Analyse und Implementation echtzeitfähiger Rauschfilter für digitalisierte Videosignale

Studienarbeit Projekt 3

Andreas Romeyke
andreas.romeyke@web.de

Telekommunikationsinformatik (TKI)

Deutsche Telekom AG – Fachhochschule Leipzig

Analyse und Implementation echtzeitfähiger Rauschfilter für digitalisierte Videosignale

Andreas Romeyke

2003/04

Betreut durch Profn. Dr. Ing. Ines Rennert

In diesem Dokument werden die gängigsten im Spatialbereich arbeitenden Filter zur Verringerung von binären und additiven Gaußschen Rauschen in digitalisiertem Videomaterial hinsichtlich ihrer Fähigkeit der subjektiven Qualitätsverbesserung und der Echtzeitfähigkeit analysiert. Dazu werden die Wirkungen der Filter auf verschiedene Testbilder beurteilt und der Rechenaufwand abgeschätzt. Desweiteren wird ausgehend davon ein eigener Filter entwickelt.

Inhaltsverzeichnis

Danksagung

Ich möchte mich an dieser Stelle bei meiner Betreuerin Frau *Professorin Dr. Ines Rennert* danken, ohne die diese Projektarbeit nicht zustande gekommen wäre.

Mein Dank geht genauso an *Benjamin Baumgarten*, *Matthias John*, *Tilo Pinkert* und *Heiko Stamer*, die mir hilfreiche Tipps gegeben oder einfach nur zugehört haben. Ohne Frau *Gisela Biermann* von der Bibliothek wäre der Weg deutlich steiniger geworden.

Diese Arbeit wäre ohne `Linux` und LaTeX nicht in der Form möglich gewesen. Hilfreich begleiteten mich der Editor `jedit`, sowie die Kleber `Perl` und `Bash`, der Compiler `GCC`, die Bildverarbeitungsbibliothek `netpbm` und das Anzeigeprogramm `ImageMagick`. Ohne den Initialfunken `XawTV` von *Gerd Knorr* hätte es das Projekt wahrscheinlich nicht gegeben.

1 Einführung

1.1 Motivation

Durch die rasante Entwicklung der Computertechnologie und durch die hohe Leistungsfähigkeit von Standard-PCs wird das heimische Wohnzimmer immer mehr von Multimedia-PCs dominiert. So kann neben der Musikwiedergabe der PC sinnvoll zur Videoprojektion, -bearbeitung, wie auch als Fernsehersatz verwendet werden. Durch den Einbau von preisgünstigen TV-Karten kann in Verbindung mit Linux[1] als Betriebssystem und einer geeigneten Software, zB. XawTV[2] oder TVTime[3] ein sehr guter Ersatz für ein Fernsehgerät geschaffen werden. Durch die Verwendung von PCs als Fernseher ist es daher auch möglich, neben der digitalen Videoaufzeichnung auch gestörte Fernsehsignale mit geeigneten Filtern aufzubereiten.

Empfangsbedingte Störungen, wie binäres Rauschen, auch als "Salz- und Pfeffer"-Störungen bei schlechtem Satellitenempfang bekannt, sowie additives Gaussches Rauschen bedingt durch niedrige Empfangspegel, sollen hier im Weiteren Gegenstand der Projektarbeit sein.

Dabei soll gezeigt werden, welche einfachen Filter für die Elimination von bestimmten Störungen geeignet sind. Dabei spielt neben der Qualität die Ausführungsgeschwindigkeit eine Rolle, damit solche Filter für den Einsatz in Fernsehsoftware verwendet werden können.

Die Anwendung der untersuchten Filter beschränkt sich dabei nicht nur auf den Fernsehempfang. Störungen, vor allem das zu betrachtende additive Gaußsche wie auch das binäre Impulsrauschen treten bei allen bildverarbeitenden Verfahren auf. Als Beispiele seien an dieser Stelle die lasergestützte Lichtmikroskopie in der Zellbiologie, sowie technische Video- und Bildüberwachungssysteme zu nennen.

[1] `http://www.linux.de`
[2] Knorr (2003)
[3] *TVTime-Entwicklerteam* (2003)

1.2 Psychovisuelle Betrachtungen

Da bei verrauschtem Videomaterial in der Regel das Originalsignal nicht vorliegt, und somit nur schwierig der Übertragungskanal modelliert werden kann, ist es die Hauptaufgabe der in dieser Projektarbeit zu untersuchenden Filter das gestörte Videosignal so aufzubereiten, daß es die optimal mögliche Qualität für einen Menschen hinsichtlich

- Informationsgehalt und
- Ergonomie

bietet.

Sie sollen also dem Betrachter ein entspanntes Anschauen möglichst ohne zusätzlichen Informationsverlust erlauben.

Diese Anforderungen werden im wesentlichen durch die folgenden Eigenschaften des menschlichen Sehens bestimmt.

Nach Haidmayer (2003) gilt für die Wahrnehmung generell das Weber-Fechner-Gesetz:

> Die subjektive Empfindungsintensität nimmt proportional mit dem Logarithmus der Reizstärke zu

Dies erklärt die verstärkte Empfindlichkeit des menschlichen Sehens für Impulsrauschen gegenüber dem additiven Gaußschem Rauschen. Damit haben aber auch Helligkeitsänderungen in hellen Bildbereichen weniger Bedeutung für die Betrachtungsqualität, als in dunkleren Bildbereichen. Diese Folgerung des Weber-Fechner-Gesetz kann somit für die Beschleunigung von Filteroperationen genutzt werden.

Ein weiterer wichtiger Aspekt des menschlichen Sehens ist, daß die Anpassung der Empfindlichkeit der Farbrezeptoren bei Bildverdunklung weitaus länger dauert, als die Anpassung der Stäbchen (Haidmayer, 2003). Da im Rahmen der vorliegenden Arbeit nur Filter zur Entrauschung im Spatialraum[4] ohne Berücksichtigung der zeitlichen Dimension untersucht werden, wird auf die Bewertung dieses Effektes verzichtet. Für die Untersuchung von Filtern und Kompressionsalgorithmen für digitales Video könnte dieser Effekt in späteren Arbeiten untersucht und ausgenutzt werden.

Interessant ist auch der Effekt der zur Erzeugung von Nachbildern (Zwisler, 1998) führt, er entspricht einer Mittelwertfilterung in der zeitlichen Abfolge der einzel-

[4] sprich: Einzelbild

nen Frames, so daß man bei der Implementierung unter Umständen auf eine zeitliche Mittelung in der Filterung verzichten kann.

1.3 Testbilder

1.3.1 Einführung

Die Qualität der Filter kann nicht allein durch eine rein mathematische Abhandlung, etwa durch Abschätzung des Signal-Rauschleistungspegels (Peak Signal Noise Ratio, kurz PSNR, sh. Gleichung 2.1 Seite 11) beurteilt werden. Aus diesem Grunde wurden für jedes im Folgenden betrachtete Filter verschiedene Testbilder zur Beurteilung herangezogen. Die Bildtafeln mit den einzelnen in den nächsten Abschnitten vorgestellten Bildern und die entsprechend Filterergebnisse finden sich im Anhang D.

1.3.2 Real verrauschte Bilder

So galt es auf der einen Seite natürliche Bilder heranzuziehen, die die optische Wirkung der einzelnen Filter unter real auftretenden Bedingungen testen. Beispiele hierfür sind die Bilder D.13(b), D.3(b), D.5(d), D.8(b), D.10(d) und D.23(b).

Es wurden auch Beispiele von stark verrauschtem Material aus Videoquellen benötigt, um feststellen zu können, wie die Filter mit extremen Bedingungen fertig werden. Dazu wurden die Bilder D.30(d) und D.33(b) verwendet.

Dabei entstammt *elle.gif* (D.13(b)) der Newsgroup `news:alt.binaries.pictures` im Jahre 1991 (Peters II, 1995) und wurde als gutes Beispiel für additives Gaußsches Rauschen ausgewählt. Das Bild *barbara.png* in der verrauschten Version (D.3(b)) entstammt Gilboa u. a. (2003) und wurde wegen seiner schwierigen Texturen gewählt.

Die Bilder *cell1a.png*, *cell1b.png* und *cell2.png* (D.5(d), D.8(b) und D.10(d)) sind Ausschnitte von mit einem Lasermikroskop von Maren Romeyke an der Universität Leipzig angefertigten Zellaufnahmen und wurden ausgewählt um die Eignung der Filter bei medizinischen, mit hart an der jeweiligen Auflösungsgrenze liegenden und damit verrauschten und unscharfen Bildern zu testen.

Das Bild *lea.png* (D.23(b)) wurde wegen der an sich weichen Struktur des Bildes bei gleichzeitiger Überbelichtung ausgesucht, um Grenzen der Filter – vor allem der auf linearer Regression basierenden – aufzuzeigen.

Das Bild *snap-E6.png* (D.30(d)) entstammt einer realen TV-Aufnahme vom Oktober 2003 und zeigt einen Ausschnitt aus einem Volksfest. In der linken unteren Ecke ist der Kopf eines Mannes im Halbprofil zu sehen.

Dagegen stellt *snap-SE10.png* (D.33(b)) einen Ausschnitt einer Werbeeinblendung dar, wobei die Zahl drei auch für das ungeübte Auge noch zu erkennen ist.

1.3.3 Künstlich verrauschte Bilder

Um die Wirkung der verschiedenen Filter auch mathematisch und damit objektiv zu untermauern, wurden die Bilder *GONY.multipl.png* und *GONY.impulse.png* durch das Tool *ImageMagick*[5] durch künstliches Verrauschen von *gony.png* erzeugt. Die erste Version (D.18(b)) wurde zur Beurteilung der Filter hinsichtlich der Qualitätsverbesserung bei multiplikativen Rauschen[6], die zweite (D.15(d)) für die Bewertung bei binärem Impulsrauschen herangezogen. Das Bild *gony.png* ist ein Teilausschnitt einer Szene der DVD `Gangs of New York`.

1.3.4 Künstliche Testbilder

Für eine genauere Bewertung der Eigenschaften der zu untersuchenden Filter war es notwendig eine Reihe von speziellen, künstlichen Testbildern zu schaffen bzw. heranzuziehen.

So kann man bei gefilterten *schachbrett.png* (D.28(b)) die Art und die Fenstergröße der benutzen Filter erkennen. Auch lassen sich Aussagen über die Qualität der Kantendetektion ableiten.

Das Bild *labyrinth.png* (D.20(d)) enthält ein künstlich erzeugtes, binäres Labyrinth mit der Strichdicke von einem Pixel und gibt bei der Anwendung der Filter Aufschluß darüber, inwieweit Details von den einzelnen Filtern zerstört oder verfälscht werden. Die zufällig erzeugten Verästelungen des Labyrinths entsprechen einer Gaußschen Normalverteilung, so daß reine Mittelwertfilter zur Erzeugung einer homogenen grauen Fläche angeregt werden.

rampn.png (D.25(d)) wurde durch Mittelwertbildung eines Grauwertkeils und eines Bildes mit Gaußverteiltem Rauschen gewonnen. Es dient zur Ermittlung des Filterverhaltens bei Gaußschem Rauschen.

Um einen Vergleich mit den Ergebnissen von Smith und Brady (1995) zu haben, wurde das Bild *susantest.png* (D.35(d)) aus dieser Quelle herangezogen. Es enthält

[5] `http://imagemagick.org`
[6] sh. 2.2.1, S. 12

Grauwertkeile und geometrische Figuren und erlaubt einen Kurzüberblick über das verwendete Filter.

2 Mathematische Grundlagen

2.1 Qualitätsabschätzung

2.1.1 Signal-Rausch-Abstand

Die Qualität der Filter kann durch die Ermittlung der Signal-Diffenzen von Originalbild und verrauschtem, sowie Original- und rekonstruiertem Bild, zB. durch Peak Signal-to-Noise Ratio abgeschätzt werden:

$$PSNR(X,\hat{X})_{\text{in dB}} = 10\log\left(\frac{255}{\sum(x_i-\hat{x_i})^2}\frac{1}{(M+N)}\right) \tag{2.1}$$

wobei M und N Kantenlängen des Bildes, X das empfangene, $\hat{X}$ das Originalbild ist, $\hat{x}$ und x die Signalwerte dazu.

2.1.2 Zeilenweise Autokorrelation

Die Autokorrelation gibt an, wie selbstähnlich ein Bild ist. Hier ist damit gemeint, ein Bild so in ein neues Bild zu überführen, dass aufeinanderfolgende Zeilen so ineinander verschoben werden, daß eine möglichst gute Übereinstimmung erreicht wird.

$$X(x,y) = \hat{X}(x,\hat{y})\forall\hat{y} = \left\{1,2,\ldots,\hat{N}\right\}\exists\min\sum_{j=0}^{c}\left(\hat{X}(j,y-1)-\hat{X}(j+n,y)\right)^2 \tag{2.2}$$

$$j+n=\hat{M}, x=\left\{0,1,\ldots,\hat{M}-c\right\}$$

wobei $\hat{M}$ und $\hat{N}$ Kantenlängen des Originalbildes $\hat{X}$, M und N des Bildes X sind und c die Jitterbreite bzw. die Suchbreite angibt. Das Bild X hat die Kantenlängen $M=\hat{M}-c$ und $N=\hat{N}-1$, da M um die Jitterbreite und N durch eine Bezugszeile verkürzt werden.

2.1.3 Statistische Ausreißer

Nach Bartsch (1999) gilt als Ausreißer x_{n+1} für angenommene Normalverteilung, wenn:

$$x_{n+1} > \bar{x}_{1\ldots n} + K\sigma_{1\ldots n} \text{mit } K \approx 4 \tag{2.3}$$

Anstelle der Standardabweichung

$$\sigma_{1\ldots n} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{2.4}$$

wird empfohlen den Quartilabstand $x_{\frac{3}{4}-\frac{1}{4}}$ zu nehmen.

Vereinfacht wird x_{n+1} mit dem Mittelwert $\bar{x}_{1\ldots n}$ wie folgt verglichen:

$$\frac{|x_{n+1} - \bar{x}_{1\ldots n}|}{\bar{x}_{1\ldots n}} > 0.5 \tag{2.5}$$

2.2 Rauschen

2.2.1 Einführung

In digitalisiertem Material können im Wesentlichen folgende Arten (Vaseghi, 1996) des Rauschens auftreten:

- Quantisierungsrauschen
- binäres bzw. Impulsrauschen
- additives Gaußsches Rauschen
- multiplikatives Rauschen
- transientes Rauschen

Das Quantisierungsrauschen ist in der Regel vernachlässigbar.

Multiplikatives Rauschen entspricht einer Amplitudenmodulation des Originalsignals mit einem Rauschsignal und tritt in der Regel bei digitalisiertem Bildmaterial nicht auf.

Transientes Rauschen ist an Einschwingvorgänge gekoppelt und hier wegen der hauptsächlichen Betrachtung der Filter im Spatialraum ebenso vernachläßigbar,

so daß im Folgenden nur das binäre und das additive Gaußsche Rauschen[1] betrachtet werden.

Für eine ausführlichere Betrachtung sei nochmals auf Vaseghi (1996) hingewiesen.

2.2.2 Binäres Rauschen

Binäres oder auch allgemeiner Impulsrauschen sind relativ kurze 'Ein-/Aus'-Pulse, verursacht durch Drop-Outs[2], Über- oder Untersteuerung. Nach Vaseghi (1996) kann man Impulsrauschen über eine Amplitudenmodulation eines Zufallsrauschprozeß $n(m)$ mit einer binären Zustandsfolge $b(m)$ beschreiben:

$$n_i(m) = n(m)b(m) \tag{2.6}$$

Das Impulsrauschen wird vereinfacht durch ein Bernoullimodell als reines Binärrauschen beschrieben. Die Wahrscheinlichkeiten der Folge $b(m)$ ist wie folgt charakterisiert:

$$P_B\left(b(m)\right) = \begin{cases} \alpha & \text{für } b(m) = 1 \\ 1-\alpha & \text{für } b(m) = 0 \end{cases} \tag{2.7}$$

Der Zufallsrauschprozess $n(m)$ wird hier entgegen der Darstellung in Vaseghi (1996) als nicht relevant angenommen. Stattdessen wird $b(m) = 1$ so interpretiert, daß der Originalwert n des Bildes erhalten bleibt und bei $b(m) = 0$ durch den Maximalwert[3] oder Minimalwert[4] ersetzt wird, wobei dies wieder durch ein Bernoullimodell beschrieben werden kann. Vereinfacht ergibt sich mit $0 \leq \alpha + \beta \leq 1$ demnach:

$$P_W\left(b(m)\right) = \begin{cases} \alpha & \text{für } b(m) = n \\ \beta & \text{für } b(m) = 0 \\ 1-\alpha-\beta & \text{für } b(m) = 255 \end{cases} \tag{2.8}$$

Weitere Betrachtungen zu Impulsrauschen findet man bei Kunz (WS 2002/03).

[1] im ff. nur noch Gaußsches Rauschen genannt
[2] Aussetzer
[3] hier: 255
[4] hier: 0

Ein Beispiel zu binär verrauschten Bildern findet sich im Anhang unter D.15(d), Seite 77. Man beachte, daß Impulsrauschen nicht mit dem multiplikativen Rauschen zu verwechseln ist.

2.2.3 Gaußsches Rauschen

Der Gaußsche Rauschprozess, auch Normalprozess, wird von allen stochastischen Modellen am häufigsten genutzt. Das Gaußsche Wahrsscheinlichkeitsmodell approximiert sehr gut physikalische Vorgänge und kann durch lineare Lösungen vorhergesagt werden. Das zentrale Grenzwerttheorem besagt, daß die Summe voneinander unabhängiger Zufallsprozesse eine Gaußverteilung annimmt (Vaseghi, 1996).

Die Gaußsche Wahrscheinlichkeitsdichtefunktion ist:

$$P(m) = \frac{1}{\sqrt{2\pi}\sigma_m} \exp\left(-\frac{(m-\mu_m)^2}{2\sigma_m^2}\right) \tag{2.9}$$

wobei μ_m und σ_m^2 der Mittelwert und die Varianz der Zufallsvariablen m ist. Beispiele für Gaußverrauschte Bilder sind D.13(b) und D.25(d) Seite 75 und folgende.

3 Grundlegende Filter

3.1 Einführung

Man unterscheidet Filter nach

- der Art der Operation: linear und nichtlinear
- dem Wirkungsradius: global und lokal

.

Im Folgenden werden nur lokal wirkende Filter betrachtet.

Seien Filter ((Pavlidis, 1990, vgl.)) solche der Form

$$g(x,y) = \sum_{i=-M}^{M} \sum_{j=-M}^{M} h(x,y,i,j) f(x+i, y+j) \tag{3.1}$$

mit $h(x,y,i,j)$ der Gewichtsfunktion, i und j der Fensterposition. Filter dieser Form heißen linear.

Wenn die Gewichtsfunktion $h(x,y,i,j)$ für das gesamte Bild gleich ist und nicht von (x,y) abhängig ist, so kann die Gleichung 3.1 auch geschrieben werden als:

$$g(x,y) = \sum_{i=-M}^{M} \sum_{j=-M}^{M} h(i,j) f(x+i, y+j) \tag{3.2}$$

Filter dieser Form sind separierbar und heißen Faltungs- oder Convolutionfilter.

Filter, die ein lokales Strukturelement $\sigma(i,j)$ mit $i,j \neq 0$ benutzen, heißen morphologisch.

Filter, die aus einer geordneten Folge der lokalen Nachbarschaftswerte einen aktuellen Wert entnehmen, heißen Rangordnungsfilter.

3.2 Gleitender Mittelwert

Die Berechnung des Mittelwertes ist eine lineare Operation (Smith, 1997a):

Der gleitende Mittelwert ist die Mittelwertbildung über alle Werte x_i eines Fensters der Größe M.

$$\bar{x} = \frac{1}{M} \sum_{i=1}^{M} x_i = \frac{1}{M} y(n) \tag{3.3}$$

Es kann gezeigt werden, daß die gleitende Mittelwertsbildung separierbar ist, so dass statt einem Aufwand $T(nm^2)$ nur der Aufwand $T(nm)$ nötig ist.

Wenn man den gleitenden Mittelwert statt

$$y(n) = x\left(n - \frac{w}{2}\right) + \ldots + x\left(n + \frac{w}{2}\right) = \sum_{i=-\frac{w}{2}}^{\frac{w}{2}} x(n+i) \tag{3.4}$$

wie folgt berechnet (nach Smith (1997b)):

$$y(n+1) = y(n) - x\left(n - \frac{w}{2}\right) + x\left(n + 1 + \frac{w}{2}\right) \tag{3.5}$$

so sinkt der Aufwand gar auf $T(m+n)$ mit n der Anzahl der zu filternden Werte und m der Fenstergröße.

Der gleitende Mittelwert[1] ist gut zur Eliminierung von Gaußschem Rauschen geeignet, dies sieht man sehr gut in den Bildern 3.1(a) und 3.1(b), sowie in den Abbildungen 3.6 und 3.7 (auf den Seiten 22 und folgende), verschmiert aber Kanten und Impulse und ist nicht für die Behandlung von binärem Rauschen geeignet. Dies zeigt ein Vergleich mit dem Medianfilter in den Abbildungen 3.4 und 3.5.

3.3 Gewichteter Mittelwert

Der gewichtete Mittelwert, genauer der gleitende, gewichtete Mittelwert entspricht dem gleitenden Mittelwert, mit dem Unterschied, daß alle Funktionswerte mit einem spezifischen Gewicht multipliziert werden. Es handelt sich um eine nichtlineare Operation.

[1] auch Meanfilter, vgl. Medianfilter

Der gewichtete Mittelwert ist die Mittelwertbildung über alle gewichteten Werte $x_i w_i$ eines Fensters der Größe M.

Es kann gezeigt werden, daß die gewichtete Mittelwertsbildung dann separierbar ist, wenn:

- das Fenster quadratisch und
- die Gewichtsfunktion symmetrisch zur Fenstermitte ist

so daß statt einem Aufwand $T(mn^2)$ nur der Aufwand $T(mn)$ nötig ist.

Die Filterung mit einem gewichtetem Mittelwert, zB. gewichtet mit Gaußfunktion, liefert gute Ergebnisse meist erst mit großer Fenstergröße ($11 < m < 21$). Das heißt pro Filterung sind bis zu 21 Pixel einzubeziehen und mit der Gewichtsfunktion zu multiplizieren.

Eine kurze Abschätzung zeigt, daß der Aufwand deutlich reduziert werden kann, wenn stattdessen das Bild, als auch die Gewichtsfunktion des Filters in den Fourierraum transformiert, gefiltert und rücktransformiert(Schimpf, 1996) wird. In Gleichung 3.6 ist die Hin- und in 3.7 die Rücktransformation dargestellt:

$$F_{u,v} = \frac{1}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f_{m,n} \exp\left(\frac{-2\pi i m u}{M}\right) \exp\left(\frac{-2\pi i n v}{N}\right) \tag{3.6}$$

$$f_{m,n} = \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F_{u,v} \exp\left(\frac{2\pi i m u}{M}\right) \exp\left(\frac{2\pi i n v}{N}\right) \tag{3.7}$$

Die Faltung im Ortsraum entspricht der Multiplikation im Frequenzraum, so daß für den Aufwand näherungsweise $O(n \log_2 n)$ gilt.

Die gewichtete Mittelwertfilterung, insbesondere die mit der Gaußfunktion gewichtete, ist gut für die Entfernung von additiven Gaußschen Rauschen geeignet (Jähne, 1997), neigt aber wie der einfache gleitende Mittelwert zur Verschmierung von Kanten, sh. Bild 3.2(b), vgl. mit Bild 3.1(b) , Seite 20. .

3.4 Median

Der Medianfilter(Winkler, 2001) zählt zu den Rangordnungsfiltern und bewirkt eine nichtlineare Operation.

Der Medianfilter liefert den Wert für das aktuelle Pixel zurück, das den Wert in der Mitte einer sortierten Folge von Pixelwerten des Fensters der Breite m darstellt.

$$\tilde{x} = x_{\frac{m+1}{2}} \forall x_i \in (x_1 \leq x_2 \leq \ldots \leq x_m) \tag{3.8}$$

Es kann gezeigt werden, daß der Medianfilter nicht separierbar ist:

$$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 3 \\ 2 & 3 & 1 \end{pmatrix}$$

der Median $\tilde{x} = 2$, aber ergibt zeilenweise

$$\begin{pmatrix} 2 \\ 3 \\ 2 \end{pmatrix}$$

somit als Zeilenmedian $\tilde{x}_i = 2$

und spaltenweise

$$\begin{pmatrix} 1 & 3 & 3 \end{pmatrix}$$

somit als Spaltenmedian $\tilde{x}_j = 3$

Der Median obiger Matrix ist aber $\tilde{x} = 2$.

Der Aufwand für die Medianfilterung beträgt $T(nm \log_2 m)$, wobei dieser bei geschickter Implementation über Heaps auf $T(n \log_2 m)$ gedrückt werden kann, für numerische Betrachtung sei alternativ auf Griffin (2000) verwiesen.

Medianfilter sind sehr gut zur Unterdrückung einzelner Impulsstörungen geeignet, dies sieht man deutlich in den Abbildungen 3.4 und 3.5, sowie den Bildern 3.3(c) und 3.3(d). Die Fensterbreite hat Einfluß auf die Erhaltung von Details und die Qualität der Rauschunterdrückung. Mediangefilterte Bilder haben die Neigung ausgefranst und unnatürlich auszusehen, wie in Bild 3.3(b). Details kleiner der Fensterbreite $\frac{m}{2}$ werden dabei unterdrückt. Medianfilter erhalten aber Kanten und Verläufe.

3.5 Mode

Der Modefilter[2] ist nicht separierbar, zählt zu den Rangordnungsfiltern und bewirkt eine nichtlineare Operation.

[2] auch Modalfilter

Der Modefilter liefert den Wert für das aktuelle Pixel zurück, der im Fenster am häufigsten vorkommt.

Der Aufwand beträgt $T(mn)$, wobei bei dieser bei geschickter Implementation[3] auf $T(n)$ gedrückt werden kann.

Da für den Median (Seite 17) die Werte im Fenster maximal r-mal verfälscht sein dürfen, unter Annahme der Gleichverteilung und mit $r \approx \frac{n}{2}$, so gilt für den Modefilter, daß der wahre Wert w mit $w > r$ öfter vorkommen muß. Bei einer einfachen Filterung wird dies nicht zum gewünschten Ergebnis führen, da oft kein relatives Maximum in der Häufigkeitsverteilung auftritt. Bei einer Requantisierung des Bildes mit zum Beispiel 16 Graustufen kann dagegen der Modefilter durchaus gute Ergebnisse bei stark verrauschtem Material erzielen.

Eine vergleichende Betrachtung zwischen zweidimensionaler Mittelwert- Median- und Modefilterung findet man bei Griffin (2000).

[3] zum Beispiel über ein Array, welches die Häufigkeit der Grauwerte der Fensterpixel zählt

(a) Elle, Original

(b) Elle, gefiltert

Abbildung 3.1: Gleitender Mittelwert

(a) Elle, Original

(b) Elle, gefiltert

Abbildung 3.2: Gewichteter Mittelwert

(a) Elle, Original

(b) Elle, gefiltert

(c) GONY, binär verrauscht

(d) GONY, gefiltert

Abbildung 3.3: Binäres Rauschen und Median

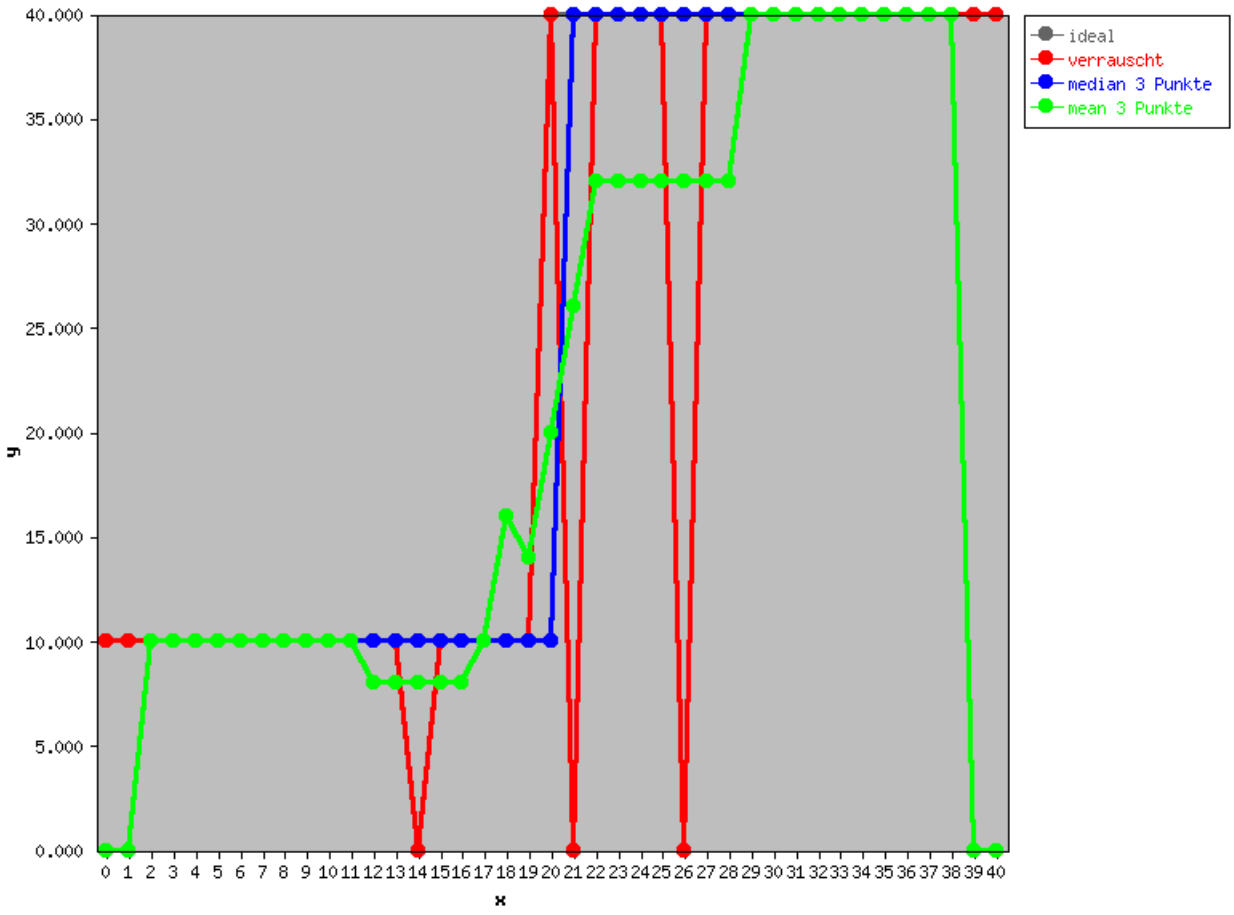


Abbildung 3.4: Binär verrauschte Kante – Wirkung des gleitenden Mittelwert- und Medianfilters

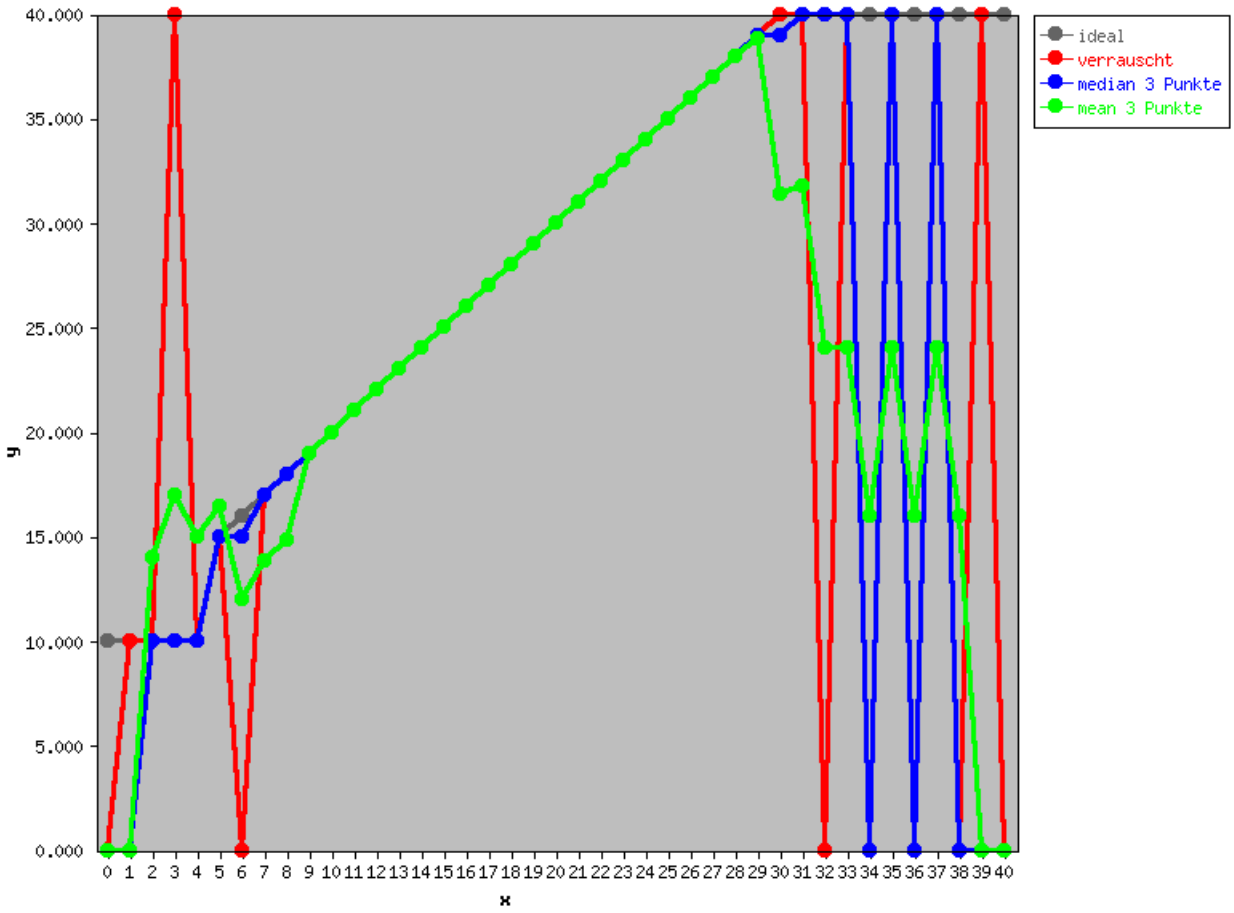


Abbildung 3.5: Binär verrauschte Rampe – Wirkung des gleitenden Mittelwert- und Medianfilters

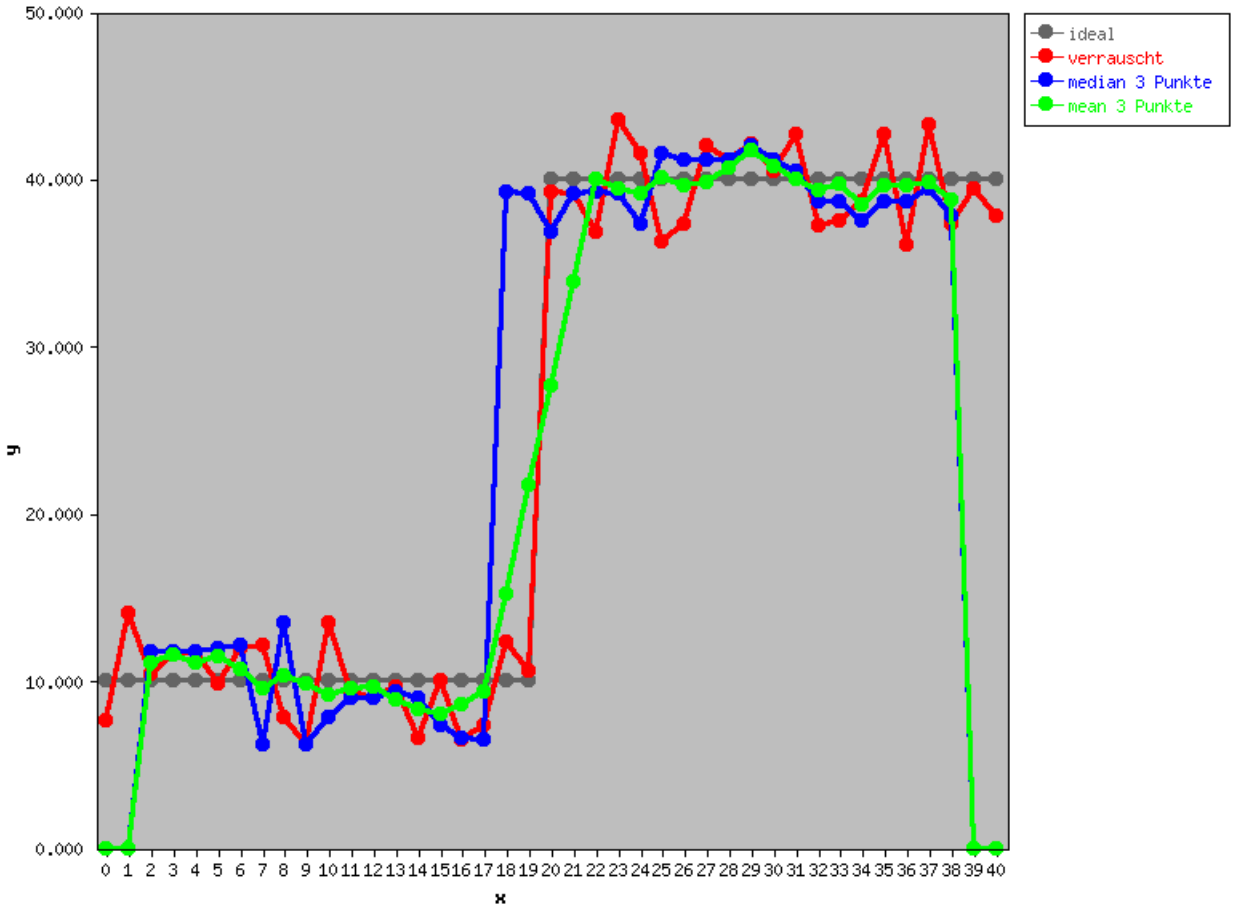


Abbildung 3.6: Gauß verrauschte Kante – Wirkung des gleitenden Mittelwert- und Medianfilters

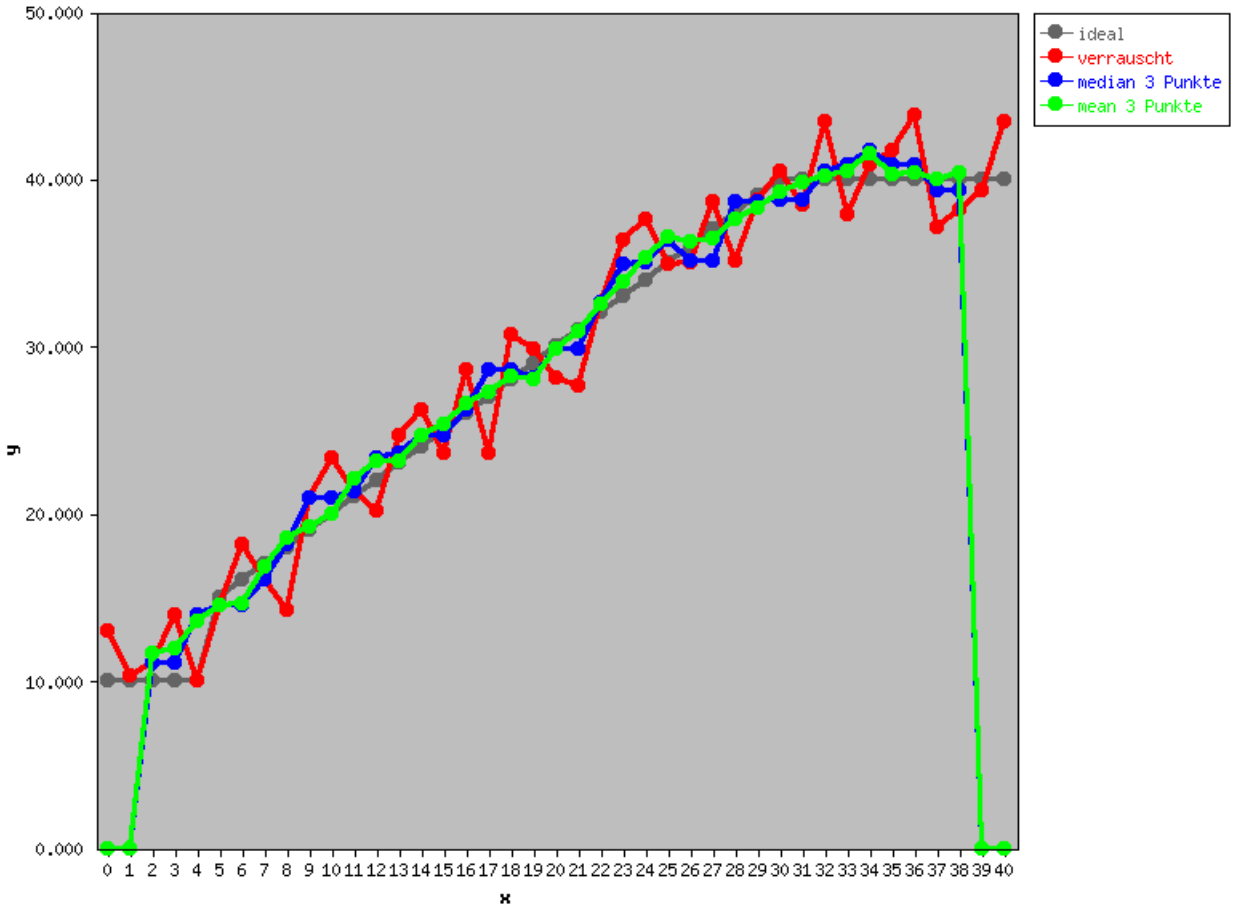


Abbildung 3.7: Gauß verrauschte Rampe – Wirkung des gleitenden Mittelwert- und Medianfilters

4 Weitere Filter

4.1 Autokorrelationsfilter

Wenn man sich die Bilder *snap-E6.png*, Seite 92 und *snap-SE10.png*, Seite 95, die einer stark verrauschten TV-Aufnahme entstammen, genauer anschaut, so ist zu vermuten, daß aufgrund fehlerhafter Synchronisation auf den Zeilenwechsel einige Zeilen ineinander verschoben sind. Um auszuschließen, daß dies tatsächlich der Fall ist, wurde zu Testzwecken ein Korrekturfilter programmiert, (sh. Seite 11,2.2 und Programm Nr. C.1 auf Seite 41), der zeilenweise die Autokorrelation berechnet und ggf. einen Jitter eliminiert.

Im Ergebnis stellte sich heraus, daß obige Annahme nicht falsch war und die Zeilen-Jitter sich nicht nur als Kammartefakte des Interlacings des TV-Signals herausstellten.

Bei der Implementierung muß man aufpassen, daß der Autokorrelationsfilter bei homogenen Zeilen nicht bis zur maximalen Verschiebung läuft, sondern den Originalwert ins Zielbild schreibt (man vergleiche Bilder Seite 93 bzw. Seite 95).

Auch hängt die Qualität der Autokorrelation der Zeilen ganz stark davon ab, wieviele Zeilen in die Berechnung einfließen. Der Grund liegt einerseits darin, daß keine Zeile als unverfälscht gelten kann, so daß eine Referenzzeile, die besonders gestört ist, durchaus zu verstärkten Fehlern führen kann.

Für eine weitere Betrachtung von Autokorrelationsfiltern kann man die hier vorgestellten Filter zur Rauschunterdrückung mit heranziehen. So ist es zum Beispiel denkbar, daß Referenzzeilen, die näher zur aktuellen [1] Zeile liegen, stärker Berücksichtigung finden, als weiter entferntere. Dies ist vergleichbar mit Gewichteter Mittelwert, S.16.

[1] also zu der zu autokorrelierenden

4.2 Morphologische Filter

Morphologische Filter, insbesondere Erosion und Dilatation, wie auch Opening und Closing, spielen eine wichtige Rolle in der aktuellen Forschung im Bereich der Bildverarbeitung (vgl. (Zlokolica u. a., 2002), (Peters II, 1995)). Im Wesentlichen werden bei Morphologischen Filtern zweidimensionale Strukturelemente[2] benutzt, diese im Quellbild gesucht und bei Vorhandensein ins Zielbild kopiert. Morphologische Filter untersuchen die Nachbarschaft eines Pixels und sind gut zur Erkennung von Kanten und Ecken geeignet (sh. (Smith und Brady, 1995) und (Hazard u. a., 1998)).

Auf eine Implementierung im Rahmen der vorliegenden Arbeit wurde verzichtet, da Aufgrund der erwarteten Komplexität (Wahl des Strukturelementes, Anwendung auf Grauwert- und Farbbilder) und des Umfangs eine separate Betrachtung angemessener erschien.

4.3 SUSAN

In der einschlägigen Fachliteratur findet man viele Verweise auf den SUSAN-Algorithmus, (vgl. Smith und Brady (1995)). Um die Qualität der zu untersuchenden Filter besser einschätzen zu können, wurde das Programm *susan2l* mit Defaultparametern verwendet. SUSAN besitzt eine spezielle Erkennungsroutine für Ecken- und Kantendetektion und arbeitet mit morphologischen Operationen. Für Details sei auf das Originalpaper verwiesen. Das Testbild D.35(d), Seite 97, stammt aus eben dieser Quelle.

[2] entspricht Mustern

5 Entwurf und Implementierung

5.1 Entwurf

5.1.1 Anforderungen

Da bei verrauschtem Videomaterial im Allgemeinen das Impuls- und das Gaußsche Rauschen in Kombination auftreten, ist es notwendig, einen Rangordnungsfilter, wie zB. Median (sh. 3.4, 17) und Glättungsfilter, wie zB. gleitender Mittelwertfilter (sh. 3.2, 16), geschickt zu kombinieren.

Desweiteren sollte der Filter wenig komplex sein, damit er auf PCs in Echtzeit Filterergebnisse liefert.

Auch sollte der Filter gegen viele verschiedene und verschieden starke Störungen robust sein und durchschnittlich gute Ergebnisse liefern.

Der Filter sollte dabei Detailinformationen so gut es geht erhalten.

Hier nochmal eine Zusammenfassung:

- geringe Komplexität
- Erhaltung Detailinformationen
- robust bei Gaußschem und Impulsrauschen
- geeignet sowohl bei geringem, als auch bei hohem Rauschpegel

Auf Grund der Vorbetrachtung und der Vergleiche der verschiedenen Basisfilter, sowie der Vorschläge in der Literatur kristallisierten sich zwei Ideen heraus:

- adaptiver Median/Meanfilter (Medmean)
- lineare Regression (Linregr, adapt)

5.1.2 Medmean

Die Idee bei diesem Filter ist, abhängig von der lokalen Umgebung eines Pixels diesen entweder durch den gleitenden Mittelwert oder durch den Median zu ersetzen. Die Umgebung wird durch die Fenstergröße definiert. Als Maß für die Beurteilung wird die Standardabweichung (sh. Seite 12, Statistische Ausreißer) bzw. der Mittelwert der Umgebung mit dem Wert des aktuellen Pixels verglichen. Über- oder unterschreitet dieses ein bestimmtes Maß, so wird der Pixelwert durch den Median seiner Umgebung ersetzt. Ist dies nicht der Fall, so wird er durch den Mittelwert ersetzt. Danach wird das Fenster weitergeschoben.

Dabei werden für die nächsten Pixel im Fenster wieder die Originalwerte benutzt, so daß sich ein FIR-Verhalten ergibt.

5.1.3 Linregr, adapt

Dieser Filter kombiniert mehrere Ideen. Eine ist die Verwendung von Regressionsfunktionen als Filter, da gezeigt werden kann, daß diese gleichverteilt auftretende Störungen[1] sehr gut eliminieren können.

Auch haben sie den Vorteil, daß man nicht jeden Funktions- bzw. Pixelwert der Originalfunktion kennen muß, um eine Anpassung zu erreichen. Daher eignen sich Regressionsfunktionen auch gut für den Einsatz bei impulsverrauschten Signalen, da man detektierte Impulstörungen nicht als Wert für die Berechnung der Regressionsfunktion heranzieht.

Um einen darauf basierenden Filter einfacher und damit schneller zu machen, bietet es sich an, die Originalfunktion stückweise mit linearen Funktionen anzunähern (zB. jeweils 16 Pixel gehören zu einer Regressionsgerade). Der Einsatz von linearer Regression ist für Filterung von Bildern besonders geeignet, da die einzelnen Bilder in der Regel nur maximal 256 Quantisierungsstufen besitzen und der Fehler gegenüber dem Original damit recht gering ausfallen wird.

Ein Beispiel für die Wirkung dieser Filterung kann man im Bild 5.1(b) sehen.

Um sowohl Überschwingen (sh. Bild 5.1(c)) zu vermeiden[2] , als auch den Fehler möglichst gering zu halten, ist es erforderlich geeignete Bereiche zu finden, die

[1] additives Gaußsches Rauschen, wie auch andere, mittelwertfreie Rauschverteilungen

[2] Randbereiche wurden gegenüber 5.1(b) nicht bewertet, bzw. nur mittlere Pixel durch Werte der Regressionsgerade ersetzt

stückweise durch lineare Regressionsfunktionen der Form

$$\tilde{y} = f(x) = ax + b \tag{5.1}$$

ergibt. Wobei die Koeffizienten a und b nach Bartsch (1999) über Optimierungsbedingung (Methode der kleinsten Quadrate):

$$Q := \sum_{i=1}^{n} (y_i - f(x_i))^2 \rightarrow \min \tag{5.2}$$

mit folgendem Gleichungssystem berechnet werden:

$$bn + a \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i \tag{5.3}$$

$$b \sum_{i=1}^{n} x_i + a \sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} y_i x_i \tag{5.4}$$

Dazu muß man abzuschätzen, wo potentielle Extrem- und Wendepunkte liegen.

Für diese Prädiktion bot sich ein gleitender Mittelwert (vgl. 16) an, der nebenbei auch wieder die Detektion von Impulsrauschen vornehmen kann.

Der aktuelle Wert y_t gilt als erkannter Rauschimpuls, wenn gilt[3]: $\frac{|y-\bar{y}|}{\bar{y}} > c$. War y als solcher erkannt, fließt sein Wert nicht in die Bestimmung der Regressionsgeraden und des gleitenden Mittels ein.

Für die Erkennung der Extrem- und Wendepunkte gab es mehrere Möglichkeiten. Hier wurde eine Trajektion benutzt. Man schiebt zwei gleitende Mittelwerte über das Bild. Dabei läuft das eine Fenster dem anderen um n-Pixel hinterher. Verändert sich die Steigung bzw. die Differenz der zwei Mittelwerte um ein bestimmtes c, so gilt ein Punkt als Wende- oder Extrempunkt erkannt, und die Regressionsgerade wird bis dahin berechnet (sh. Bild 5.1(d)).

Da der inhärente Einfluß des Rauschens unter Umständen die Lage der Regressionsgerade, vor allem die Konstanten b_i beeinflußt, können Überschwingeffekte entstehen. Durch eine Berechnung des Schnittpunktes zweier benachbarter Regressionsgeraden können diese vermieden werden.

Zur Zeit wurde die Schnittpunktberechnung noch nicht implementiert.

[3] Hier wird auch die Abweichung vom lokalen Mittelwert benutzt, siehe 5.1.2, Seite 29

5.2 Implementierung

5.2.1 Verwendung als PGM-Programme

PGM (portable gray map) ist ein Abkömmling der in der Linuxwelt bekannten PPM (portable picture map) Familie. Das Format wurde gewählt, da es sich sowohl durch seine Einfachheit, als auch durch die große Unterstützung durch Bildverarbeitungsprogramme auszeichnet. Ein weiterer Pluspunkt war die mögliche automatisierte Verarbeitung über die *netPBM*-Tools.

Es wurde ein Grundgerüst genutzt, um Graustufenbilder in einen Speicherbereich ein- und auszulesen. Dann wurde die Funktion `filter()` aufgerufen, die bei jedem Programm durch eine angepaßte Routine ersetzt wurde. Diese `filter()`-Funktionen finden sich im Anhang. Durch diese Modularität konnte der Implementierungs- und Anpassungsaufwand niedrig gehalten werden.

Es zeigte sich im ganzen Verlauf des Projektes, daß die Wahl der Implementierung als PGM-Programme die richtige wahr, da dadurch scriptgesteuerte Auswertungen ermöglicht wurden. Durch die Implementierung in C konnte die Kompatibilität auf Quellcodeebene gesichert werden, so daß dem Einsatz der Programme zum Beispiel unter Windows nichts im Wege steht.

5.2.2 Verwendung in XawTV

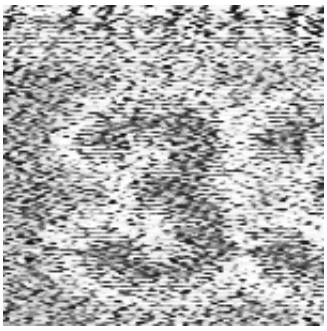
XawTV (Knorr, 2003) ist ein Viewer zum Fernsehempfang am PC. Es unterstützt unter anderem die Schnittstelle *video4linux* des Betriebssystems Linux. Neben etlichen Features besitzt XawTV ein PlugIn-System um Filter in die Verarbeitungskette einklinken zu können. Dabei können über eine generische Struktur verschiedene Parameter, wie zum Beispiel Auflösung, Bild- und Farbformat mit dem PlugIn ausgetauscht werden.

Durch die C-Implementierung der untersuchten Filter als PGM-Filter für die Kommandozeile konnte ohne Anpassungsschwierigkeiten ein solches Plugin realisiert werden. Es bot sich der Einfachheit halber an als Farbformate nur die Formate `RGB` und `gray` zu unterstützen.

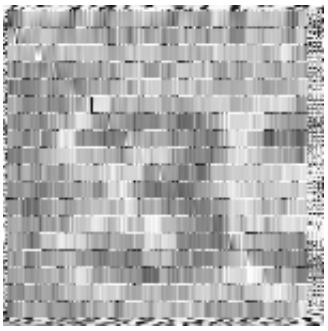
Eine Unterstützung für YUV im 4:2:2 Modus scheiterte daran, daß U und V gegenüber Y eine geringere Auflösung besitzen und damit bei einer Impulsdetektion bzw. Medianfilterung zu Farbpixelverfälschungen kam, was sich in störendem Farbrauschen niederschlug.

Einfacher war da die Umsetzung für das Format RGB, da dort einfach für jeden Farbkanal der Filter aufgerufen wurde. Da im PAL-Format die Bildauflösung 768×512 Pixel üblich ist, ergibt sich im RGB-Modus pro Frame die stattliche Anzahl von $768 \times 512 \times 3 = 1.179.648$ Pixeln, das macht bei einer Abtastrate von 24 Frames/s pro Sekunde 28.311.552 Pixel, die bearbeitet werden wollen. Man vergleiche die Laufzeit mit den Werten der Tabelle A.1 im Anhang auf Seite 39.

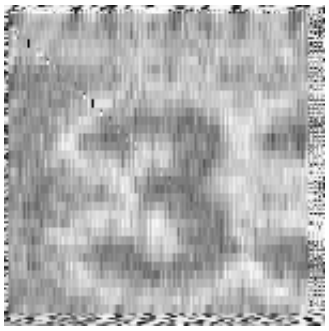
Diese Werte sind in der Praxis dennoch unkritisch, da der Filter nicht hintereinander pro Kanal aufgerufen wird und die gemeinsamen Schleifen herausgezogen wurden, so dass gegenüber dem Aufwand eines Filters als PGM-Programm von $T(n) = f(x)$ nur eine zusätzliche Konstante wirkt $T(n) = f(cx)$.



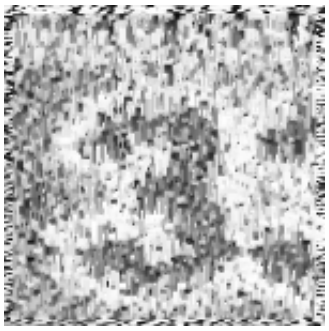
(a) snap-SE10, Original



(b) snap-SE10, lineare Regression



(c) snap-SE10, lineare Regression, Variante 2



(d) snap-SE10, lineare Regression, adaptiv

Abbildung 5.1: Lineare Regression

6 Abschließende Betrachtungen

6.1 Zeitbedarf vs. Qualität

6.1.1 Allgemeines

Um einen Vergleich durchführen zu können, war es notwendig, alle[1] Filter mit der gleichen Fenstergröße zu implementieren. Um die Wirkung des gewichteten Mittelwertfilters (sh. 3.3, Seite 16) sichtbar zu machen, war es notwendig ein 5×5-Fenster zu benutzen.

Wie erwartet, zeigten die Rangordnungsfilter Median und Mode eine klare Qualitätsverbesserung bei impulsverrauschten Bildern, führten ansonsten aber zu Artefakten und Kontrastverstärkung. Die Faltungsfilter 'Gleitender Mittelwert' und 'Gewichteter Mittelwert' zeigten gute Ergebnisse bei Gaußverrauschten Bildern, schnitten aber bei impulsverrauschten Bildern schlecht ab, da die Störimpulse breit geschmiert wurden. Diese subjektive Wertung anhand der testbilder deckt sich mit den Ergebnissen des PSNR-Vergleichs (sh. Tabelle B.2 auf Seite 40).

Sehr deutlich sieht man den Vorsprung des *Linregr, adapt* in der PSNR-Bewertung.

Interessanterweise war die Geschwindigkeit des Mittelwertfilters signifikant niedriger, als die des 'Lineare Regression'-filters und ist wahrscheinlich auf Cacheeffekte zurückzuführen. Eine Erhöhung kann uU. durch eine sorgfältige Optimierung vorgenommen werden.

6.1.2 Medmean

Aus oben angeführtem Grund mußte der ursprüngliche Medmean mit einem 3×3 auf ein 5×5-Fenster umgestellt werden. Dabei stellte sich heraus, daß die Implementierung des Medmean-Filters (Seite 29) mit diesem Fenster gegenüber der originalen Variante mit 3×3-Fenster zu einer qualitativen Verschlechterung der ge-

[1] *Linreg, adapt* war eine Ausnahme, da er als solche nicht mit Fenstern arbeitet, bzw. die Fenster nicht zur Filterung, sondern nur zur Erkennung der Extrema benutzt werden.

filterten Testbilder führte. Die Ursache liegt darin begründet, daß der Medmean-Filter hart zwischen Median- und Mittelwertfilterung schaltet. Durch die Mittelwertfilterung kommt es aber zu einer Verschmierung von Pixeln und Kanten. Wird an so einer Grenzstelle auf Median umgeschaltet, so wird in die weiche Umgebung hart ein neuer Pixel gesetzt. Durch den entstehenden Kontrast ergibt sich diese verschlechternde Wirkung. Bei der Implementierung mit 3×3-Fenster ist der Kontrast zwischen den weichen Pixeln der Mittelwertfilterung und dem Pixel des Medians nicht so hoch, so daß der Filter für diese Variante deutlich bessere Ergebnisse zeigt.

Es wurden auch zwei Varianten der Umschaltung des Medmean getestet. Die erste basierte auf der lokalen Standardabweichung $s > c$, die zweite auf Umschaltung durch Abweichung vom lokalen Mittelwert $\frac{|y-\bar{y}|}{\bar{y}} > c$. Interessanterweise lieferte die letzte Variante die besseren Ergebnisse, so daß im vorliegenden Dokument nur darauf Bezug genommen wird. Es empfiehlt sich c in den Grenzen $0,05 \leq c \leq 0,5$ zu halten Als günstiger Wert hat sich $c = 0,1$ herausgestellt.

Im Zeitbedarf liegt der Medmean-Filter hinten. Aus der Tabelle A.1 im Anhang A auf Seite 39 wird dies sehr stark deutlich. Hier liegt die Ursache im Erkennen von Impulspixeln und in der Medianfilterung. Durch eine geschicktere Programmierung kann dieser Filter noch deutlich beschleunigt werden. Neben der Verbesserung des Algorithmus zur Ermittelung des Medians[2] bietet die Optimierung von Cachezugriffen genug Optimierungsspielraum.

Auch wenn der Medmean-Algorithmus scheinbar solche Schwächen zeigt, so steckt in ihm sehr viel Potential, der ihn interessant für eine Weiterentwicklung zu einem guten adaptiven Filter macht, sh. Bild 6.1(b). So könnte durch Ersetzen des harten Umschaltens zwischen Median und Mittelwert mit einer soften Durchsteuerung (vgl. (Donoho, 1995)), zB. durch Überblendung oder durch stufenweises Umschalten zwischen Median, 3×3 Mittelwert und 5×5-Mittelwert die Qualität verbessert werden. Auch eine Variante, die statt des einfachen Mittelwertfilters den gewichteten Mittelwertfilter benutzt, wäre denkbar.

In der Implementierung mit einem 3×3-Fenster stellt er einen guten Kompromiß zwischen Qualität und Geschwindigkeit bei mäßig verrauschtem Bildmaterial dar.

6.1.3 Linregr, adapt

In der vorliegenden Implementierung überraschte dieser Filter durch seine guten Ergebnisse und der schnellen Filterung. Er lag dabei weit unter dem Rechenzeitbedarf des SUSAN-Filters und des Medmean-Filters. Auch wenn im Testbild D.35(d)

[2] sh. 3.4, S. 17, sowie vgl. Winkler (2001)

Artefakte sichtbar wurden, so zeigte er sowohl bei schwach- als auch stark verrauschten Bildern, wie auch bei verschiedenen Rauscharten eine durchweg konstant gute Qualität (sh. Bild 6.1(c)). In der vorliegenden Implementierung wurde eine einfache Impulspixelerkennung eingebaut, als günstiger Wert für c stellte sich $c = 0,05$ heraus (sh. 5.1.3, Seite 29). .

Auch wurde der Filter als separierbarer Filter angelegt, dh. in einem ersten Durchlauf wird nur vertikal die jeweilige lineare Regression, in einem zweiten Lauf horizontal die linearen Regressionsgeraden berechnet.

Korrekterweise müsste eine zweidimensionale lineare Regression verwendet werden. Dies und die noch fehlende Schnittpunktberechnung von zwei aneinandergrenzenden Regressionsgeraden zur Vermeidung von Überschwingen versprechen eine weitere Qualitätsteigerung.

6.2 Zusammenfassung

Diese Projektarbeit konnte aufgrund des begrenzten zeitlichen Rahmens nur einen Überblick über die grundlegenden Verfahren der Filterung von verrauschten Bildern geben. Die Verfahren wurden zwecks einfacherer Untersuchung als standalone Filterprogramme in der Programmiersprache C implementiert. Dadurch kann der Quellcode unter den Bedingungen der GPL[3] frei verwendet werden. Die Programme unterstützen das weitverbreite PGM-Format der *netpbm*-Bibliothek. Die vorliegende Arbeit zeigte auf, welche Arten von Rauschen beim Empfang von analogen TV-Signalen zu erwarten sind, und wie dieses Rauschen modelliert werden kann. Der Schwerpunkt bei der Untersuchung der verschiedenen Filter lag in der Abschätzung der Qualität des gefilterten Bildes im Verhältnis zu dem erforderlichen Berechnungsaufwand. Dabei wurde die Qualität der Filter einerseits formal durch die Bestimmung der PSNR (sh. Seite 11), wie auch subjektiv durch die Analyse der Filterantwort auf verschiedene Testbilder bewertet. Anhand dieser Einordnung wurden die zwei möglichen kombinierten Filter vorgeschlagen, die geeignet erscheinen, in einer Implementierung für TV-Software in Echtzeit verrauschtes Videomaterial zu verarbeiten.

6.3 Ausblick

Für eine weitere Untersuchung empfiehlt es sich, auf der vorliegenden Arbeit aufzubauen. So verdienen vorallem die statistischen Parameter von verrausch-

[3] GPL – General Public License, sh. `http://www.fsf.org` für Details

ten Aufnahmen eine gründlichere Betrachtung. Auch konnte in dieser Arbeit nur der Filterung im Spatialbereich Aufmerksamkeit geschenkt werden, so daß es sich anbietet, vergleichbare Untersuchungen für Videofilter mit zeitlicher und örtlicher Filterung anzustellen. Unerwähnt blieben hiermit auch die FFT-[4], DCT- und Waveletbasierten Filtermethoden (vgl. (Aitnouh, 1999) und (Romer und Pösch, 2002)). Fuzzy-Systeme und adaptive, nichtlineare Prädiktoren für die Vorhersage und Elimination von Videostörungen (Abbas und Domanski, 2000), sogenannte Blind-convolution, wie auch Filter, welche die Textureigenschaften von Bildern heranziehen, bieten ein weiteres Feld für Untersuchungen.

So reifte beispielsweise die Idee, anhand der Schwarzbalken (bei 16:9-Ausstrahlungen im 4:3-Format oben und unten, bei 4:3-Austrahlungen im 16:9 Aufzeichnungsformat rechts und links) ein Rauschmodell aufzustellen und gestörte Videosequenzen adaptiv zu entstören.

Für weitere Anregungen sollte man sich die Projekte *tvtime*[5] und *transcode*[6] anschauen.

[4](Schimpf, 1996)
[5]*TVTime-Entwicklerteam* (2003)
[6]*Bitterberg, Tilmann and Östreich, Thomas and Entwicklerteam* (2003)

(a) Elle, Original

(b) Elle, Medmean

(c) Elle, Linreg, adapt

Abbildung 6.1: Vergleich Medmean und Linregr, adapt

A Vergleichstabelle Zeitaufwand

Filter	128x128	256x256	512x512	1024x1024
	= 16384	= 65536	= 262144	= 1048576
pgm_linregr2	0.005s	0.013s	0.047s	0.189s
pgm_linregr_adapt	0.016s	0.058s	0.357s	1.399s
pgm_mean	0.006s	0.013s	0.055s	0.165s
pgm_linregr	0.005s	0.012s	0.033s	0.130s
pgm_median	0.053s	0.219s	0.788s	3.252s
pgm_wmean	0.005s	0.013s	0.043s	0.184s
pgm_medmean	0.054s	0.242s	0.815s	4.250s
pgm_autocorr	0.004s	0.117s	0.948s	3.359s
pgm_mode	0.042s	0.184s	0.713s	2.927s

Tabelle A.1: Zeitmessung

Der Zeitaufwand, den jeder Filter benötigte, wurde für je vier Bilder mit Gaußschem Rauschen gemessen. Die Bilder wurden mit dem Programm *pgmnoise* der *pbmtools* erzeugt. Als Rechner stand ein PC, Duron 1300 Mhz, 512MB RAM zur Verfügung, die Messung erfolgte unter Linux mit Kernel 2.6.0 unter niedriger Last. Die Meßwerte entsprechen den *real*-Werten des Bash-builtins *time*. Durch die Verwendung von Rauschbildern wurde sichergestellt, daß alle Filter unter Bedingungen getestet wurden, die dem worst case entsprechen. Alle Programme wurden mit dem *gcc*[1] in der Version 3.3 mit folgender Sequenz `gcc -Os -Wall -m foo.c foo` kompiliert.

[1]GNU C Compiler

B Vergleichstabelle PSNR

Mit dem Tool *pnmpsnr* der *netpbm*-Bibliothek wurden die PSNR-Werte gegenüber dem Originalbild *GONY.pgm* ermittelt. Der PSNR-Wert ist umso höher, je ähnlicher sich zwei Bilder sind. In der folgenden Tabelle wurde die Leistung der Fil-

Verrauscht	PSNR gegenüber Original
GONY, impulse	18.53 dB
GONY, multipl	8.00 dB

Tabelle B.1: PSNR verrauscht zu Original

terung [1] mit verrauschten, gefilterten Bildern mit dem Originalbild *GONY.pgm* bestimmt:

Filter	PSNR Impulse	PSNR Multipl
pgm_linregr2	22.06 dB	12.01 dB
pgm_linregr_adapt	25.95 dB	11.12 dB
pgm_mean	24.11 dB	12.45 dB
pgm_linregr	21.14 dB	11.47 dB
pgm_median	19.85 dB	11.29 dB
pgm_wmean	24.86 dB	12.20 dB
pgm_medmean	23.56 dB	10.42 dB
pgm_autocorr	4.90 dB	4.29 dB
pgm_mode	19.82 dB	9.48 dB

Tabelle B.2: Messung PSNR

[1] *pgm_autocorr* schneidet hier scheinbar schlecht ab, liegt aber an den Jitter-Rändern, sh. auch 4.1, Seite 26

C Quellcode PGM-Filter

C.1 pgm_autocorr.c

```
void simple_autocorrel_filter(unsigned char ** in, unsigned char ** out, int * xsize, int * ysize) {
#define PIX(a,b) ( tin[((b) * maxx) + (a)])
#define OPI(a,b) ( tout[((b) * maxx) + (a)])
    unsigned char * tin;
    unsigned char * tout;
    int maxx, maxy;
    int x, y;
    int j, minj;
    double min;
#define mincorrpix 128
    double delta; /* mincorrpix entries to hold minimum about 2 lines */
    assert(*in != NULL);
    assert(*out == NULL);
    maxx= *xsize;
    maxy= *ysize;
    /* alloc memory to store stuff */
    * out = (unsigned char *) malloc(maxx * maxy *sizeof(unsigned char)); /* should be freed, if closed */
    if (* out == NULL) {
        fprintf(stderr, "could not allocate memory for output-buffer\n");
        exit(-1);
    }
    /* init with original image */
    *out = memcpy(*out, *in, maxx * maxy * sizeof(unsigned char));
    tin= *in;
    tout= *out;
    /* fill target with white pixels, to detect autocorr-jitter */
    for (y=0; y<(*ysize);y++) {
        for (x=0; x<(*xsize); x++) {
            OPI(x,y)=255;
        }
    }
    /* implement a simple autocorrelation per line */
    for (y=2; y<(*ysize)-2; y++) {
        j=0; min=256*256*256; minj=0;
        for (j= -mincorrpix/2; j< mincorrpix/2; j++) {
            delta=0;
```

```
            for (x=mincorrpix/2; x<((*xsize)−mincorrpix/2); x++) {
                delta+=
                (
                    ((PIX(x+j,y)−PIX(x,y−1)))*
                    ((PIX(x+j,y)−PIX(x,y−1)))+
                    ((PIX(x+j,y)−PIX(x,y+1)))*
                    ((PIX(x+j,y)−PIX(x,y+1)))+
                    ((PIX(x+j,y)−PIX(x,y−2)))*
                    ((PIX(x+j,y)−PIX(x,y−2)))+
                    ((PIX(x+j,y)−PIX(x,y+2)))*
                    ((PIX(x+j,y)−PIX(x,y+2)))
                    );
            }
            if (min > delta) {
                min=delta;
                minj=j;
            }
        }
        for (x=mincorrpix/2; x<((*xsize)−mincorrpix/2); x++) {
//          OPI(x+minj,y)=PIX(x,y);
            OPI(x−minj,y)=PIX(x,y);
        }
    }
}
```

C.2 pgm_linregr2.c

```
void simple_linregr_filter(unsigned char ** in, unsigned char ** out, int * xsize, int * ysize) {
#define PIX(a,b) ( tin[((b) * maxx) + (a)])
#define OPI(a,b) ( tout[((b) * maxx) + (a)])
    unsigned char * tin;
    unsigned char * tout;
    int maxx, maxy;
    int Y;
    int x,y;

    float a, b; /* a+bx */
    float meanx, meany;
    float sumxy, sumxx;
    float sumx, sumy;

    assert(*in != NULL);
    assert(*out == NULL);
    maxx= *xsize;
```

```
    maxy= *ysize;
    /* alloc memory to store stuff */
    * out = (unsigned char *) malloc(maxx * maxy *sizeof(unsigned char)); /* should be freed, if closed */
    if (* out == NULL) {
        fprintf(stderr, "could not allocate memory for output-buffer\n");
        exit(-1);
    }
    /* init with original image */
    *out = memcpy(*out, *in, maxx * maxy * sizeof(unsigned char));
    tin= *in;
    tout= *out;
    /* implement a simple linregr */
    for (y=2; y<maxy-8; y++) {
#ifndef DEMO
        for (x=2; x<maxx-8; x+=4) {
#else
        for (x=2; x<y; x+=4) {
#endif
                sumxy=
                    (x)*PIX((x),y) +
                    (x+1)*PIX((x+1),y)+
                    (x+2)*PIX((x+2),y)+
                    (x+3)*PIX((x+3),y)+
                    (x+4)*PIX((x+4),y) +
                    (x+5)*PIX((x+5),y)+
                    (x+6)*PIX((x+6),y)+
                    (x+7)*PIX((x+7),y);
                sumx=
                    (x)+
                    (x+1)+
                    (x+2)+
                    (x+3)+
                    (x+4)+
                    (x+5)+
                    (x+6)+
                    (x+7);
                sumxx=
                    (x)*(x)+
                    (x+1)*(x+1)+
                    (x+2)*(x+2)+
                    (x+3)*(x+3)+
                    (x+4)*(x+4)+
                    (x+5)*(x+5)+
                    (x+6)*(x+6)+
                    (x+7)*(x+7);
                sumy=
                    PIX(x,y)+
                    PIX(x+1,y)+
                    PIX(x+2,y)+
                    PIX(x+3,y)+
```

```
                    PIX(x+4,y)+
                    PIX(x+5,y)+
                    PIX(x+6,y)+
                    PIX(x+7,y);
                meanx=sumx/8;
                meany=sumy/8;
                b= (sumxy - 8*meanx*meany)/(sumxx-8*meanx*meanx);
                a= meany - b*meanx;

/*              Y=a+b*(x);
                if (Y > 255) Y=255;
                OPI(x,y)= Y;

                Y=a+b*(x+1);
                if (Y > 255) Y=255;
                OPI(x+2,y)=Y;
*/
                Y=a+b*(x+2);
                if (Y > 255) Y=255;
                OPI(x+4,y)=Y;

                Y=a+b*(x+3);
                if (Y > 255) Y=255;
                OPI(x+6,y)=Y;

                Y=a+b*(x+4);
                if (Y > 255) Y=255;
                OPI(x+8,y)= Y;

                Y=a+b*(x+5);
                if (Y > 255) Y=255;
                OPI(x+10,y)=Y;
/*
                Y=a+b*(x+6);
                if (Y > 255) Y=255;
                OPI(x+12,y)=Y;

                Y=a+b*(x+7);
                if (Y > 255) Y=255;
                OPI(x+14,y)=Y;
*/

        }
    }
    /* copy back */
    for (y=2; y<maxy-2; y++) {
        for (x=2; x<maxx-2; x++) {
            PIX(x,y)=OPI(x,y);
        }
    }
```

```
    for (y=2; y<maxy-8; y+=4) {
#ifndef DEMO
        for (x=2; x<maxx-8; x++) {
#else
        for (x=2; x<y; x++) {
#endif
                sumxy=
                    (y)*PIX((x),y) +
                    (y+1)*PIX((x),y+1)+
                    (y+2)*PIX((x),y+2)+
                    (y+3)*PIX((x),y+3)+
                    (y+4)*PIX((x),y+4) +
                    (y+5)*PIX((x),y+5)+
                    (y+6)*PIX((x),y+6)+
                    (y+7)*PIX((x),y+7);
                sumx=
                    (y)+
                    (y+1)+
                    (y+2)+
                    (y+3)+
                    (y+4)+
                    (y+5)+
                    (y+6)+
                    (y+7);
                sumxx=
                    (y)*(x)+
                    (y+1)*(y+1)+
                    (y+2)*(y+2)+
                    (y+3)*(y+3)+
                    (y+4)*(y+4)+
                    (y+5)*(y+5)+
                    (y+6)*(y+6)+
                    (y+7)*(y+7);
                sumy=
                    PIX(x,y)+
                    PIX(x,y+1)+
                    PIX(x,y+2)+
                    PIX(x,y+3)+
                    PIX(x,y+4)+
                    PIX(x,y+5)+
                    PIX(x,y+6)+
                    PIX(x,y+7);
                meanx=sumx/8;
                meany=sumy/8;
                b= (sumxy - 8*meanx*meany)/(sumxx-8*meanx*meanx);
                a= meany - b*meanx;

    /*          Y=a+b*(y);
                if (Y > 255) Y=255;
```

```
                OPI(x,y)=Y;

                Y=a+b*(y+1);
                if (Y > 255) Y=255;
                OPI(x,y+1)=Y;
    */
                Y=a+b*(y+2);
                if (Y > 255) Y=255;
                OPI(x,y+2)=Y;

                Y=a+b*(y+3);
                if (Y > 255) Y=255;
                OPI(x,y+3)=Y;

                Y=a+b*(y+4);
                if (Y > 255) Y=255;
                OPI(x,y+4)= Y;

                Y=a+b*(y+5);
                if (Y > 255) Y=255;
                OPI(x,y+5)=Y;
    /*
                Y=a+b*(y+6);
                if (Y > 255) Y=255;
                OPI(x,y+6)=Y;

                Y=a+b*(y+7);
                if (Y > 255) Y=255;
                OPI(x,y+7)=Y;
    */
        }
    }

}
```

C.3 pgm_linregr_adapt.c

```
void simple_linregr_adapt_filter(unsigned char ** in, unsigned char ** out, int * xsize, int * ysize) {
#define PIX(a,b) ( tin[((b) * maxx) + (a)])
#define OPI(a,b) ( tout[((b) * maxx) + (a)])
/* SMOOTHNESS defines Threshold for impuls-detection, should be between 0.01 and 0.06, a good value is 0.05 */
#define SMOOTHNESS (0.05)
/* MAXSTEPWIDE defines maximum count of values which should be in a linregression, default is 8 */
```

```
#define MAXSTEPWIDE (8)
    unsigned char * tin;
    unsigned char * tout;
    int maxx, maxy;
    int i,Y;
    int x,y;
    int j;
    float a, b; /* a+bx */
    float meanx, meany;
    float sumxy, sumxx;
    float sumx, sumy;
    long int prevmean;
    long int actmean;
    assert(*in != NULL);
    assert(*out == NULL);
    maxx= *xsize;
    maxy= *ysize;
    /* alloc memory to store stuff */
    * out = (unsigned char *) malloc(maxx * maxy *sizeof(unsigned char)); /* should be freed, if closed */
    if (* out == NULL) {
        fprintf(stderr, "could not allocate memory for output-buffer\n");
        exit(-1);
    }
    /* init with original image */
    *out = memcpy(*out, *in, maxx * maxy * sizeof(unsigned char));
    tin= *in;
    tout= *out;
    /* implement a simple linregr */
#ifndef BLXA
    for (y=4; y<maxy-4; y++) {
#ifndef DEMO
        for (x=4; x<maxx-4; ) {
#else
        for (x=4; x<y; ) {
#endif
            /* search endwindow */
            /* calc prevmean */
            prevmean=
            (
                PIX(x+1,y)+
                PIX(x+2,y)+
                PIX(x+3,y)+
                PIX(x+4,y)+
                PIX(x-4,y)+
                PIX(x-3,y)+
                PIX(x-2,y)+
                PIX(x-1,y)+
                PIX(x-0,y)
            );
```

```
for (j=1; j<maxx-4-x;j++) { /* to slide window forwards */
    if (j>MAXSTEPWIDE) break;
    /* calc actmean */
    actmean=
    (
        PIX(x+0+j,y)+
        PIX(x+1+j,y)+
        PIX(x+2+j,y)+
        PIX(x+3+j,y)+
        PIX(x+4+j,y)+
        PIX(x-1+j,y)+
        PIX(x-2+j,y)+
        PIX(x-3+j,y)+
        PIX(x-4+j,y)
        );
        /* filter noisy peaks out */
        if (abs(PIX(x+j,y)*9-(actmean))>(0.05*actmean)) {
            /* substitute it with 3x3 median */
            //actmean=actmean*9;
            actmean=actmean-PIX(x+j,y);
            PIX(x+j,y)=median(PIX(x+j-1,y),PIX(x+j,y),PIX(x+j+1,y));
            actmean=actmean+PIX(x+j,y);
            //actmean=actmean/9;
        }

        if (delta(prevmean,actmean,j)==0) {/* same thing as before, slide window forwards */
        ;
    }
    else {/* new situation, mark window position, recalc regression and forward startpoint to window pos
        break; /* break out the for loop, to do the things described above */
    }
}
sumxy=0;
sumx=0;
sumxx=0;
sumy=0;
meanx=0;
meany=0;
for (i=x; i<=(x+j); i++) {
    sumxy+=
        (i)*PIX(i,y);
    sumx+=(i);
    sumxx+=(i)*(i);
    sumy+=
        PIX(i,y);
}
meanx=sumx/j;
meany=sumy/j;
b= (sumxy - j*meanx*meany)/(sumxx-j*meanx*meanx);
a= meany - b*meanx;
```

```
            for (i=x; i<=(x+j); i++) {
                Y=a+b*(i);
                if (Y > 255) Y=255;
                if (Y < 0) Y=0;
                OPI(i,y)= Y;
            }
            x+=j; /* set actpos to windowpos */
        }
    }


    /* copy back
    for (y=2; y<maxy-2; y++) {
        for (x=2; x<maxx-2; x++) {
            PIX(x,y)=OPI(x,y);
        }
    }

    */

#endif

    for (x=4; x<maxx-4; x++) {
        for (y=4; y<maxy-4; ) {
            /* search endwindow */
            /* calc prevmean */
            prevmean=
            (
                PIX(x,y+1)+
                PIX(x,y+2)+
                PIX(x,y+3)+
                PIX(x,y+4)+
                PIX(x,y-4)+
                PIX(x,y-3)+
                PIX(x,y-2)+
                PIX(x,y-1)+
                PIX(x,y)
            );

            for (j=1; j<maxy-4-y;j++) { /* to slide window forwards */
                if (j>MAXSTEPWIDE) break;
                /* calc actmean */
                actmean=
                (
                    PIX(x,y+0+j)+
                    PIX(x,y+1+j)+
                    PIX(x,y+2+j)+
                    PIX(x,y+3+j)+
                    PIX(x,y+4+j)+
                    PIX(x,y-1+j)+
```

```
                    PIX(x,y−2+j)+
                    PIX(x,y−3+j)+
                    PIX(x,y−4+j)
160                 );
                    /* filter noisy peaks out */
                    if (abs(PIX(x+j,y)*9−(actmean))>(0.05*actmean)) {
//                  if (abs(PIX(x,y+j)*9−actmean)>(7)) {
                        /* substitute it with 3x3 median */
                        //actmean=actmean*9;
                        actmean=actmean−PIX(x,y+j);
                        PIX(x,y+j)=median(PIX(x,y+j−1),PIX(x,y+j),PIX(x,y+j+1));
                        actmean=actmean+PIX(x,y+j);
                        //actmean=actmean/9;
170                 }

                    if (delta(prevmean,actmean,j)==0) {/* same thing as before, slide window forwards */
                    ;
                }
                else {/* new situation, mark window position, recalc regression and forward startpoint to window pos
                    break; /* break out the for loop, to do the things described above */
                }
            }
            sumxy=0;
180         sumx=0;
            sumxx=0;
            sumy=0;
            meanx=0;
            meany=0;
            for (i=y; i<=(y+j); i++) {
                sumxy+=
                    (i)*PIX(x,i);
                sumx+=(i);
                sumxx+=(i)*(i);
190             sumy+=
                    PIX(x,i);
            }
            meanx=sumx/j;
            meany=sumy/j;
            b= (sumxy − j*meanx*meany)/(sumxx−j*meanx*meanx);
            a= meany − b*meanx;
            for (i=y; i<=(y+j); i++) {
                Y=a+b*(i);
                if (Y > 255) Y=255;
200             if (Y < 0) Y=0;
                OPI(x,i)=((Y)+PIX(x,i))/2;
            }

            y+=j; /* set actpos to windowpos */
        }
    }
```

```
}
```

C.4 pgm_linregr.c

```
void simple_linregr_filter(unsigned char ** in, unsigned char ** out, int * xsize, int * ysize) {
#define PIX(a,b) ( tin[((b) * maxx) + (a)])
#define OPI(a,b) ( tout[((b) * maxx) + (a)])
    unsigned char * tin;
    unsigned char * tout;
    int maxx, maxy;
    int Y;
    int x,y;

    float a, b; /* a+bx */
    float meanx, meany;
    float sumxy, sumxx;
    float sumx, sumy;

    assert(*in != NULL);
    assert(*out == NULL);
    maxx= *xsize;
    maxy= *ysize;
    /* alloc memory to store stuff */
    * out = (unsigned char *) malloc(maxx * maxy *sizeof(unsigned char)); /* should be freed, if closed */
    if (* out == NULL) {
        fprintf(stderr, "could not allocate memory for output-buffer\n");
        exit(-1);
    }
    /* init with original image */
    *out = memcpy(*out, *in, maxx * maxy * sizeof(unsigned char));
    tin= *in;
    tout= *out;
    /* implement a simple linregr */
    for (y=2; y<maxy-8; y+=1) {
#ifndef DEMO
        for (x=2; x<maxx-8; x+=8) {
#else
        for (x=2; x<y; x+=8) {
#endif
                sumxy=
                    (x)*PIX((x),y) +
                    (x+1)*PIX((x+1),y)+
```

```
                (x+2)*PIX((x+2),y)+
                (x+3)*PIX((x+3),y)+
                (x+4)*PIX((x+4),y) +
                (x+5)*PIX((x+5),y)+
                (x+6)*PIX((x+6),y)+
                (x+7)*PIX((x+7),y);
            sumx=
                (x)+
                (x+1)+
                (x+2)+
                (x+3)+
                (x+4)+
                (x+5)+
                (x+6)+
                (x+7);
            sumxx=
                (x)*(x)+
                (x+1)*(x+1)+
                (x+2)*(x+2)+
                (x+3)*(x+3)+
                (x+4)*(x+4)+
                (x+5)*(x+5)+
                (x+6)*(x+6)+
                (x+7)*(x+7);
            sumy=
                PIX(x,y)+
                PIX(x+1,y)+
                PIX(x+2,y)+
                PIX(x+3,y)+
                PIX(x+4,y)+
                PIX(x+5,y)+
                PIX(x+6,y)+
                PIX(x+7,y);
            meanx=sumx/8;
            meany=sumy/8;
            b= (sumxy - 8*meanx*meany)/(sumxx-8*meanx*meanx);
            a= meany - b*meanx;

            Y=a+b*(x);
            if (Y > 255) Y=255;
            OPI(x,y)= Y;

            Y=a+b*(x+1);
            if (Y > 255) Y=255;
            OPI(x+2,y)=Y;

            Y=a+b*(x+2);
            if (Y > 255) Y=255;
            OPI(x+4,y)=Y;
```

```
                Y=a+b*(x+3);
                if (Y > 255) Y=255;
                OPI(x+6,y)=Y;

                Y=a+b*(x+4);
                if (Y > 255) Y=255;
                OPI(x+8,y)= Y;

                Y=a+b*(x+5);
                if (Y > 255) Y=255;
                OPI(x+10,y)=Y;

                Y=a+b*(x+6);
                if (Y > 255) Y=255;
                OPI(x+12,y)=Y;

                Y=a+b*(x+7);
                if (Y > 255) Y=255;
                OPI(x+14,y)=Y;


        }
    }
    /* copy back */
    for (y=2; y<maxy-2; y++) {
        for (x=2; x<maxx-2; x++) {
            PIX(x,y)=OPI(x,y);
        }
    }

    for (y=2; y<maxy-8; y+=8) {
#ifndef DEMO
        for (x=2; x<maxx-8; x+=1) {
#else
        for (x=2; x<y; x+=1) {
#endif
                sumxy=
                    (y)*PIX((x),y) +
                    (y+1)*PIX((x),y+1)+
                    (y+2)*PIX((x),y+2)+
                    (y+3)*PIX((x),y+3)+
                    (y+4)*PIX((x),y+4) +
                    (y+5)*PIX((x),y+5)+
                    (y+6)*PIX((x),y+6)+
                    (y+7)*PIX((x),y+7);
                sumx=
                    (y)+
                    (y+1)+
                    (y+2)+
                    (y+3)+
```

```
                (y+4)+
                (y+5)+
                (y+6)+
                (y+7);
            sumxx=
                (y)*(x)+
                (y+1)*(y+1)+
                (y+2)*(y+2)+
                (y+3)*(y+3)+
                (y+4)*(y+4)+
                (y+5)*(y+5)+
                (y+6)*(y+6)+
                (y+7)*(y+7);
            sumy=
                PIX(x,y)+
                PIX(x,y+1)+
                PIX(x,y+2)+
                PIX(x,y+3)+
                PIX(x,y+4)+
                PIX(x,y+5)+
                PIX(x,y+6)+
                PIX(x,y+7);
            meanx=sumx/8;
            meany=sumy/8;
            b= (sumxy - 8*meanx*meany)/(sumxx-8*meanx*meanx);
            a= meany - b*meanx;

            Y=a+b*(y+1);
            if (Y > 255) Y=255;
            OPI(x,y+1)=Y;

            Y=a+b*(y+2);
            if (Y > 255) Y=255;
            OPI(x,y+2)=Y;

            Y=a+b*(y+3);
            if (Y > 255) Y=255;
            OPI(x,y+3)=Y;

            Y=a+b*(y+4);
            if (Y > 255) Y=255;
            OPI(x,y+4)= Y;

            Y=a+b*(y+5);
            if (Y > 255) Y=255;
            OPI(x,y+5)=Y;

            Y=a+b*(y+6);
            if (Y > 255) Y=255;
            OPI(x,y+6)=Y;
```

```
                Y=a+b*(y+7);
                if (Y > 255) Y=255;
                OPI(x,y+7)=Y;
            }
        }

    }
```

C.5 pgm_mean.c

```
    void simple_mean_filter(unsigned char ** in, unsigned char ** out, int * xsize, int * ysize) {
    #define PIX(a,b) ( tin[((b) * maxx) + (a)])
    #define OPI(a,b) ( tout[((b) * maxx) + (a)])
        unsigned char * tin;
        unsigned char * tout;
        int maxx, maxy;
        int x, y;
        assert(*in != NULL);
        assert(*out == NULL);
        maxx= *xsize;
        maxy= *ysize;
        /* alloc memory to store stuff */
        * out = (unsigned char *) malloc(maxx * maxy *sizeof(unsigned char)); /* should be freed, if closed */
        if (* out == NULL) {
            fprintf(stderr, "could not allocate memory for output-buffer\n");
            exit(-1);
        }
        /* init with original image */
        *out = memcpy(*out, *in, maxx * maxy * sizeof(unsigned char));
        tin= *in;
        tout= *out;
        /* implement a simple mean */
        for (y=2; y<maxy-2; y++) {
    #ifndef DEMO
            for (x=2; x<maxx-2; x++) {
    #else
            for (x=2; x<y; x++) {
    #endif
                OPI(x,y)=(
                    PIX(x-2,y)+
                    PIX(x-1,y)+
                    PIX(x,y)+
```

```
                PIX(x+1,y)+
                PIX(x+2,y))/5;
        }
    }
    /* copy back */
    for (y=2; y<maxy-2; y++) {
        for (x=2; x<maxx-2; x++) {
            PIX(x,y)=OPI(x,y);
        }
    }
    for (y=2; y<maxy-2; y++) {
#ifndef DEMO
        for (x=2; x<maxx-2; x++) {
#else
        for (x=2; x<y; x++) {
#endif
            OPI(x,y)=(
                PIX(x,y-2)+
                PIX(x,y-1)+
                PIX(x,y)+
                PIX(x,y+1)+
                PIX(x,y+2))/5;
        }
    }
}
```

C.6 pgm_median.c

```
void simple_median_filter(unsigned char ** in, unsigned char ** out, int * xsize, int * ysize) {
#define PIX(a,b) ( tin[((b) * maxx) + (a)])
#define OPI(a,b) ( tout[((b) * maxx) + (a)])
    unsigned char * tin;
    unsigned char * tout;
    int maxx, maxy;
    int x, y;
    int i, j, min,Y;
    int temp[25]; /* 5x5 matrix */
    assert(*in != NULL);
    assert(*out == NULL);
    maxx= *xsize;
    maxy= *ysize;
    /* alloc memory to store stuff */
    * out = (unsigned char *) malloc(maxx * maxy *sizeof(unsigned char)); /* should be freed, if closed */
```

```
    if (* out == NULL) {
        fprintf(stderr, "could not allocate memory for output-buffer\n");
        exit(-1);
    }
    /* init with original image */
    *out = memcpy(*out, *in, maxx * maxy * sizeof(unsigned char));
    tin= *in;
    tout= *out;
    /* implement a simple median */
    for (y=2; y<maxy-2; y++) {
#ifndef DEMO
        for (x=2; x<maxx-2; x++) {
#else
        for (x=2; x<y; x++) {
#endif
            temp[0]=PIX(x-2,y-2);
            temp[1]=PIX(x-1,y-2);
            temp[2]=PIX(x-0,y-2);
            temp[3]=PIX(x+1,y-2);
            temp[4]=PIX(x+2,y-2);
            temp[5]=PIX(x-2,y-1);
            temp[6]=PIX(x-1,y-1);
            temp[7]=PIX(x-0,y-1);
            temp[8]=PIX(x+1,y-1);
            temp[9]=PIX(x+2,y-1);
            temp[10]=PIX(x-2,y);
            temp[11]=PIX(x-1,y);
            temp[12]=PIX(x-0,y);
            temp[13]=PIX(x+1,y);
            temp[14]=PIX(x+2,y);
            temp[15]=PIX(x-2,y+1);
            temp[16]=PIX(x-1,y+1);
            temp[17]=PIX(x-0,y+1);
            temp[18]=PIX(x+1,y+1);
            temp[19]=PIX(x+2,y+1);
            temp[20]=PIX(x-2,y+2);
            temp[21]=PIX(x-1,y+2);
            temp[22]=PIX(x-0,y+2);
            temp[23]=PIX(x+1,y+2);
            temp[24]=PIX(x+2,y+2);
            Y=0;
            for (i=0; i<24-1; i++) { /* selection sort temp to get the median */
                min=i;
                for (j=i+1; j<24; j++) {
                    if (temp[j]<temp[min]) {
                        min=j;
                        Y=temp[min]; temp[min]=temp[i]; temp[i]=Y;
                    }
                }
            }
```

```
            OPI(x,y)=temp[4];
        }
    }
}
```

C.7 pgm_medmean.c

```
void simple_medmean_filter(unsigned char ** in, unsigned char ** out, int * xsize, int * ysize) {
#define PIX(a,b) ( tin[((b) * maxx) + (a)])
#define OPI(a,b) ( tout[((b) * maxx) + (a)])
    unsigned char * tin;
    unsigned char * tout;
    int maxx, maxy;
    int x, y;
    int i,Y;
    int mdi, mdj, min;
    int temp[25]; /* 5x5 matrix */
    assert(*in != NULL);
    assert(*out == NULL);
    maxx= *xsize;
    maxy= *ysize;
    /* alloc memory to store stuff */
    * out = (unsigned char *) malloc(maxx * maxy *sizeof(unsigned char)); /* should be freed, if closed */
    if (* out == NULL) {
        fprintf(stderr, "could not allocate memory for output-buffer\n");
        exit(-1);
    }
    /* init with original image */
    *out = memcpy(*out, *in, maxx * maxy * sizeof(unsigned char));
    tin= *in;
    tout= *out;
    /* implement a simple medmean */
    for (y=2; y<maxy-2; y++) {
#ifndef DEMO
        for (x=2; x<maxx-2; x++) {
#else
        for (x=2; x<y; x++) {
#endif
            temp[0]=PIX(x-2,y-2);
            temp[1]=PIX(x-1,y-2);
            temp[2]=PIX(x-0,y-2);
            temp[3]=PIX(x+1,y-2);
            temp[4]=PIX(x+2,y-2);
```

```
            temp[5]=PIX(x-2,y-1);
            temp[6]=PIX(x-1,y-1);
            temp[7]=PIX(x-0,y-1);
            temp[8]=PIX(x+1,y-1);
            temp[9]=PIX(x+2,y-1);
            temp[10]=PIX(x-2,y);
            temp[11]=PIX(x-1,y);
            temp[12]=PIX(x-0,y);
            temp[13]=PIX(x+1,y);
            temp[14]=PIX(x+2,y);
            temp[15]=PIX(x-2,y+1);
            temp[16]=PIX(x-1,y+1);
            temp[17]=PIX(x-0,y+1);
            temp[18]=PIX(x+1,y+1);
            temp[19]=PIX(x+2,y+1);
            temp[20]=PIX(x-2,y+2);
            temp[21]=PIX(x-1,y+2);
            temp[22]=PIX(x-0,y+2);
            temp[23]=PIX(x+1,y+2);
            temp[24]=PIX(x+2,y+2);
            Y=0;
            for (i=0; i<=24; i++) { /* to get the mean */
                Y+=temp[i];
            }
//          if (abs(PIX(x,y)-(Y/25))>6) { /* over threshold (10 percent of 255 gray levels), median */
            if (abs(PIX(x,y)-(Y/25))>(0.1*(Y/25))) { /* over threshold (10 percent of 255 gray levels), median */
                for (mdi=0; mdi<24-1; mdi++) { /* selection sort temp to get the median */
                    min=mdi;
                    for (mdj=mdi+1; mdj<24; mdj++) {
                        if (temp[mdj]<temp[min]) {
                            min=mdj;
                            Y=temp[min]; temp[min]=temp[mdi]; temp[mdi]=Y;
                        }
                    }
                }
                OPI(x,y)=temp[12];
                //OPI(x,y)=255;
            } else {
                OPI(x,y)=Y/25;
            }
        }
    }
}
```

C.8 pgm_mode.c

```
void simple_mode_filter(unsigned char ** in, unsigned char ** out, int * xsize, int * ysize) {
#define PIX(a,b) ( tin[((b) * maxx) + (a)])
#define OPI(a,b) ( tout[((b) * maxx) + (a)])
    unsigned char * tin;
    unsigned char * tout;
    int maxx, maxy;
    int x, y;
    int i, j, max=0,Y;
    int temp[256];
    assert(*in != NULL);
    assert(*out == NULL);
    maxx= *xsize;
    maxy= *ysize;
    /* alloc memory to store stuff */
    * out = (unsigned char *) malloc(maxx * maxy *sizeof(unsigned char)); /* should be freed, if closed */
    if (* out == NULL) {
        fprintf(stderr, "could not allocate memory for output-buffer\n");
        exit(-1);
    }
    /* init with original image */
    *out = memcpy(*out, *in, maxx * maxy * sizeof(unsigned char));
    tin= *in;
    tout= *out;
    /* implement a simple mode */
    for (y=2; y<maxy-2; y++) {
#ifndef DEMO
        for (x=2; x<maxx-2; x++) {
#else
        for (x=2; x<y; x++) {
#endif
            for (i=0; i< 255; i++) { /* empty all */
                temp[i]= 0;
            }
            for (i= -2; i<2; i++) { /* 5x5 window */
                for (j= -2; j<2; j++) {
                    temp[PIX(x+i,y+j)]++;
                }
            }
            Y=0;
            for (i=0; i< 255; i++) { /* search max */
                if (temp[i]>Y) {
                    Y=temp[i];
                    max=i;
                }
            }
            if (max>1) OPI(x,y)=max;
            else OPI(x,y)=PIX(x,y);
```

```
        }
    }
}
```

C.9 pgm_wmean.c

```
void simple_wmean_filter(unsigned char ** in, unsigned char ** out, int * xsize, int * ysize) {
#define PIX(a,b) ( tin[((b) * maxx) + (a)])
#define OPI(a,b) ( tout[((b) * maxx) + (a)])
    unsigned char * tin;
    unsigned char * tout;
    int maxx, maxy;
    int x, y;

    assert(*in != NULL);
    assert(*out == NULL);
    maxx= *xsize;
    maxy= *ysize;
    /* alloc memory to store stuff */
    * out = (unsigned char *) malloc(maxx * maxy *sizeof(unsigned char)); /* should be freed, if closed */
    if (* out == NULL) {
        fprintf(stderr, "could not allocate memory for output-buffer\n");
        exit(-1);
    }
    /* init with original image */
    *out = memcpy(*out, *in, maxx * maxy * sizeof(unsigned char));
    tin= *in;
    tout= *out;
    /* implement a simple weighted mean */
    for (y=2; y<maxy-2; y++) {
#ifndef DEMO
        for (x=2; x<maxx-2; x++) {
#else
        for (x=2; x<y; x++) {
#endif
            OPI(x,y)=(
                1*PIX(x-2,y)+
                4*PIX(x-1,y)+
                6*PIX(x,y)+
                4*PIX(x+1,y)+
                1*PIX(x+2,y))/16;
        }
    }
```

```
    /* copy back */
    for (y=2; y<maxy-2; y++) {
        for (x=2; x<maxx-2; x++) {
            PIX(x,y)=OPI(x,y);
        }
    }
    for (y=2; y<maxy-2; y++) {
#ifndef DEMO
        for (x=2; x<maxx-2; x++) {
#else
        for (x=2; x<y; x++) {
#endif
            OPI(x,y)=(
                1*PIX(x,y-2)+
                4*PIX(x,y-1)+
                6*PIX(x,y)+
                4*PIX(x,y+1)+
                1*PIX(x,y+2))/16;
        }
    }
}
```

D Bildverzeichnis

(a) barbara.pgm.autocorr.png

(b) barbara.pgm.linregr2.png

(c) barbara.pgm.linregr.adapt.png

(d) barbara.pgm.linregr.png

Abbildung D.1:

(a) barbara.pgm.mean.png

(b) barbara.pgm.median.png

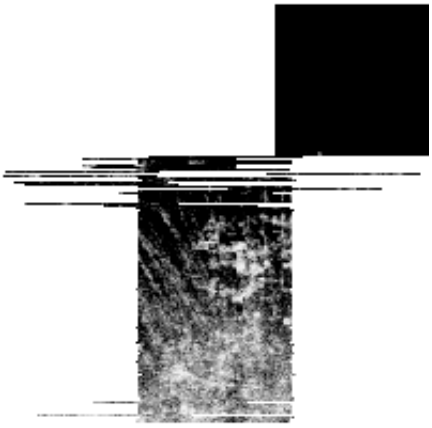
(c) barbara.pgm.medmean.png

(d) barbara.pgm.mode.png

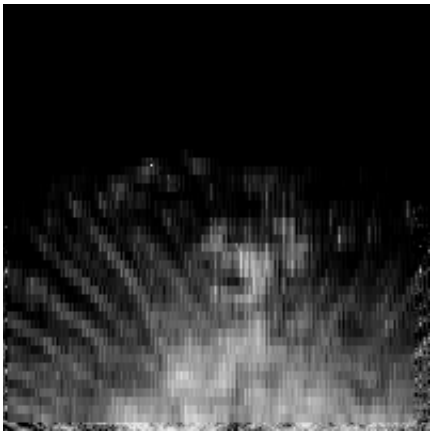
Abbildung D.2:

(a) barbara.pgm.wmean.png

(b) barbara.png

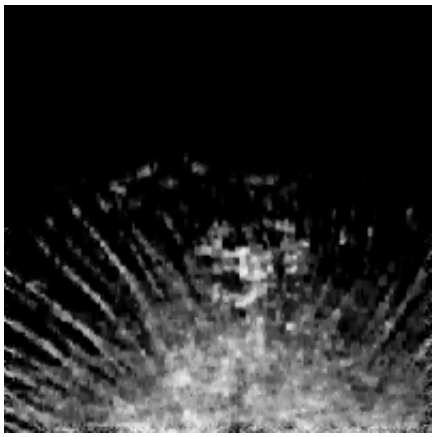


(c) cell1a.pgm.autocorr.png

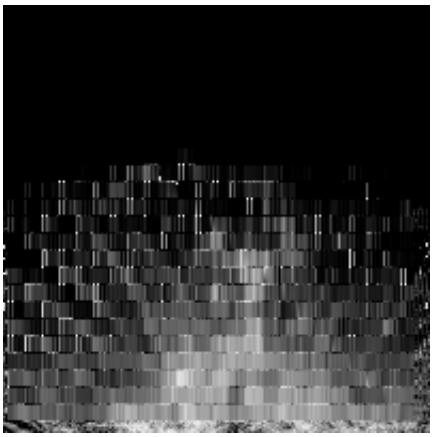


(d) cell1a.pgm.linregr2.png

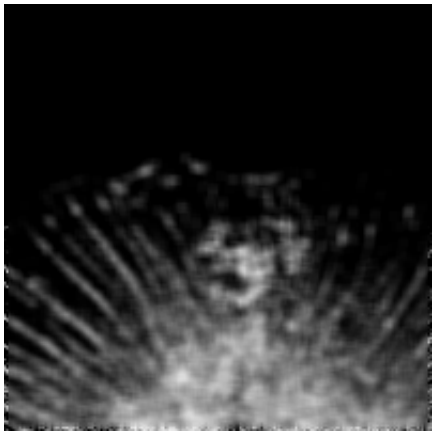
Abbildung D.3:



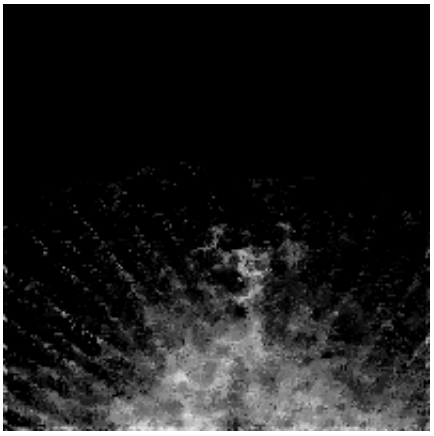
(a) cell1a.pgm.linregr.adapt.png



(b) cell1a.pgm.linregr.png

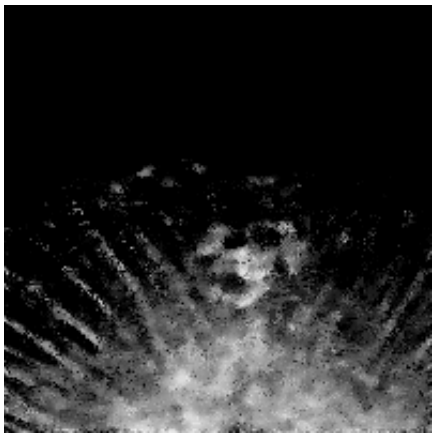


(c) cell1a.pgm.mean.png

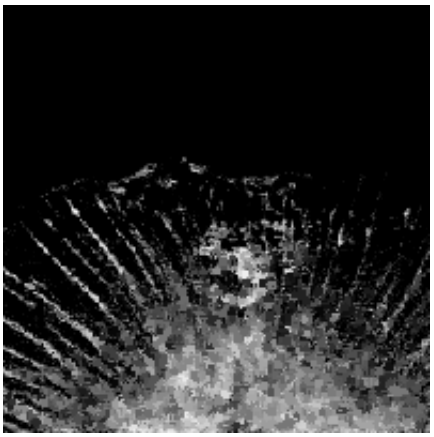


(d) cell1a.pgm.median.png

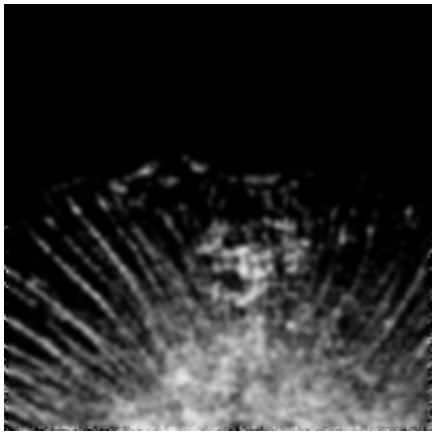
Abbildung D.4:



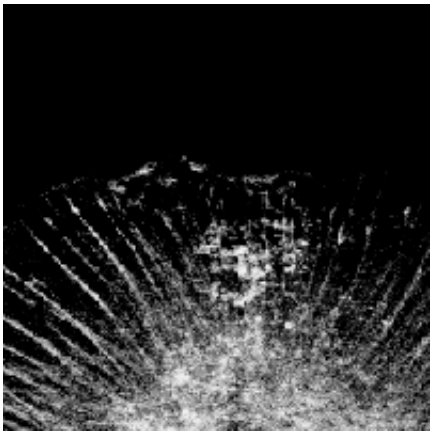
(a) cell1a.pgm.medmean.png



(b) cell1a.pgm.mode.png

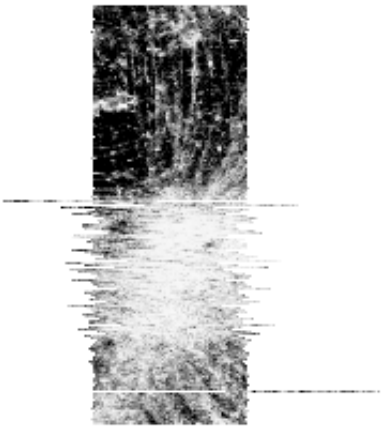


(c) cell1a.pgm.wmean.png

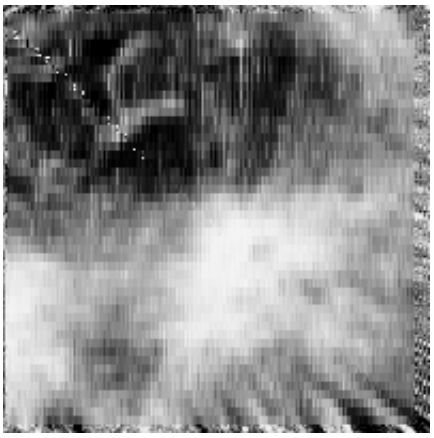


(d) cell1a.png

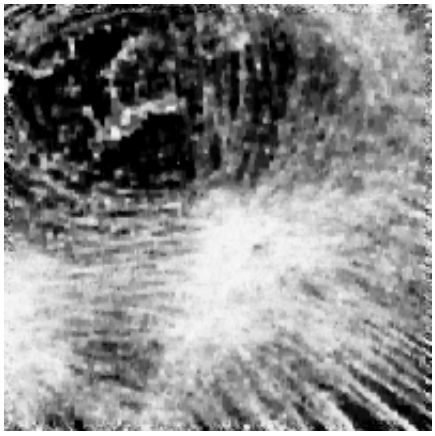
Abbildung D.5:



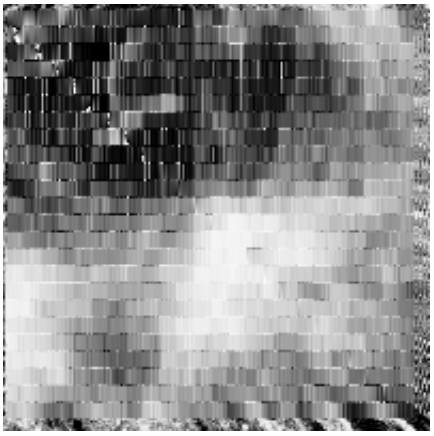
(a) cell1b.pgm.autocorr.png



(b) cell1b.pgm.linregr2.png

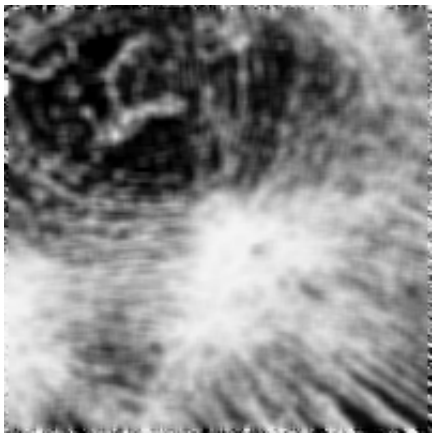


(c) cell1b.pgm.linregr.adapt.png

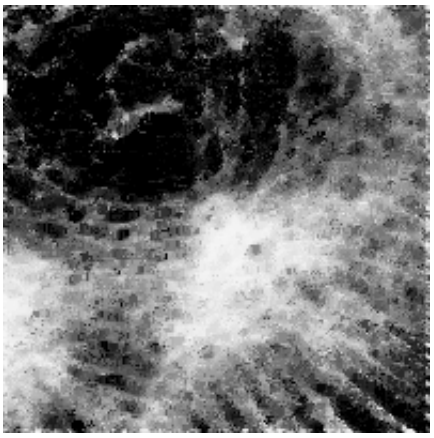


(d) cell1b.pgm.linregr.png

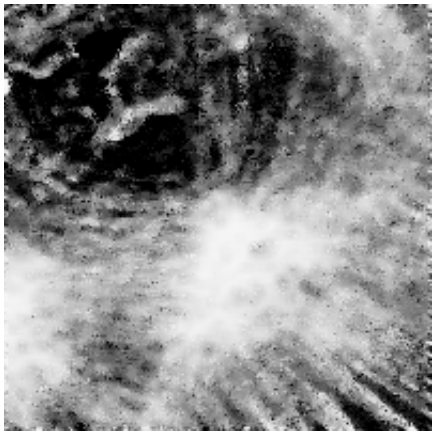
Abbildung D.6:



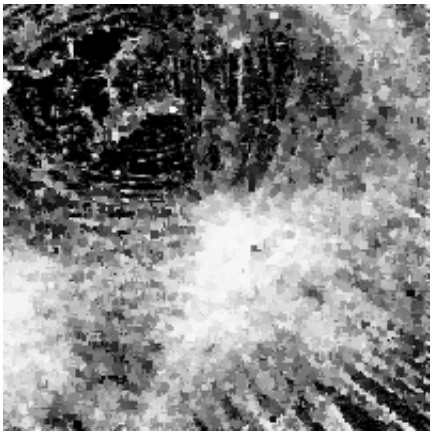
(a) cell1b.pgm.mean.png



(b) cell1b.pgm.median.png

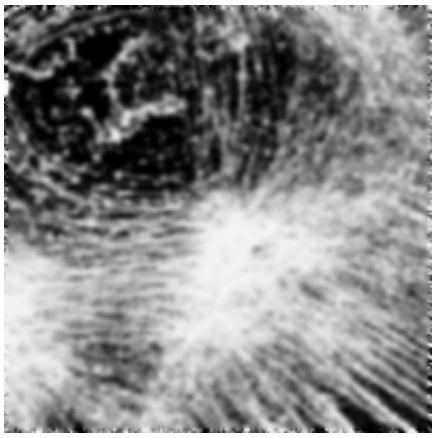


(c) cell1b.pgm.medmean.png

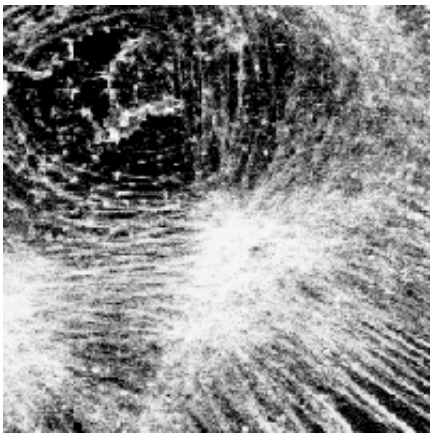


(d) cell1b.pgm.mode.png

Abbildung D.7:



(a) cell1b.pgm.wmean.png

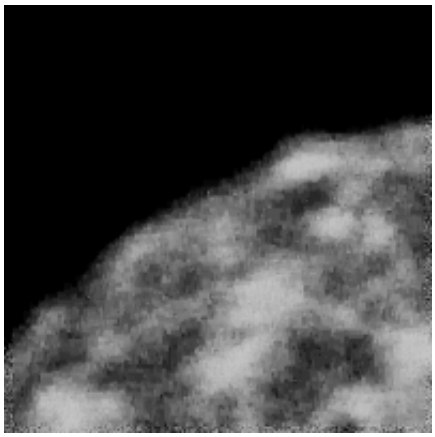


(b) cell1b.png

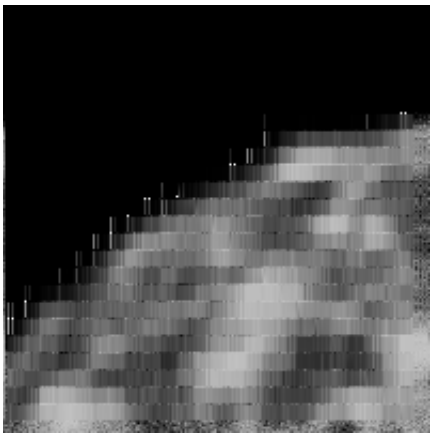
(c) cell2.pgm.autocorr.png

(d) cell2.pgm.linregr2.png

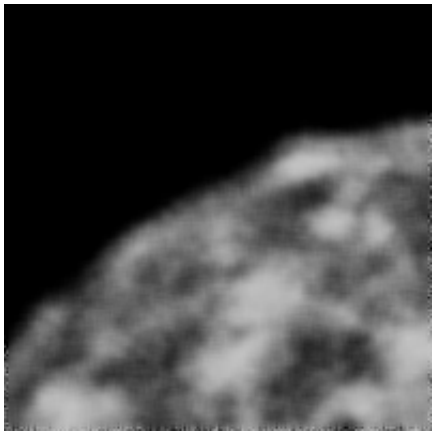
Abbildung D.8:



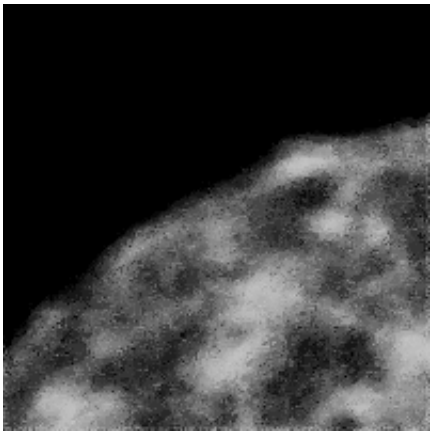
(a) cell2.pgm.linregr.adapt.png



(b) cell2.pgm.linregr.png

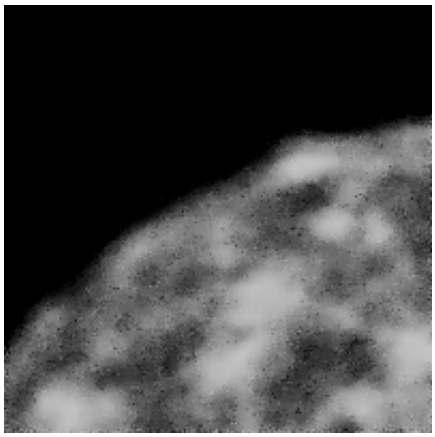


(c) cell2.pgm.mean.png



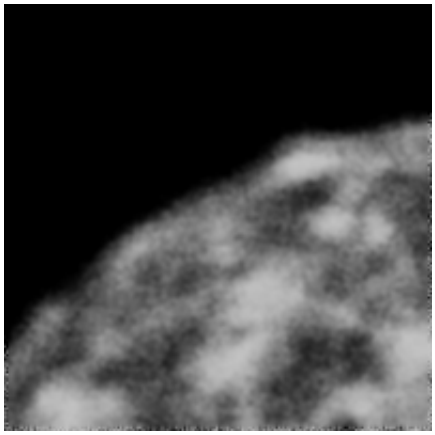
(d) cell2.pgm.median.png

Abbildung D.9:

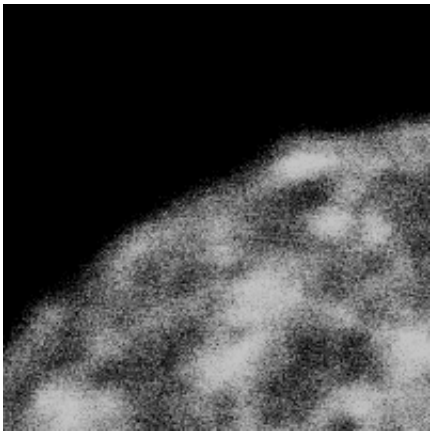


(a) cell2.pgm.medmean.png

(b) cell2.pgm.mode.png



(c) cell2.pgm.wmean.png



(d) cell2.png

Abbildung D.10:

(a) elle.pgm.autocorr.png

(b) elle.pgm.linregr2.png

(c) elle.pgm.linregr.adapt.png

(d) elle.pgm.linregr.png

Abbildung D.11:

(a) elle.pgm.mean.png

(b) elle.pgm.median.png

(c) elle.pgm.medmean.png

(d) elle.pgm.mode.png

Abbildung D.12:

(a) elle.pgm.wmean.png

(b) elle.png

(c) GONY.impulse.pgm.autocorr.png

(d) GONY.impulse.pgm.linregr2.png

Abbildung D.13:

(a)
GONY.impulse.pgm.linregr.adapt.png

(b)
GONY.impulse.pgm.linregr.png

(c)
GONY.impulse.pgm.mean.png

(d)
GONY.impulse.pgm.median.png

Abbildung D.14:

(a)
GONY.impulse.pgm.medmean.png

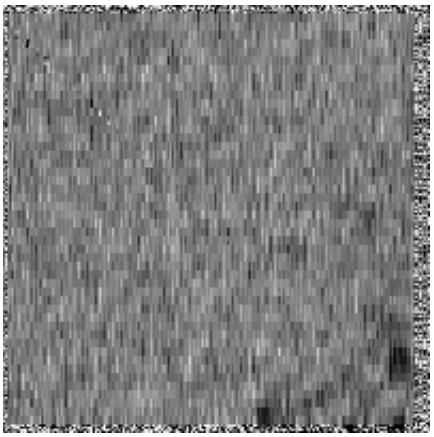
(b)
GONY.impulse.pgm.mode.png

(c)
GONY.impulse.pgm.wmean.png

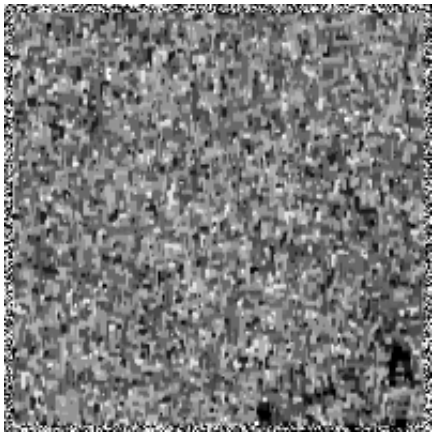
(d) GONY.impulse.png

Abbildung D.15:

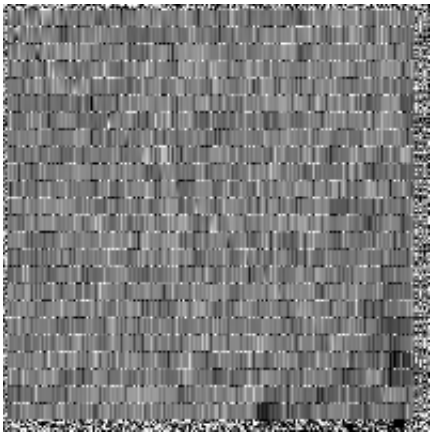
(a)
GONY.multipl.pgm.autocorr.png



(b)
GONY.multipl.pgm.linregr2.png

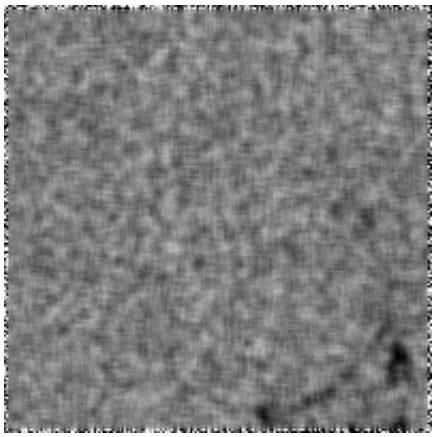


(c)
GONY.multipl.pgm.linregr.adapt.png

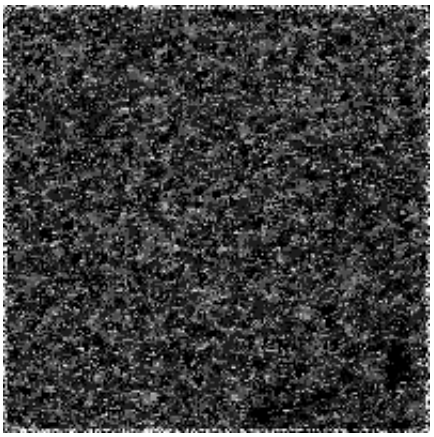


(d)
GONY.multipl.pgm.linregr.png

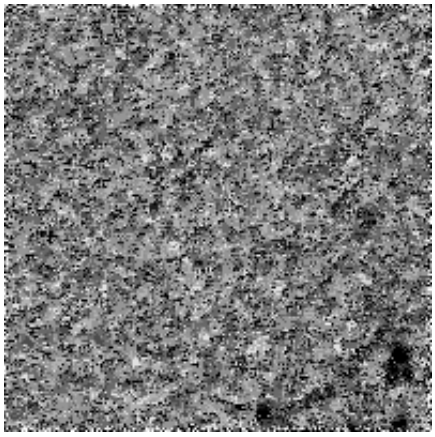
Abbildung D.16:



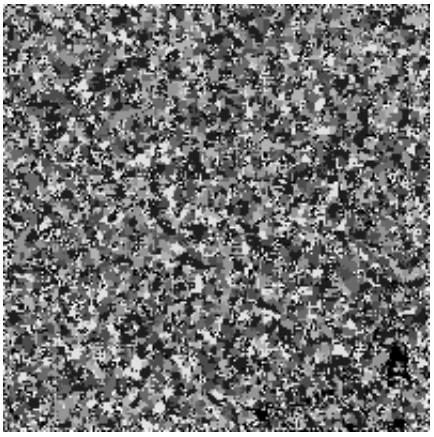
(a)
GONY.multipl.pgm.mean.png



(b)
GONY.multipl.pgm.median.png

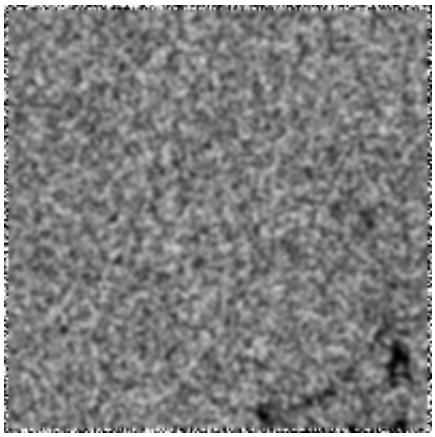


(c)
GONY.multipl.pgm.medmean.png

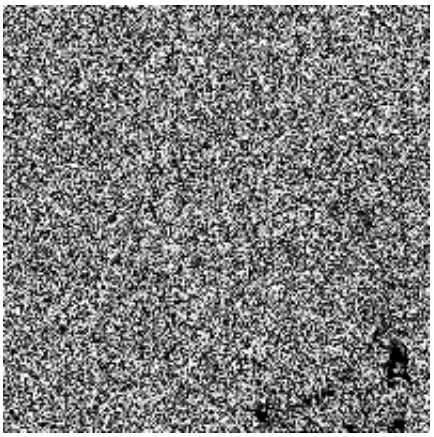


(d)
GONY.multipl.pgm.mode.png

Abbildung D.17:

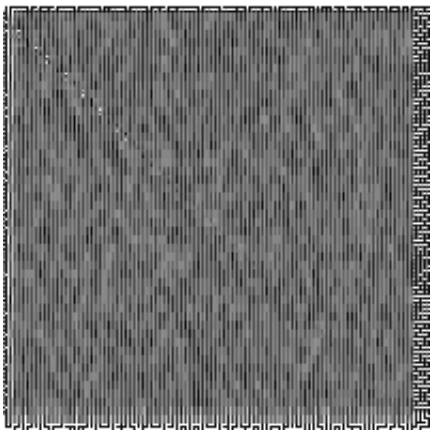


(a) GONY.multipl.pgm.wmean.png



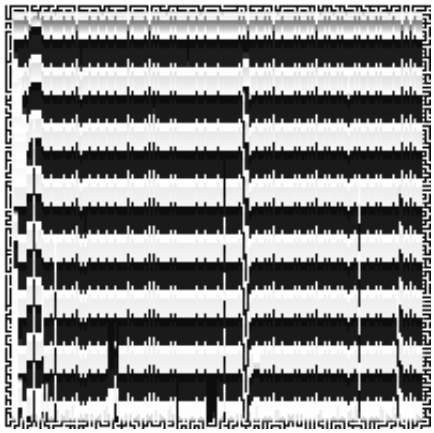
(b) GONY.multipl.png

(c) labyrinth.pgm.autocorr.png

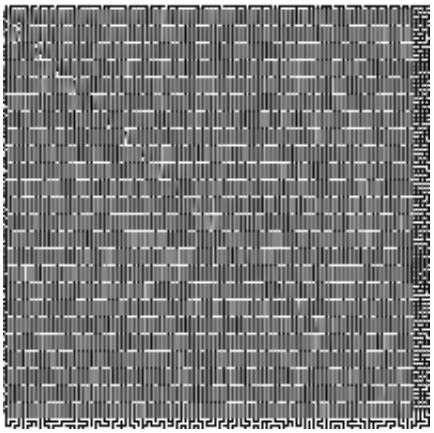


(d) labyrinth.pgm.linregr2.png

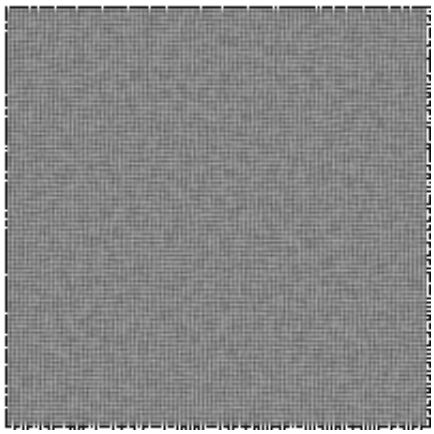
Abbildung D.18:



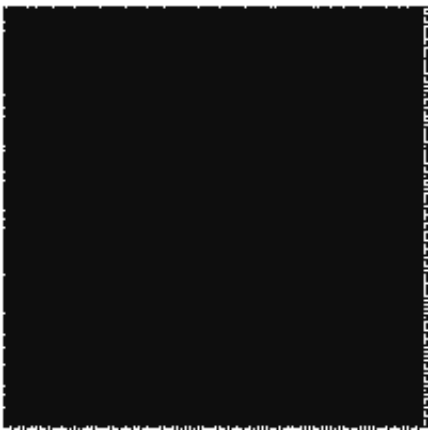
(a) labyrinth.pgm.linregr.adapt.png



(b) labyrinth.pgm.linregr.png

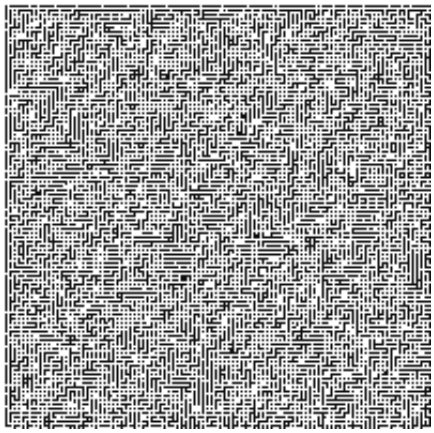


(c) labyrinth.pgm.mean.png

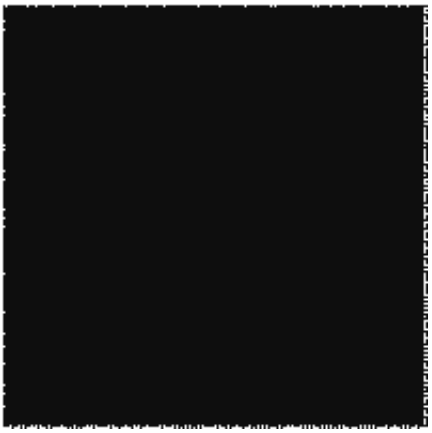


(d) labyrinth.pgm.median.png

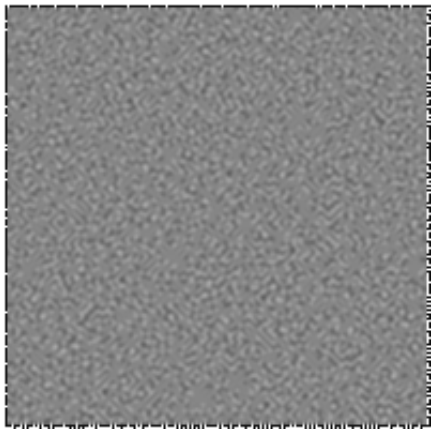
Abbildung D.19:



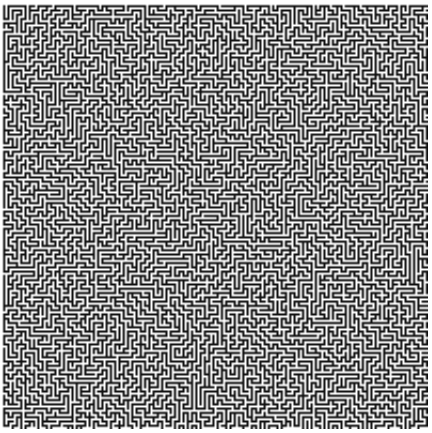
(a) labyrinth.pgm.medmean.png



(b) labyrinth.pgm.mode.png



(c) labyrinth.pgm.wmean.png



(d) labyrinth.png

Abbildung D.20:

(a) lea.pgm.autocorr.png

(b) lea.pgm.linregr2.png

(c) lea.pgm.linregr.adapt.png

(d) lea.pgm.linregr.png

Abbildung D.21:

(a) lea.pgm.mean.png

(b) lea.pgm.median.png

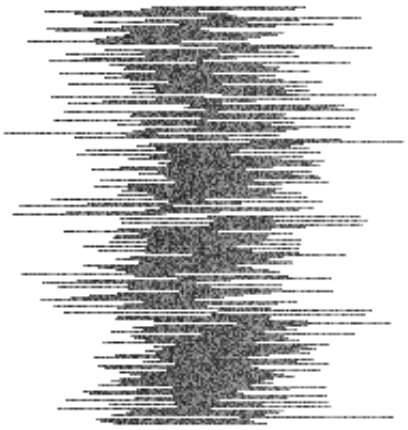
(c) lea.pgm.medmean.png

(d) lea.pgm.mode.png

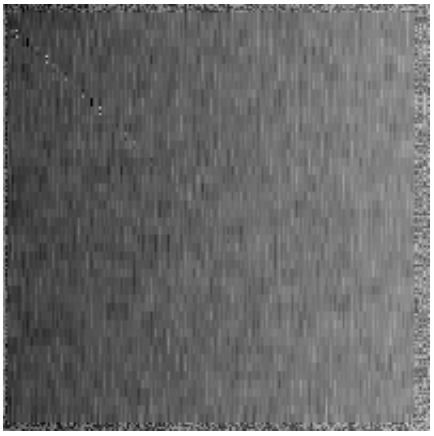
Abbildung D.22:

(a) lea.pgm.wmean.png

(b) lea.png

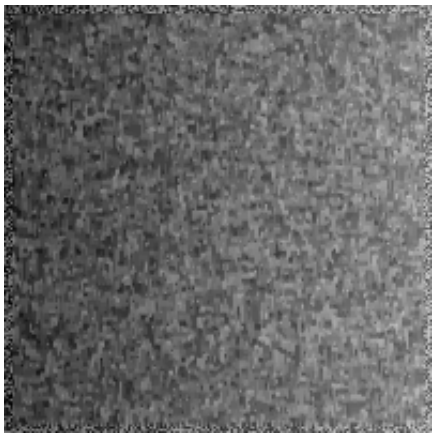


(c) rampn.pgm.autocorr.png

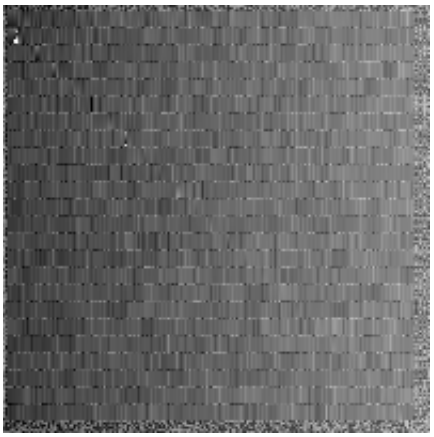


(d) rampn.pgm.linregr2.png

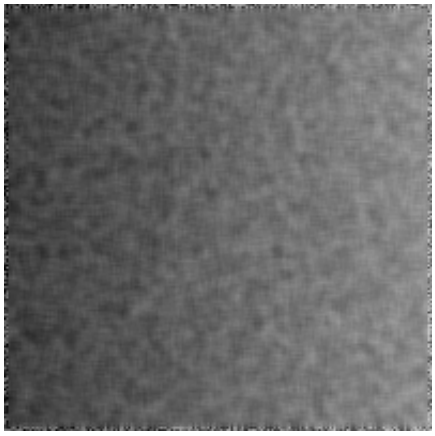
Abbildung D.23:



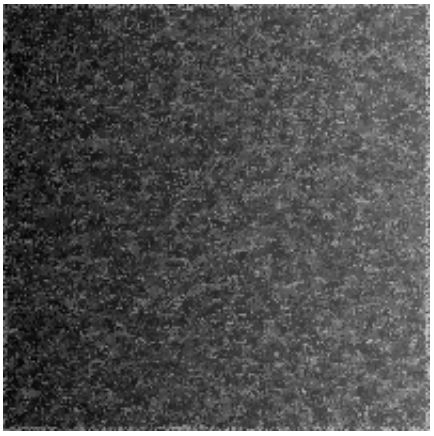
(a) rampn.pgm.linregr.adapt.png



(b) rampn.pgm.linregr.png

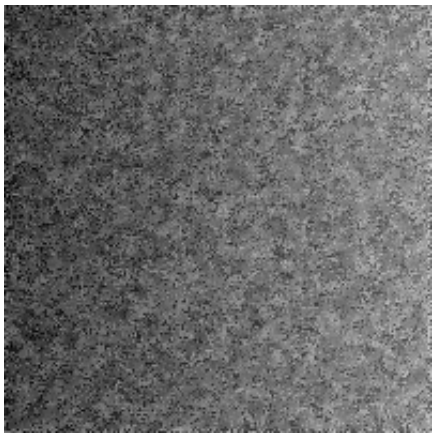


(c) rampn.pgm.mean.png

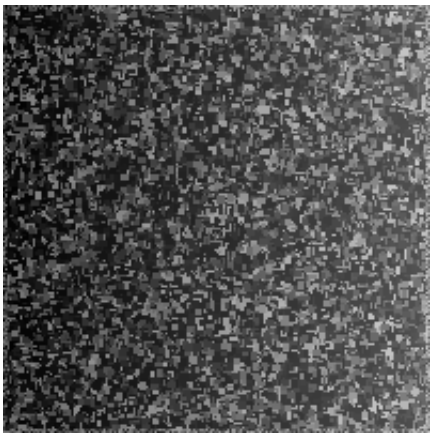


(d) rampn.pgm.median.png

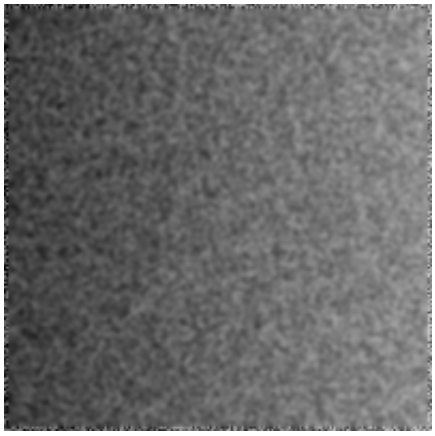
Abbildung D.24:



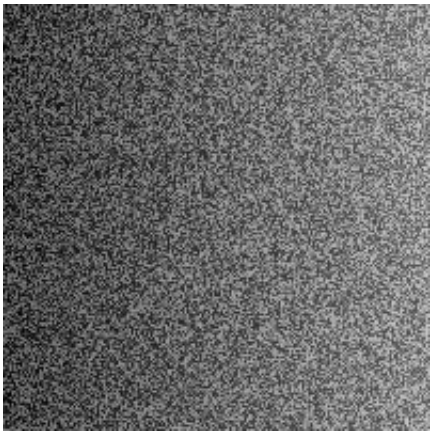
(a) rampn.pgm.medmean.png



(b) rampn.pgm.mode.png

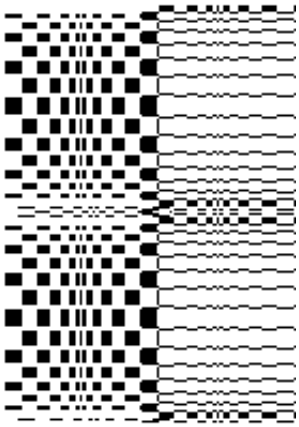


(c) rampn.pgm.wmean.png

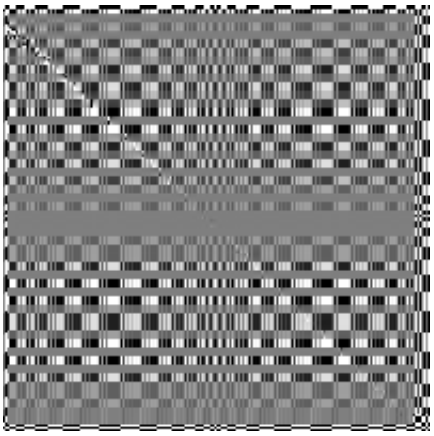


(d) rampn.png

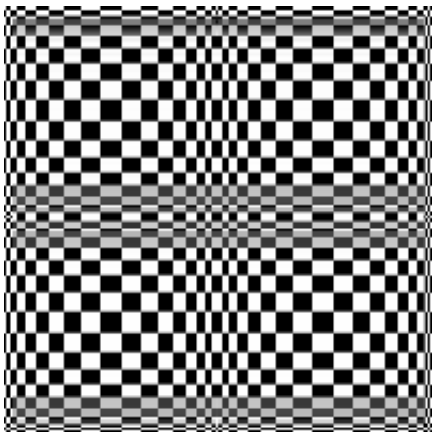
Abbildung D.25:



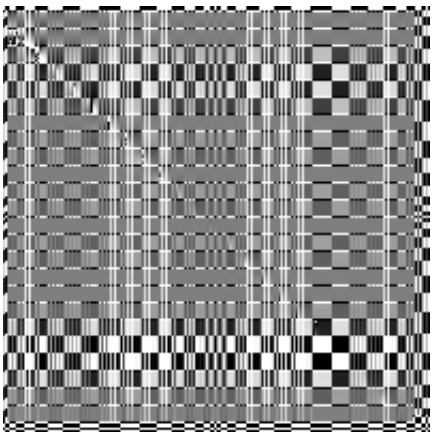
(a) schachbrett.pgm.autocorr.png



(b) schachbrett.pgm.linregr2.png

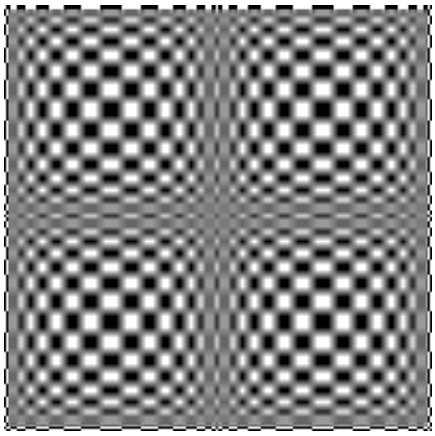


(c) schachbrett.pgm.linregr.adapt.png

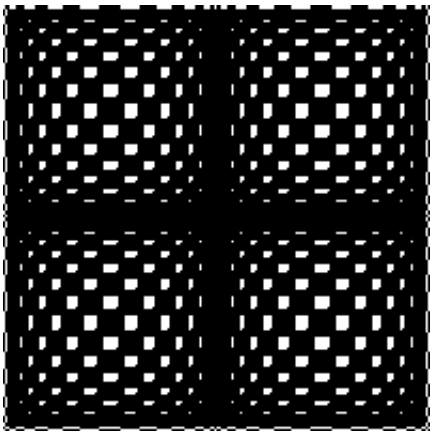


(d) schachbrett.pgm.linregr.png

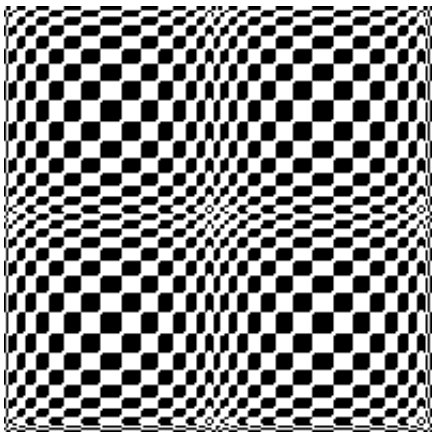
Abbildung D.26:



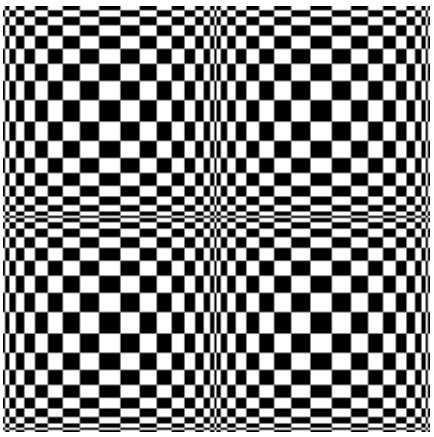
(a) schachbrett.pgm.mean.png



(b) schachbrett.pgm.median.png

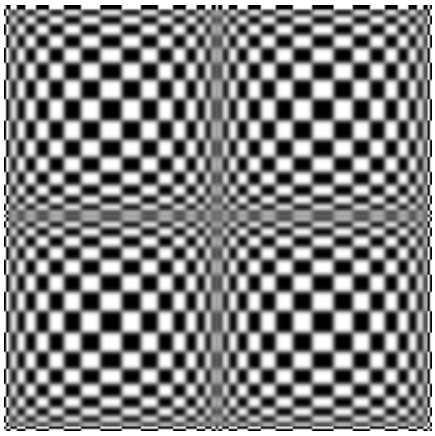


(c) schachbrett.pgm.medmean.png

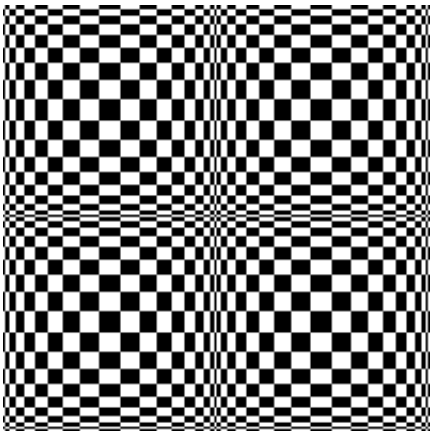


(d) schachbrett.pgm.mode.png

Abbildung D.27:

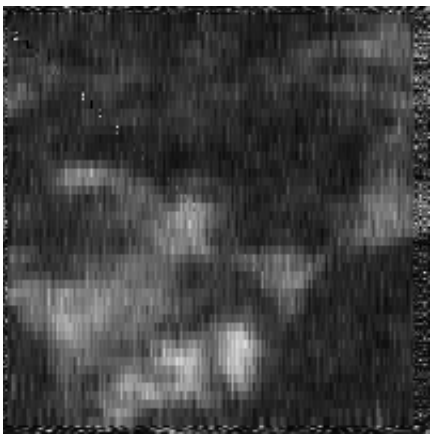


(a) schachbrett.pgm.wmean.png



(b) schachbrett.png

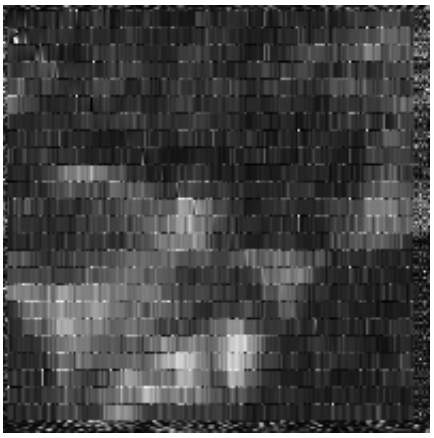
(c) snap-E6.pgm.autocorr.png



(d) snap-E6.pgm.linregr2.png

Abbildung D.28:

(a) snap-E6.pgm.linregr.adapt.png



(b) snap-E6.pgm.linregr.png

(c) snap-E6.pgm.mean.png

(d) snap-E6.pgm.median.png

Abbildung D.29:

(a) snap-E6.pgm.medmean.png

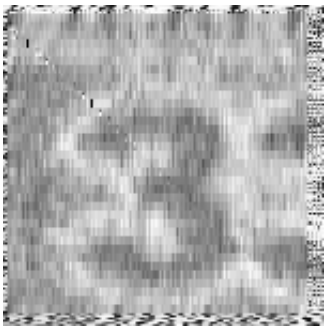
(b) snap-E6.pgm.mode.png

(c) snap-E6.pgm.wmean.png

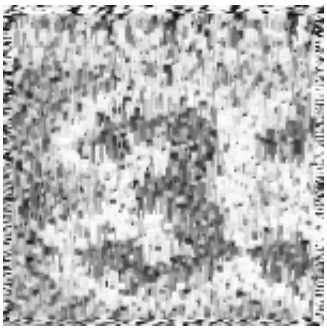
(d) snap-E6.png

Abbildung D.30:

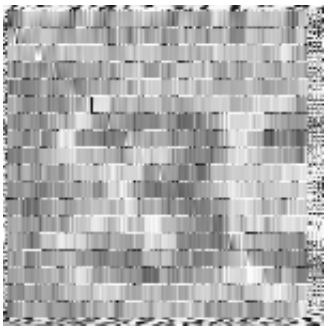
(a) snap-SE10.pgm.autocorr.png



(b) snap-SE10.pgm.linregr2.png

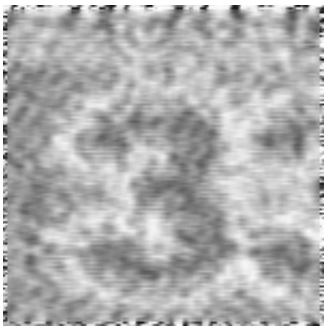


(c) snap-SE10.pgm.linregr.adapt.png

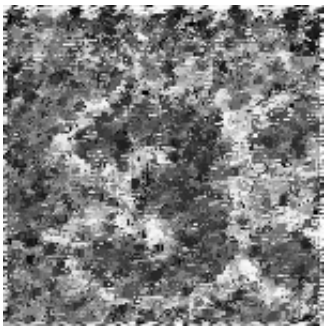


(d) snap-SE10.pgm.linregr.png

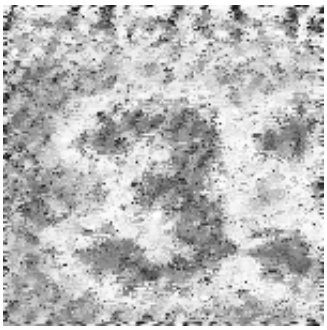
Abbildung D.31:



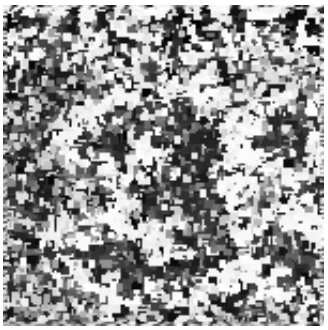
(a) snap-SE10.pgm.mean.png



(b) snap-SE10.pgm.median.png

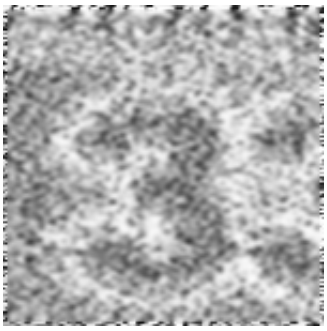


(c) snap-SE10.pgm.medmean.png

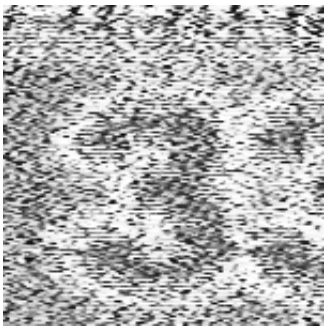


(d) snap-SE10.pgm.mode.png

Abbildung D.32:

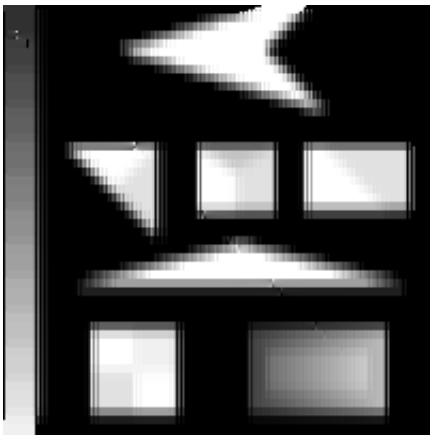


(a) snap-SE10.pgm.wmean.png



(b) snap-SE10.png

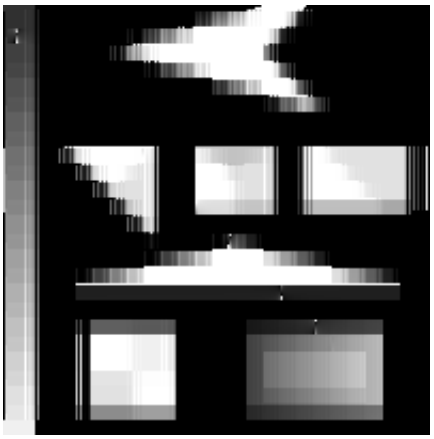
(c) susantest.pgm.autocorr.png



(d) susantest.pgm.linregr2.png

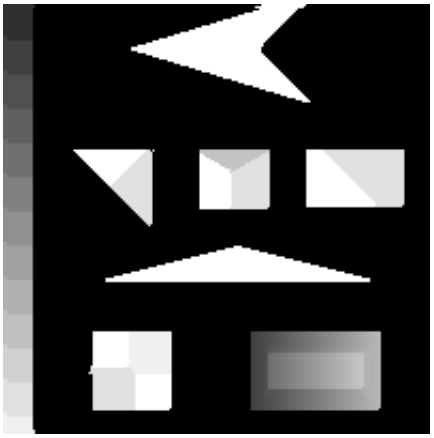
Abbildung D.33:

(a) susan-test.pgm.linregr.adapt.png



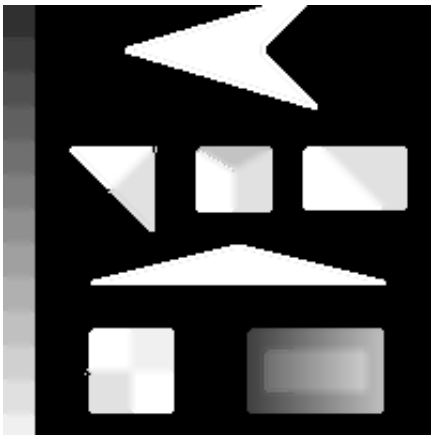
(b) susantest.pgm.linregr.png

(c) susantest.pgm.mean.png

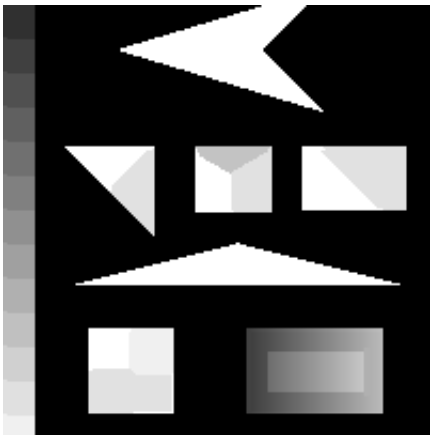


(d) susantest.pgm.median.png

Abbildung D.34:

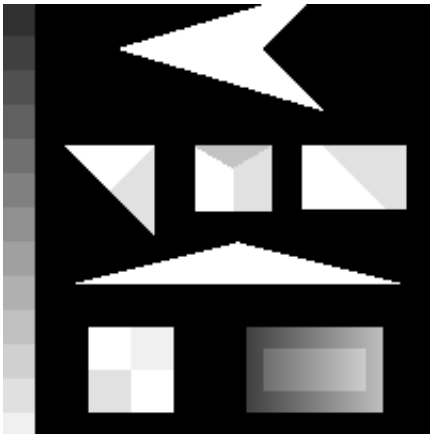


(a) susantest.pgm.medmean.png



(b) susantest.pgm.mode.png

(c) susantest.pgm.wmean.png



(d) susantest.png

Abbildung D.35:

Quellenverzeichnis

[Abbas und Domanski 2000] ABBAS, Jamal K. ; DOMANSKI, Marek: Rejection of Scratches from Video by use of Filters with Prediction Error Processing / Poznan University of Technology, Institute of Electronic and Telecommunications. 2000. – Forschungsbericht. [1] 6.3

[Aitnouh 1999] AITNOUH, Rachid: *Bildkompression mit der Wavelettransformation bei „Schachbrett"-Abtastung*, Fachhochschule Köln, Dissertation, 1999 6.3

[Bartsch 1999] BARTSCH, Hans-Jochen: *Taschenbuch Mathematischer Formeln*. 8. Fachbuchverlag Leipzig, 1999 2.1.3, 5.1.3

[Donoho 1995] DONOHO, David L.: De-noising by soft-thresholding. In: *IEEE Transactions on Information Theory* Bd. 41 IEEE (Veranst.), 1995, S. 613–627 6.1.2

[*Bitterberg, Tilmann and Östreich, Thomas and Entwicklerteam* 2003] *Bitterberg, Tilmann and Östreich, Thomas and Entwicklerteam*: *transcode*. Internet `http://zebra.fh-weingarten.de/~transcode/` and `http://www.theorie.physik.uni-goettingen.de/~ostreich/transcode/`. 2003 6

[*TVTime-Entwicklerteam* 2003] *TVTime-Entwicklerteam*: *TVTime*. Internet `http://tvtime.sourceforge.net/`. 2003 3, 5

[Gilboa u. a. 2003] GILBOA, Guy ; YEHOSHUA, Zeevi Y. ; SOCHEN, Nir: Texture Preserving Variational Denoising Using an Adaptive Fidelity Term / Israel Institute of Technology and University of Tel-Aviv. Internet `http://http://tiger.technion.ac.il/~gilboa/pub/vlsm03.pdf`, 2003. – Forschungsbericht 1.3.2

[Griffin 2000] GRIFFIN, Lewis D.: Mean, median and mode filtering of images / Medical Imaging Science Interdisciplinary Research Group, King's College, London. 2000. – Forschungsbericht 3.4, 3.5

[Haidmayer 2003] HAIDMAYER, Reinhard: *Allgemeine Sinnesphysiologie*. Internet `http://www.kfunigraz.ac.at/phywww/physiotherapie/sinnesphysiologie_auge.pdf`. 2003 1.2

[1] Guter Videofilter um Impulsrauschen zu eliminieren

[Hazard u. a. 1998] HAZARD, Christoph ; HUANG, Jian ; CRAWFIS, Roger: 2D Image Reconstruction and Zooming with Simultaneous Edge Preservation and Blur / Ohio State University;. 1998. – Forschungsbericht. [2] 4.2

[Jähne 1997] JÄHNE, Bernd: *Digitale Bildverarbeitung*. 4. `http://www.berndjaehne.de/englisch/intro_e.html` : Springer Verlag Berlin,Heidelberg, 1997. – [3] 3.3

[Knorr 2003] KNORR, Gerd: *XawTV*. Internet `http://www.bytesex.org/xawtv`. 2003 2, 5.2.2

[Kunz WS 2002/03] KUNZ, Dietmar: Digitale Bildverarbeitung / FH Köln, Institut für Medien- und Phototechnik. WS 2002/03. – Forschungsbericht. [4] 2.2.2

[Pavlidis 1990] PAVLIDIS, Theo: *Algorithmen zur Grafik und Bildverarbeitung*. 5. Verlag Heinz Heise GmbH, 1990 3.1

[Peters II 1995] PETERS II, Richard A.: A New Algorithm for Image Noise Reduction using Mathematical Morphology. In: *IEEE Transactions on Image Processing* 4 (1995), May, Nr. 3, S. 554–568. – [5] 1.3.2, 4.2

[Romer und Pösch 2002] ROMER, Eric ; PÖSCH, Harald: *Entrauschen und Verschärfen digitaler Bilder mit der Wavelettransformation*, Fachhochschule Köln, Dissertation, 2002 6.3

[Schimpf 1996] SCHIMPF, Uwe: *Fourieranalyse mikroskaliger Temperaturfluktuationen der Wasseroberfläche*, Ruprecht-Karls-Universität Heidelberg, Fakultät für Physik und Astronomie, Dissertation, 1996. – `http://klimt.iwr.uni-heidelberg.de/PublicFG/ProjectB/CFT/dipluschimpf/node14.html` 3.3, 4

[Smith und Brady 1995] SMITH, S.M. ; BRADY, J.M.: SUSAN – A new approach to Low Level Image Processing / Oxford University. 1995. – Forschungsbericht. [6] 1.3.4, 4.2, 4.3

[Smith 1997a] SMITH, Steven W.: *The Scientist and Engineer's Guide to Digital Signal Processing*. Kap. 24, S. 397–422. `http://www.dspguide.com/` : California Technical Publishing, 1997. – [7] 3.2

[2] Edge_Mag.PDF
[3] 5. Auflage verfügbar, gute Grundlagen
[4] bv11.doc, Vertiefung Impulsrauschen
[5] sehr gute Ergebnisse, opening/closing
[6] patentiert
[7] Linear Image Processing, Separability of Convolution

[Smith 1997b] SMITH, Steven W.: *The Scientist and Engineer's Guide to Digital Signal Processing*. Kap. 15, S. 277–284. http://www.dspguide.com/ : California Technical Publishing, 1997. – [8] 3.2

[Vaseghi 1996] VASEGHI, Saeed V.: *Advanced Signal Processing and Digital Noise Reduction*. John Wiley & Sons Inc. and B.G. Teubner, 1996 2.2.1, 2.2.2, 2.2.2, 2.2.3

[Winkler 2001] WINKLER, Prf. G.: Probalistische Aspekte der kantenerhaltenden Glättung / GSF – Forschungszentrum für Umwelt und Gesundheit GmbH, IBB – Institut für Biomathematik und Biometrie. 2001. – Forschungsbericht. `http://ibb.gsf.de/preprints/2001/pp01-05.pdf`[9] 3.4, 2

[Zlokolica u. a. 2002] ZLOKOLICA, Vladimir ; PIZURICA, Aleksandra ; PHILIPS, Wilfried: Video denoising using multiple class averaging with Multiresolution / University of Ghent. Internet `http://telin.ugent.be/~vzlokoli`, 2002. – Forschungsbericht 4.2

[Zwisler 1998] ZWISLER, Rainer: *Geschichte der Psychologie des Sehens*. Internet `http://www.zwisler.de/scripts/boring/`. 1998 1.2

[8] Moving Average Filter, Rekursive Implementierung, Konzentration auf Gaussches Rauschen
[9] sehr gute Herleitung und mathematische Analyse Medianfilter ua.

Stichwortverzeichnis

E Selbstständigkeitserklärung

Hiermit erkläre ich, daß ich diese Projektarbeit selbstständig und in eigenständiger Arbeit und nur unter Gebrauch der angegebenen Hilfsmittel und Quellen angefertigt habe.

Leipzig, den 27. Juni 2004