessor(x+2),
                              rect, par, maski, maskn);
   //consoleDataOutput(x);
   if (outputRGB) Extend.ext(w, imgStackColor.getProcessor(x+2), Proc.a,
                                          outputRGB, diffImg, doExt);
      else Extend.ext(w, imgStack.getProcessor(x+2), Proc.a,
                                         outputRGB, diffImg, doExt);
   GUI.paintPan(w.panoim, 10, 10);
   if (xArea) GUI.paintXarea(w.X_Area, 400, 10);
   if (diffImg) GUI.paintDiff(w.Xerr, 400, 10);
   if (delay)
      try{Thread.sleep(500);}catch(InterruptedException e){};
```

```
        }
        timeComplete = System.nanoTime() - timeComplete;
        timeComplete = timeComplete/1000000;
        //consoleTimeOutput(timeComplete, imgCount);
        GUI.GUITimeOutput(timeComplete, imgCount);
        GUI.canvas.setGraphics(GUI.canvas.getGraphics(), w.panoim.createImage(), 10, 10);
        if (xArea) GUI.canvas.setGraphics2(GUI.canvas.getGraphics(),
                            w.X_Area.createImage(), 400, 10);
        if (diffImg) GUI.canvas.setGraphics2(GUI.canvas.getGraphics(),
                           w.Xerr.createImage(), 400, 10);
    }
/*-------------------------------------------------------------------------------------------*/
    private static void consoleDataOutput(int x){
        String format = "#0.0000";
        DecimalFormat df = new DecimalFormat(format);
        System.out.print("\n\n\nRefFrame: "+imgStack.getSliceLabel(x+1)+"  CurrFrame:
                                  "+imgStack.getSliceLabel(x+2));
        System.out.print("\t delta_a: "+ df.format(Proc.delta_a));
        System.out.print("\t Acceptpixel: "+ Util.sum(Proc.accept)+"
                   "+"("+df.format((Util.sum(Proc.accept) / Util.sum(maskn))*100)+" %)\n");
        System.out.print("-------------------------------------------------------------------------
-------------------------------------------------------------");
        System.out.println();
           for (int y=0;y<=5;y++){
              System.out.print(df.format(Proc.a[0][y])+"\t\t\t" );
           }
    }
/*-------------------------------------------------------------------------------------------*/
    private static void consoleTimeOutput(long timeComplete, int imgCount){
        String format = "#0.00";
        DecimalFormat df = new DecimalFormat(format);
        System.out.println("\n\n");
        System.out.println("Zeit komplett: "+ timeComplete);
        System.out.println("Zeit pro Bild: "+ df.format((double)timeComplete/(imgCount-1)));
        System.out.println("fps: "+ df.format((double)1000/(timeComplete/(imgCount-1))));
    }
/*-------------------------------------------------------------------------------------------*/
    private static void init(int sizeX, int sizeY, Rectangle rect){
        Proc.I_t = new double[sizeY][sizeX];
        Proc.accept = new double[sizeY][sizeX];
        Proc.I_x = new double[sizeY][sizeX];
        Proc.I_y = new double[sizeY][sizeX];
        Proc.uc = new double[sizeY][sizeX];
        Proc.vc = new double[sizeY][sizeX];
        Proc.up = new double[sizeY][sizeX];
        Proc.vp = new double[sizeY][sizeX];
        Proc.a = new double[1][6];
        Proc.a_temp = new double[6][1];
        Extend.rect = null;
        Extend.firstRun = true;
        //Extend.rect = rect;
        Extend.xMin = (int)rect.getMinX();
        Extend.xMax = (int)rect.getMinX()+rect.width;
        Extend.yMin = (int)rect.getMinY();
        Extend.yMax = (int)rect.getMinY()+rect.height;
        //GUI.can.setForeground(GUI.blk);
    }
```

A.2 Quellcode der Klasse Proc

```
import ij.process.*;
import java.awt.*;
import Jama.Matrix;

public final class Proc {
    public static double    cntx, cnty, sum_maskn, delta_a, wx, wy;
    public static int   sizeXim, sizeYim, xMin, xMax, yMin, yMax, cnt;
    private static byte[]   imPixels, newImPixels;
    public static double[][]   uc, vc, up, vp, accept, I_x, I_y, I_t, a, a_prev, weight_a,
                                   a_temp, xmat, ymat;
    /**
     * this method calculates the pseudo and compensated motion, and
     * returns the affine estimate a
     * IN:  imRef, imCurr, maski, maskn, rect, par
     * OUT: a
     * @param imRef reference image
     * @param imCurr current image
     * @param maski image mask for the reference frame
     * @param maskn image mask for the current frame
     * @param rect the bounding box of the mask
```

```
 * @param par an object, which contains important parameters
 */
public static void motion(ImageProcessor imRef, ImageProcessor imCurr, Rectangle rect,
                          Var par,double[][] maski, double[][] maskn){
   int x, y, floorY, floorX;
   double  dx, dy, Ip, val1, val2, val3, val4, delta, rpx, rpy;
   final int THRESH   = par.thresh;
   final int SAMFAC   = par.samfac;
   final int ITERMAX  = par.itermax;
   final double EPS_A = par.eps_a;
   imPixels = (byte[]) imRef.getPixels();
   newImPixels = (byte[]) imCurr.getPixels();
   if ((sizeXim!=imCurr.getWidth())||(sizeYim!=imCurr.getHeight())){
      double[] cntXY = new double[2];
      cntXY = Util.getXmatYmat(maskn, xmat, ymat);
      cntx = cntXY[0];
      cnty = cntXY[1];
      for (y=0;y<sizeYim;y++){
         for (x=0;x<sizeXim;x++){
            xmat[y][x] -= cntx;
            ymat[y][x] -= cnty;
         }
      }
      for (y=0;y<sizeYim;y++){
         for (x=0;x<sizeXim;x++){
            wx += xmat[y][x] * xmat[y][x] * maskn[y][x];
            wy += ymat[y][x] * ymat[y][x] * maskn[y][x];
         }
      }
      wx = wx/sum_maskn;
      wy = wy/sum_maskn;
      weight_a[0][0] = wx;
      weight_a[0][1] = wy;
      weight_a[0][2] = 1;
      weight_a[0][3] = wx;
      weight_a[0][4] = wy;
      weight_a[0][5] = 1;
      sizeXim = imRef.getWidth();
      sizeYim = imRef.getHeight();
   }
   if (accept==null){accept = new double[sizeYim][sizeXim];}
   if (a_temp==null){a_temp = new double[6][1];}
   Proc.I_t = new double[sizeYim][sizeXim];
   a_prev = new double[1][6];
   xMin = (int)rect.getMinX();
   xMax = (int)rect.getMinX()+rect.width;
   yMin = (int)rect.getMinY();
   yMax = (int)rect.getMinY()+rect.height;
   dFilter(imPixels, I_x, I_y, xMin, xMax, yMin, yMax, sizeXim);
   double a0 = a[0][0];
   double a1 = a[0][1];
   double a2 = a[0][2];
   double a3 = a[0][3];
   double a4 = a[0][4];
   double a5 = a[0][5];
   for (y=yMin;y<yMax;y=y+SAMFAC){
      for (x=xMin;x<xMax;x=x+SAMFAC){
         uc[y][x] = (xmat[y][x]*a0) + (ymat[y][x]*a1) + a2;
         vc[y][x] = (xmat[y][x]*a3) + (ymat[y][x]*a4) + a5;
      }
   }
   delta_a = 100;
   for (cnt=1; cnt<=ITERMAX; cnt++){
   for (y=yMin;y<yMax;y=y+SAMFAC){
      for (x=xMin;x<xMax;x=x+SAMFAC){
         if (I_x[y][x]!=0 && I_y[y][x]!=0 && maski[y][x]!=0){
            rpx = x+1 + uc[y][x];
            rpy = y+1 + vc[y][x];
            floorX = (int)rpx;
            floorY = (int)rpy;
            if ((!(floorX<0 || floorX>=sizeXim))&&(!(floorY<0 ||
                   floorY>=sizeYim))&&(!(maskn[floorY][floorX]== 0))){
               dx = rpx - floorX;
               dy = rpy - floorY;
               val1 = 0xFF & newImPixels[(floorY-1)*sizeXim+floorX+1-1];
               val2 = 0xFF & newImPixels[(floorY-1)*sizeXim+floorX-1];
               val3 = 0xFF & newImPixels[(floorY+1-1)*sizeXim+floorX+1-1];
               val4 = 0xFF & newImPixels[(floorY+1-1)*sizeXim+floorX-1];
               Ip = ( dx*val1 + (1-dx)*val2 ) * (1-dy) +(dx*val3+(1-dx)*val4) * dy;
               I_t[y][x] = Ip - (0xFF & imPixels[y*sizeXim+x]);
            }
         }
         up[y][x] = ((I_t[y][x]*-1)/(I_x[y][x]+1E-6))+uc[y][x];
         vp[y][x] = ((I_t[y][x]*-1)/(I_y[y][x]+1E-6))+vc[y][x];
```

```
            accept[y][x] = 0;
         } // end for(x
      } // end for(y
      for (y=yMin;y<yMax;y=y+SAMFAC){
         for (x=xMin;x<xMax;x=x+SAMFAC){
            if (I_x[y][x]!=0 && I_y[y][x]!=0 && maski[y][x]!=0){
               rpx = x+1 + up[y][x];
               rpy = y+1 + vp[y][x];
               floorX = (int)rpx;
               floorY = (int)rpy;
               if ((!(floorX<0 || floorX>=sizeXim))&&(!(floorY<0 ||
                     floorY>=sizeYim))&&(!(maskn[floorY][floorX]== 0))){
                  dx = rpx - floorX;
                  dy = rpy - floorY;
                  val1 = 0xFF & newImPixels[(floorY-1)*sizeXim+floorX+1-1];
                  val2 = 0xFF & newImPixels[(floorY-1)*sizeXim+floorX-1];
                  val3 = 0xFF & newImPixels[(floorY+1-1)*sizeXim+floorX+1-1];
                  val4 = 0xFF & newImPixels[(floorY+1-1)*sizeXim+floorX-1];
                  Ip = (dx*val1 + (1-dx)*val2) * (1-dy) + (dx*val3+(1-dx)*val4) * dy;
                  delta = Ip - (0xFF & imPixels[y*sizeXim+x]);
                  if (Math.abs(delta)<THRESH){
                     accept[y][x] = 1;
                  }
               }
            }
         } //end for(x
      } //end for(y
      for (int i=0; i<=5; i++){
         a_prev[0][i] = a[0][i];
      }
      lsAffine(SAMFAC, xmat, ymat, accept, up, vp, a, xMin, xMax, yMin, yMax);
      for (int i=0; i<=5; i++){
         a_temp[i][0] = a[0][i]-a_prev[0][i];
      }
      for (int i=0; i<=5; i++){
         a_temp[i][0] = a_temp[i][0]*a_temp[i][0];//Math.pow(a_temp[i][0], 2);
      }
      delta_a = Math.sqrt(Util.sum(Util.arrTimes(weight_a, a_temp)));
      a0 = a[0][0];
      a1 = a[0][1];
      a2 = a[0][2];
      a3 = a[0][3];
      a4 = a[0][4];
      a5 = a[0][5];
      for (y=yMin;y<yMax;y=y+SAMFAC){
         for (x=xMin;x<xMax;x=x+SAMFAC){
            uc[y][x] = (xmat[y][x]*a0) + (ymat[y][x]*a1) + a2;
            vc[y][x] = (xmat[y][x]*a3) + (ymat[y][x]*a4) + a5;
         }
      }
      if (delta_a<EPS_A){
         //System.out.println(cnt);
         break;
      }
   }//end itermax
}
/*-------------------------------------------------------------------------------------------*/

   /**
    * least square estimate of affine parameters
    */
   private static void lsAffine(int samfac, double[][] xmat, double[][] ymat, double[][]
                                accept,double[][] up, double[][]vp, double[][] a, int
                                xMin, int xMax, int yMin, int yMax){
      int x, y;
      double s;
      Matrix rhs_u = new Matrix(3,1);
      Matrix rhs_v = new Matrix(3,1);
      Matrix M = new Matrix(3,3);
      Matrix a_u = new Matrix(3,1);
      Matrix a_v = new Matrix(3,1);
      double xx = 0.0;
      double xy = 0.0;
      double yy = 0.0;
      double x1 = 0.0;
      double y1 = 0.0;
      double xu = 0.0;
      double yu = 0.0;
      double u = 0.0;
      double xv = 0.0;
      double yv = 0.0;
      double v = 0.0;
      double xacc = 0.0;
      double yacc = 0.0;
```

```
        double upval = 0.0;
        double vpval = 0.0;
        double acc = 0.0;
        s  = Util.sum(accept);
        for (y=yMin;y<yMax;y=y+samfac){
            for (x=xMin;x<xMax;x=x+samfac){
                acc = accept[y][x];
                xacc = xmat[y][x] * acc;
                yacc = ymat[y][x] * acc;
                upval = up[y][x];
                vpval = vp[y][x];
                xx += xacc * xmat[y][x];
                xy += xacc * ymat[y][x];
                yy += yacc * ymat[y][x];
                x1 += xacc;
                y1 += yacc;
                xu += xacc * upval;
                yu += yacc * upval;
                u  += acc * upval;
                xv += xacc * vpval;
                yv += yacc * vpval;
                v  += acc * vpval;
            }
        }
        rhs_u.set(0, 0, xu);
        rhs_u.set(1, 0, yu);
        rhs_u.set(2, 0, u);
        rhs_v.set(0, 0, xv);
        rhs_v.set(1, 0, yv);
        rhs_v.set(2, 0, v);
        M.set(0, 0, xx);
        M.set(1, 0, xy);
        M.set(2, 0, x1);
        M.set(0, 1, xy);
        M.set(1, 1, yy);
        M.set(2, 1, y1);
        M.set(0, 2, x1);
        M.set(1, 2, y1);
        M.set(2, 2, s);
        a_u = M.solve(rhs_u);
        a_v = M.solve(rhs_v);
        for (int i=0; i<=2; i++){
            a[0][i] = a_u.get(i, 0);
            a[0][i+3] = a_v.get(i, 0);
        }
    }
/*-------------------------------------------------------------------------------------------*/

    private static void dFilter(byte[] imPixels, double[][] I_x, double[][] I_y,
                  int xMin, int xMax, int yMin, int yMax, int sizeX){
        int x, y, pos;
        for(y=yMin; y<=yMax; y++){
            pos = y*sizeX;
            for(x=xMin; x<=xMax; x++){
                I_x[y][x] = 0.5*((0xFF & imPixels[pos+x+1])-(0xFF & imPixels[pos+x-1]));
                I_y[y][x] = 0.5*((0xFF & imPixels[(sizeX+pos+x)])-(0xFF &imPixels[pos+x-
                                                                                  sizeX]));
            }
        }
    }
```

A.3 Quellcode der Klasse Extend

```
import Jama.Matrix;
import ij.process.*;

import java.awt.*;
import ij.ImageStack;

public class Extend {
    private static byte[] xAreaPixels, maskpPixels, newImPixels, panoimPixels;
    public static int xMin, xMax, yMin, yMax;
    private static int xTempLeft, xTempRight, yTempTop, yTempDown, val1, val2, val3, val4;
    private static Matrix A, id;
    public static boolean firstRun = true;
    public static Rectangle rect;
    private static double dx, dy, Ip, rpx, rpy;// val1, val2, val3, val4;
    /**
     * adds the new frame newIm to the existing panorama. If necessary (bounding box of
     * w.maskp too close to mosaic image border), extend all mosaic images by an
```

```
 * extra rim, such that the image mosaic does not reach the border.
 * IN:  w, newIm, a, outputRGB, diffImg, doExt
 * OUT: w
 * @param w PanoAffine object
 * @param newIm current frame
 * @param a parameter vector of affine transform
 * @param outputRGB create a colored panorama yes/no
 * @param diffImg create a difference Image yes/no
 * @param doExt extend the panorama automatically yes/no
 */
public static void ext(PanoAffine w, ImageProcessor newIm, double[][] a, boolean
                          outputRGB, boolean diffImg, boolean doExt){
   //w.X_Area.multiply(0);
   xTempLeft = (int) w.cntx;
   xTempRight = (int) w.cntx;
   yTempTop = (int) w.cnty;
   yTempDown = (int) w.cnty;
   if (firstRun){
      rect = new Rectangle();
      id = new Matrix(3,3, 0);
      A = new Matrix(3,3, 0);
      id.set(0, 0, 1);
      id.set(1, 1, 1);
      id.set(2, 2, 1);
      firstRun = false;
   } else if (doExt) bb_extend(w, rect);

   int floorX, floorY;
   int my = w.maskp.getHeight();
   int mx = w.maskp.getWidth();

   int nx = newIm.getWidth();
   //int ny = newIm2.getHeight();

   if (!outputRGB){
      newImPixels = (byte[]) newIm.getPixels();
      panoimPixels = (byte[])w.panoim.getPixels();
   }
   xAreaPixels = (byte[])w.X_Area.getPixels();
   maskpPixels = (byte[])w.maskp.getPixels();
   A.set(0, 0, a[0][0]);
   A.set(0, 1, a[0][1]);
   A.set(0, 2, a[0][2]);
   A.set(1, 0, a[0][3]);
   A.set(1, 1, a[0][4]);
   A.set(1, 2, a[0][5]);
   A.set(2, 0, 0);
   A.set(2, 1, 0);
   A.set(2, 2, 0);
   w.t = id.plus(A).times(w.t);
   w.tA = w.t.getArray();
   int pos;
   int mx20 = mx*20;
   for (int y=yMin;y<yMax;y++){
      pos = y*mx;
      for (int x=xMin;x<xMax;x++){
         if (maskpPixels[x+pos]==1){
         //if (w.maskp.getPixelValue(x, y)==1){
            if (!(x-20<0)&&!(y-20<0)){xAreaPixels[(x-20)+pos-mx20]= 1;}
            if (!(x+20>=mx)&&!(y+20>=my)) {xAreaPixels[(x+20)+pos+mx20]= 1;}
            if (!(y-20<0)&&!(x+20>=mx)){xAreaPixels[(x+20)+pos-mx20]= 1;}
            if (!(x-20<0)&&!(y+20>=my)){xAreaPixels[(x-20)+pos+mx20]= 1;}
            /*w.X_Area.putPixel(x+20, y+20, 1);
            w.X_Area.putPixel(x-20, y-20, 1);
            w.X_Area.putPixel(x+20, y-20, 1);
            w.X_Area.putPixel(x-20, y+20, 1);*/
            if ((x-20<xTempLeft)&&(x-20>0)){xTempLeft = x-20;}
            if ((x+20>xTempRight)&&(x+20<mx)){xTempRight =x+20;}
            if ((y-20<yTempTop)&&(y-20>0)){yTempTop = y-20;}
            if ((y+20>yTempDown)&&(y+20<my)){yTempDown = y+20;}
         }
      }
   }
   /*w.X_Area.setColor(1);
   w.X_Area.drawRect(xTempLeft, yTempTop, xTempRight-xTempLeft+1, yTempDown-yTempTop+1);*/
   rect.setBounds(xTempLeft, yTempTop, (xTempRight-xTempLeft)+1, (yTempDown-yTempTop)+1);
   xMin = (int)rect.getMinX();
   xMax = (int)rect.getMinX()+rect.width;
   yMin = (int)rect.getMinY();
   yMax = (int)rect.getMinY()+rect.height;
   for (int y=yMin;y<yMax;y++){
      pos = y*mx;
      for (int x=xMin;x<xMax;x++){
         if (maskpPixels[x+pos]==1){
```

```
                xAreaPixels[x+pos]=0;
            }
        }
    }

    int xArCount = rect.height*rect.width;
    double[] xvec = new double[xArCount];
    double[] yvec = new double[xArCount];
    int count = 0;
    for (int y=yMin;y<yMax;y++){
        pos = y*mx;
        for (int x=xMin;x<xMax;x++){
            if (xAreaPixels[x+pos]==1){
                xvec[count] = w.xmat[y][x];
                yvec[count] = w.ymat[y][x];
                count++;
            }
        }
    }
    double x,y, rp1, rp2;
    double cntxMinox = w.cntx+w.ox;
    double cntyMinoy = w.cnty+w.oy;

    if (outputRGB)interpolateRGB(w, xvec, yvec, newIm, cntxMinox, cntyMinoy,
                                 w.cntx, w.cnty, xArCount, mx, my);
    else {
        for (int k=0; k<xArCount; k++){
            x = xvec[k]-cntxMinox;
            y = yvec[k]-cntyMinoy;
            rp1 = w.tA[0][0] * x;
            rp1 += w.tA[0][1] * y;
            rp1 += w.tA[0][2];
            rp2 = w.tA[1][0] * x;
            rp2 += w.tA[1][1] * y;
            rp2 += w.tA[1][2];
            rpx = rp1 + w.cntx;
            rpy = rp2 + w.cnty;
            floorX = (int) rpx;
            floorY = (int) rpy;
            if ((!(floorX<0 || floorX>=mx))&&(!(floorY<0 ||
                    floorY>=my))&&(!(w.maskn[floorY][floorX]== 0))){
                dx = rpx - floorX;
                dy = rpy - floorY;
                val1 = 0xFF & newImPixels[(floorY-1)*nx+floorX+1-1];
                val2 = 0xFF & newImPixels[(floorY-1)*nx+floorX-1];
                val3 = 0xFF&newImPixels[(floorY+1-1)*nx+floorX+1-1];
                val4 = 0xFF & newImPixels[(floorY+1-1)*nx+floorX-1];
                Ip = ((dx*val1)+((1-dx)*val2)) * (1-dy) + ((dx*val3)+(1-dx)*val4) * dy;
                panoimPixels[(int)yvec[k]*mx+(int)xvec[k]] = (byte)(int)Ip;
                maskpPixels[(int)yvec[k]*mx+(int)xvec[k]] =  1;
                if (diffImg)
                    w.Xerr.putPixelValue((int)xvec[k],(int)yvec[k],
                                Math.abs(newIm.getPixelValue (floorX, floorY)-(int)Ip)+100);
            }
        }
    }
  }
/*-------------------------------------------------------------------------------------------*/
    private static void interpolateRGB(PanoAffine w, double[] xvec, double[] yvec,
                                       ImageProcessor newIm, double cntxMinox,
                                       double cntyMinoy, double cntx, double cnty, int
                                       xArCount, int mx, int my){
        int r, g, b;
        int [][] rgb = new int[4][3];
        int floorX, floorY;
        double rp1, rp2;
        for (int k=0; k<xArCount; k++){
            double x = xvec[k]-cntxMinox;
            double y = yvec[k]-cntyMinoy;
            rp1 = 0.0;
            rp2 = 0.0;
            rp1 += w.tA[0][0] * x;
            rp1 += w.tA[0][1] * y;
            rp1 += w.tA[0][2];
            rp2 += w.tA[1][0] * x;
            rp2 += w.tA[1][1] * y;
            rp2 += w.tA[1][2];
            rpx = rp1 + w.cntx;
            rpy = rp2 + w.cnty;
            floorX = (int) rpx;
            floorY = (int) rpy;
            if ((!(floorX<0 || floorX>=mx))&&(!(floorY<0 ||
                    floorY>=my))&&(!(w.maskn[floorY][floorX]== 0))){
                dx = rpx - floorX;
```

```
            dy = rpy - floorY;
            val1 = newIm.getPixel(floorX, floorY-1);
            rgb[0][0] = (val1 & 0xff0000) >> 16;
            rgb[0][1] = (val1 & 0x00ff00) >> 8;
            rgb[0][2] = (val1 & 0x0000ff);
            val2 = newIm.getPixel(floorX-1, floorY-1);
            rgb[1][0] = (val2 & 0xff0000) >> 16;
            rgb[1][1] = (val2 & 0x00ff00) >> 8;
            rgb[1][2] = (val2 & 0x0000ff);
            val3 = newIm.getPixel(floorX, floorY);
            rgb[2][0] = (val3 & 0xff0000) >> 16;
            rgb[2][1] = (val3 & 0x00ff00) >> 8;
            rgb[2][2] = (val3 & 0x0000ff);
            val4 = newIm.getPixel(floorX-1, floorY);
            rgb[3][0] = (val4 & 0xff0000) >> 16;
            rgb[3][1] = (val4 & 0x00ff00) >> 8;
            rgb[3][2] = (val4 & 0x0000ff);
            r = (int)(((dx*rgb[0][0])+((1-dx)*rgb[1][0])) * (1-dy) +
                      ((dx*rgb[2][0])+(1-dx)*rgb[3][0]) * dy);
            g = (int)(((dx*rgb[0][1])+((1-dx)*rgb[1][1])) * (1-dy) +
                      ((dx*rgb[2][1])+(1-dx)*rgb[3][1]) * dy);
            b = (int)(((dx*rgb[0][2])+((1-dx)*rgb[1][2])) * (1-dy) +
                     ((dx*rgb[2][2])+(1-dx)*rgb[3][2]) * dy);
            Ip = ((r & 0xff)<<16)|((g & 0xff)<<8)|b & 0xff;
            w.panoim.putPixel((int)xvec[k], (int)yvec[k], (int)Ip);
            maskpPixels[(int)yvec[k]*mx+(int)xvec[k]] =  1;
         }
      }
   }
/*-------------------------------------------------------------------------------------------*/
   /**
    * private helper function for ext
    * Check the bounding box of w.maskp: if it is too large, then extend
    * w.maskp, w.panoim, imagestack (and finally w.xmat, w.ymat, w.cntx, w.cnty)
    * @param w PanoAffine object
    * @param bb bounding box
    */
   private static void bb_extend(PanoAffine w, Rectangle bb){
      int sx = w.panoim.getWidth();
      int sy = w.panoim.getHeight();
      int nbord = 25;
      int ngrow = 25;
      boolean isgrown = false;

      if ((bb.getMinY()-0.5)<nbord){ //upper
         w.oy += ngrow;
         bb_grow(w, 0, ngrow, 0, ngrow);
         isgrown = true;
      }
      if ((sy-bb.getMaxY()+0.5)<nbord){ //lower
         bb_grow(w, 0, ngrow, 0, 0);
         isgrown = true;
      }
      if ((bb.getMinX()-0.5)<nbord){ //left
         w.ox += ngrow;
         bb_grow(w, ngrow, 0, ngrow, 0);
         isgrown = true;
      }
      if ((sx-(bb.getMaxX())+0.5)<nbord){ //right
         bb_grow(w, ngrow, 0, 0, 0);
         isgrown = true;
      }
      if (isgrown){
         double[][] maskpA = new
                        double[w.maskp.getHeight()][w.maskp.getWidth()];
         for (int y=0;y<w.maskp.getHeight();y++){
            for (int x=0;x<w.maskp.getWidth();x++){
               maskpA[y][x] = w.maskp.getPixel(x, y);
            }
         }
         w.xmat = new double [w.maskp.getHeight()][w.maskp.getWidth()];
         w.ymat = new double [w.maskp.getHeight()][w.maskp.getWidth()];
         double[] cntXY = new double[2];
         cntXY = Util.getXmatYmat(maskpA, w.xmat, w.ymat);
         //w.cntx = cntXY[0];
         //w.cnty = cntXY[1];
      }
      //return w;
   }
/*-------------------------------------------------------------------------------------------*/
```

```
    /**
     * private helper function for bb_extend
     * @param w PanoAffine object
     * @param bb bounding box
     * @param ngx the new ImageProcessors are bigger by this number of pixels in x- and y-
                       direction
     * @param ngy the new ImageProcessors are bigger by this number of pixels in x- and y-
                       direction
     * @param w_ngx offset in x-direction for ImageProcessors w.* and ImageStack
     * @param w_ngy offset in y-direction for ImageProcessors w.* and ImageStack
     */
    private static void bb_grow(PanoAffine w, int ngx, int ngy, int w_ngx, int w_ngy){
        ImageProcessor n_maskp, n_panoim, n_X_Area;
        int sx = w.panoim.getWidth();
        int sy = w.panoim.getHeight();
        double[][] n_wmaskn = new double[sy+ngy][sx+ngx];

        n_maskp = w.maskp.duplicate();
        n_maskp.multiply(0);
        n_maskp = n_maskp.resize(sx+ngx, sy+ngy);
        n_maskp.copyBits(w.maskp, w_ngx, w_ngy, Blitter.COPY);
        w.maskp = n_maskp;

        n_X_Area = w.X_Area.duplicate();
        n_X_Area.multiply(0);
        n_X_Area = n_X_Area.resize(sx+ngx, sy+ngy);
        n_X_Area.copyBits(w.X_Area, w_ngx, w_ngy, Blitter.COPY);
        w.X_Area = n_X_Area;

        n_panoim = w.panoim.duplicate();
        n_panoim.multiply(0);
        n_panoim = n_panoim.resize(sx+ngx, sy+ngy);
        n_panoim.copyBits(w.panoim, w_ngx, w_ngy, Blitter.COPY);
        w.panoim = n_panoim;

        for (int y=0;y<sy;y++){
            for (int x=0;x<sx;x++){
                n_wmaskn[y][x] = w.maskn[y][x];
            }
        }
        w.maskn = n_wmaskn;
        //return w;
    }
```

A.4 Quellcode der Klasse Util

```
}import java.awt.*;

public class Util {
    /**
     * this method creates two coordiante matrices, and
     * calculates the origin cntx and cnty
     * IN:  maskn, xmat, ymat
     * OUT: xmat, ymat, cntx, cnty
     * @param maskn this mask is needed for the calculation
     *           of xmat, ymat and cntx, cnty
     * @param xmat reference from xmat array
     * @param ymat reference from ymat array
     * @return cntXY an array, which contains cntx and cnty
     */
    public static double[] getXmatYmat(double[][] maskn, double[][] xmat, double[][] ymat){
        int sizeY = maskn.length;
        int sizeX = maskn[0].length;
        int sumMaskn;
        double cntx, cnty;
        double countsumXmat = 0.0;
        double countsumYmat = 0.0;
        for (int y=0;y<sizeY;y++){
            for (int x=0;x<sizeX;x++){
                xmat[y][x] = x;
                ymat[y][x] = y;
                countsumXmat += xmat[y][x] * maskn[y][x];
                countsumYmat += ymat[y][x] * maskn[y][x];
            }
        }
        sumMaskn = (int)sum(maskn);
        cntx = countsumXmat/sumMaskn;
```

```
        cnty = countsumYmat/sumMaskn;
        double[] cntXY = new double[2];
        cntXY[0] = cntx;
        cntXY[1] = cnty;
        return cntXY;
    }
/*-------------------------------------------------------------------------------------------*/
    /**
     * calculates the bounding box around a given mask, and
     * returns a rectangle
     * IN:  mask
     * OUT: rect
     * @param mask   the needed mask, for the calculation
     *               of the bounding box
     * @return rect the bounding box rectangle
     */
    public static Rectangle bb(double[][] mask){
        Rectangle rect = new Rectangle();
        int sizeX = mask[0].length;
        int sizeY = mask.length;
        int xTempLeft = sizeX/2;
        int xTempRight = sizeX/2;
        int yTempTop = sizeY/2;
        int yTempDown = sizeY/2;
        for (int y=0;y<sizeY;y++){
            for (int x=0;x<sizeX;x++){
                if (mask[y][x]==1){
                    if (x<xTempLeft){xTempLeft = x;}
                    if (x>xTempRight){xTempRight = x;}
                    if (y<yTempTop){yTempTop = y;}
                    if (y>yTempDown){yTempDown = y;}
                }
            }
        }
        rect.setBounds(xTempLeft, yTempTop, (xTempRight-xTempLeft)+1, (yTempDown-
                                yTempTop)+1);
        return rect;
    }
/*-------------------------------------------------------------------------------------------*/
    /**
     * sums up all pixels in a given mask
     * IN:  m
     * OUT: sum
     * @param m the mask
     * @return sum the sum of all mask-pixels
     */
    public static double sum(double[][] m){
        double sum = 0;
        int x, y, sizeX, sizeY;
        sizeX = m[0].length;
        sizeY = m.length;
        for (y=0;y<sizeY;y++){
            for (x=0;x<sizeX;x++){
                sum += m[y][x];
            }
        }
        return sum;
    }
/*-------------------------------------------------------------------------------------------*/
    /**
     * this method multiplicates two matrices and returns
     * the matrix product
     * IN:  a, b
     * OUT: c
     * @param a matrix a
     * @param b matrix b
     */
    public static double[][] arrTimes(double[][] a, double[][] b){
        int m = a.length; //Spaltenlänge
        int l = a[0].length; //Zeilenlänge;
        int n = b[0].length;
        double[][] c = new double[m][n];
        for (int i = 0; i < m; i++) {
          for (int j = 0; j < n; j++) {
            c[i][j] = 0.0;
            for (int k = 0; k < l; k++) {
              c[i][j] += a[i][k] * b[k][j];
            }
          }
        }
        return c;
    }
```

A.5 Quellcode der Klasse PreProc

```
import ij.process.Blitter;
import ij.process.ImageConverter;
import ij.process.ImageProcessor;
import ij.ImagePlus;

public class PreProc {
/*-----------------------------------------------------------------------------------------*/

   /**
    * creates standard masks for rectangular images.
    * the masks are an exact copy of the given mask, except for
    * the border pixels, where the calculated gradient is not valid
    * IN:  mask, maski, maskn
    * OUT: maski, maskn
    * @param mask image mask
    * @param maski a mask for the reference frame
    * @param maskn a mask for the current frame
    */
   public static void getMasksRect(ImageProcessor mask, double[][] maski, double[][]
                                       maskn){
      int x, y;
      int sizeX = mask.getWidth();
      int sizeY = mask.getHeight();
      for ( y=0;y<sizeY;y++){
         for (x=0;x<sizeX;x++){
            if (mask.getPixelValue(x, y)==1){
               maski[y][x] = 1;
               maskn[y][x] = 1;
            }
         }
      }
      for (x=0;x<sizeX;x++){
         maski[0][x] = 0;
         maski[sizeY-1][x] = 0;
         maskn[sizeY-1][x] = 0;
      }
      for (y=0;y<sizeY;y++){
         maski[y][0] = 0;
         maski[y][sizeX-1] = 0;
         maskn[y][sizeX-1] = 0;
      }
   }
/*-----------------------------------------------------------------------------------------*/

   /**
    * creates standard masks for non-rectangular regions.
    * the method derives smaller masks from the given mask, such
    * that it is possible to calculate the gradient at each "1" in
    * maski, and can make bilinear interpolation at each "1" in maskn
    * @param mask image mask
    * @param maski a mask for the reference frame
    * @param maskn a mask for the current frame
    */
   public static void getMasks(ImageProcessor mask, double[][] maski, double[][] maskn){
      int[][] structI = { {0,1,0}, {1,1,1}, {0,1,0} };
      int[][] structN = { {0,0,0}, {0,1,1}, {0,1,1} };
      int sizeY = mask.getHeight();
      int sizeX = mask.getWidth();
      for (int y=0;y<sizeY;y++){
         for (int x=0;x<sizeX;x++){
            if (mask.getPixelValue(x, y)==0){
               mask.putPixelValue(x, y, 255);
            }
            if (mask.getPixelValue(x, y)==1){
               mask.putPixelValue(x, y, 0);
            }
         }
      }
      ImageProcessor maskiIP = mask.duplicate();
      ImageProcessor masknIP = mask.duplicate();
      maskiIP = erode(maskiIP, structI);
      masknIP = erode(masknIP, structN);
      maskiIP = maskiIP.convertToByte(false);
      masknIP = masknIP.convertToByte(false);

      for (int y=0;y<sizeY;y++){
         for (int x=0;x<sizeX;x++){
            if (maskiIP.getPixel(x, y) == 255){
```

```
                maski[y][x] = 1;
            } else
                maski[y][x] = 0;
            if (masknIP.getPixel(x, y)==255){
                maskn[y][x] = 1;
            } else
                maskn[y][x] = 0;
        }
    }
    //System.out.println(Util.sum(Proc.maskn));
  }
/*-----------------------------------------------------------------------------------------*/

    private static ImageProcessor dilate(ImageProcessor I, int[][] H){
        int ic = (H[0].length-1)/2;
        int jc = (H.length-1)/2;
        ImageProcessor ip = I.createProcessor(I.getWidth(), I.getHeight());
        for (int j=0; j<H.length; j++){
            for (int i=0; i<H[j].length; i++){
                if (H[j][i] == 1){
                    ip.copyBits(I, i-ic, j-jc, Blitter.MIN);
                }
            }
        }
        I.copyBits(ip, 0, 0, Blitter.COPY);
        return I;
    }
/*-----------------------------------------------------------------------------------------*/

    private static ImageProcessor erode(ImageProcessor I, int[][] H){
        I.invert();
        I = dilate(I, H);
        //I.invert();
        return I;
    }
}
```

A.6 Quellcode der Klasse PanoAffine

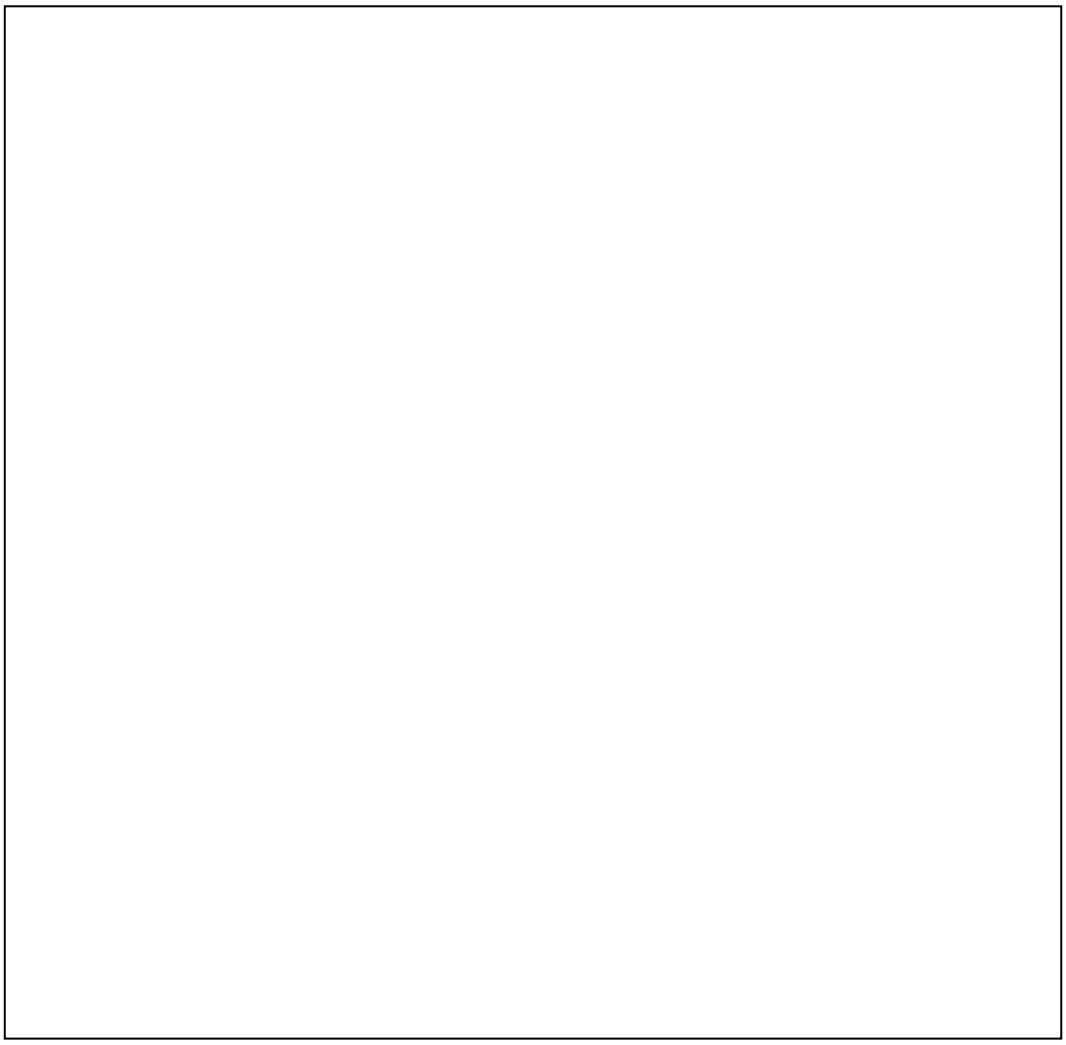
```
import Jama.Matrix;
import ij.process.*;
import java.awt.*;

public class PanoAffine {
    public int ox;
    public int oy;
    public  double cntx;
    public  double cnty;
    public double[][] maskn, tA, xmat, ymat;
    public Matrix t;
    public ImageProcessor maskp, panoim, X_Area, Xerr;

    /**
     * This constructor first calculates a proper offset w.ox, w.oy. Then it
     * initializes the panoramic image w.panoim by copying the first frame from the
     * ImageStack to the offset location.
     * It initializes w.maskp with mask (w.maskp may grow when more frames are added,
     * see Extend.java) and the transformation w.T with the identity transform
     * @param im reference frame
     * @param maskn the corresponding mask
     * @param rect the bounding box of the mask
     */
    public PanoAffine(ImageProcessor im, double[][] mask, Rectangle rect){
        int sx = im.getWidth();
        int sy = im.getHeight();
        int xMin = (int)rect.getMinX();
        int xMax = (int)rect.getMinX()+rect.width;
        int yMin = (int)rect.getMinY();
        int yMax = (int)rect.getMinY()+rect.height;
        this.ox = 0;
        this.oy = 0;
        xmat = new double[sy][sx];
        ymat = new double[sy][sx];
        double[] cntXY = new double[2];
        cntXY = Util.getXmatYmat(mask, xmat, ymat);
        cntx = cntXY[0];
        cnty = cntXY[1];
        panoim = im.duplicate();
        X_Area = im.convertToByte(false).duplicate();
```

```
    X_Area.multiply(0);
    Xerr = X_Area.duplicate();
    maskp = im.convertToByte(false).duplicate();
    maskp.multiply(0);
    byte[] maskpPixels = (byte[]) maskp.getPixels();
    maskn = new double[sy][sx];
    for (int y=yMin;y<yMax;y++){
       for (int x=xMin;x<xMax;x++){
          maskpPixels[y*sx+x] = (byte)(int)mask[y][x];
          if (mask[y][x]==1){
             maskn[y][x] = 1;
          }
       }
    }
    t = new Matrix(3, 3, 0);
    t.set(0, 0, 1);
    t.set(1, 1, 1);
    t.set(2, 2, 1);
}
```

CD-Rom



Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen teilen noch keiner anderen Prüfungsbehörde vorgelegen.

_____	_____
Ort, Datum	Martin Naderi