GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

18. November 2019

Bachelorarbeit im Studiengang
„Angewandte Informatik“

SUSAN

Ein Ansatz zur Strukturerkennung in Bildern

Institut für
Numerische und Angewandte Mathematik

Bachelor und Masterarbeiten des Zentrums
für angewandte Informatik an der
Georg-August-Universität Göttingen

Julian Lüken
julian.lueken@stud.uni-goettingen.de

Georg-August-Universität Göttingen
Institut für Informatik

☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de

www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 18. November 2019

Inhaltsverzeichnis

Kapitel 1

Einführung

Ein Kantendetektor ist ein Werkzeug, welches die Stellen der größten Intensitätsunterschiede in Digitalbildern feststellt und markiert. Der Mensch kann durch sein Sehvermögen mit Leichtigkeit Kanten lokalisieren. Um diesen Prozess zu automatisieren, benötigen wir allerdings ein mathematisches Prinzip, welches unabhängig vom spezifischen Bild pixelgenaue Resultate liefert.

Auf der Kantendetektion basieren viele weiterführende Verfahren, zum Beispiel aus den Feldern der kantenbasierten Bildverbesserung und des maschinellen Sehens, in denen die für das menschliche Auge wichtige optische Information über die Position, Länge und Orientierung der Kanten auf den jeweiligen digitalen Bildern in besonderem Maße berücksichtigt wird.

Ein solcher Kantendetektor, nämlich der *SUSAN*-Kantendetektor (*smallest univalue segment assimilating nucleus*) aus [1], wird in dieser Arbeit im Detail vorgestellt. Die Idee dabei ist, für alle Punkte des Bildes festzustellen, welchen Anteil einer den Punkt umgebenden Fläche des Bildes „ähnlich“ zu dem jeweiligen Punkt ist. Dabei meint „Ähnlichkeit“ in diesem Fall, dass der Intensitätsunterschied unter einer vorgegebenen Grenze liegt. Jede einem Punkt (oder Nukleus) zugehörigen Fläche ähnlicher Punkte nennen wir *USAN* (*univalue segment assimilating nucleus*). Unser Ziel ist es, die Punkte (oder Nuklei) der kleinsten USANs zu markieren. Zusätzlich habe ich den *SUSAN*-Kantendetektor in Python 3 implementiert ([10]).

Im ersten Kapitel dieser Arbeit besprechen wir die mathematischen Grundlagen der Bildverarbeitung. Im zweiten Kapitel wird der Kantendetektor neben meiner persönlichen Implementation des Kantendetektors vorgestellt und mit dem Kantendetektor von Canny verglichen. Zuletzt diskutiere ich die Vor- und Nachteile des *SUSAN*-Kantendetektors und meiner Implementation.

Kapitel 2

Mathematische Grundlagen

2.1 Bildverarbeitung

In der digitalen Bildverarbeitung für Graustufenbilder betrachten wir Abbildungen der Form

$$I : [0, B-1]_{\mathbb{Z}} \times [0, H-1]_{\mathbb{Z}} \to [0, 255]_{\mathbb{Z}}.$$

Solche Abbildungen werden im weiteren Verlauf dieser Arbeit schlichtweg als Bilder bezeichnet. H steht für die Höhe und B für die Breite des Bildes. Ferner steht 0 für schwarz und 255 für weiß.

Eines der wichtigen Instrumente der Bildverarbeitung ist die Faltung. Die Faltung ist für Funktionen $f, g : \mathbb{R}^k \to \mathbb{R}$ definiert als

$$(f * g)(x) = \int_{\mathbb{R}^k} f(s) ds.$$

Für $k = 2$ und $(x, y) \in \mathbb{R}^2$ bedeutet diese Definition insbesondere

$$(f * g)(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(s, t)\ g(x - s, y - t)\ ds\ dt$$

Digitale Bilder liegen zwar in der Praxis zweidimensional vor, allerdings nur diskret; passend zur Definition von I. Wir benötigen den Begriff der Faltung nun also für den diskreten Fall und mit endlichen Grenzen. Diese Definition trägt den Namen *Filtermaske*. Eine Filtermaske ist eine Matrix $K \in \mathbb{R}^{n \times m}$, die folgendermaßen Anwendung findet:

$$O(x, y) = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} K(i, j) \cdot I(x - i + a, y - j + b),$$

für $(x, y) \in [0, B-1]_{\mathbb{Z}} \times [0, H-1]_{\mathbb{Z}}$, wobei (a, b) der Nucleus der Filtermaske ist, I das Eingangsbild und O das neu gewonnene Ausgangsbild. Außerhalb des Definitionsbereichs

von I sei für diese Definition $I \equiv 0$. Wir schreiben, wie bei der Faltung

$$O = K * I.$$

Ein Beispiel für eine solche Filtermaske bietet der sogenannte Gaußsche Weichzeichner, der in [3] näher beschrieben wird. Im stetigen Fall würde man ihn anwenden, indem man eine Funktion mit der Funktion der Normalverteilung in zwei Dimensionen faltet. Die Funktion der Normalverteilung lautet für einen Parameter $\sigma > 0$:

$$g_\sigma(x, y) := \frac{1}{2\pi\sigma}\, e^{-\frac{x^2+y^2}{2\sigma^2}}.$$

Bevor wir mit einem diskreten Bild falten können, müssen wir eine endlich große, diskrete Filtermaske aus der Definition von g gewinnen. Dies gelingt dadurch, dass wir die Funktion um jedes Pixel integrieren:

$$\hat{G}_\sigma(x, y) = \int_{y-\frac{1}{2}}^{y+\frac{1}{2}} \int_{x-\frac{1}{2}}^{x+\frac{1}{2}} g_\sigma(s, t)\, dsdt$$

Für eine (große) Anzahl an Stichproben N können wir diese Integration durch Rechtecke approximieren. Man wähle eine Größe $n = m$ der Filtermaske und summiere für $x, y \in [-n, n]_\mathbb{Z}$ jeweils

$$G_\sigma(x, y) = \frac{1}{N} \sum_{i=-\lceil\frac{N-1}{2}\rceil}^{\lfloor\frac{N-1}{2}\rfloor} \sum_{j=-\lceil\frac{N-1}{2}\rceil}^{\lfloor\frac{N-1}{2}\rfloor} g_\sigma(x + \frac{i}{N}, y + \frac{j}{N})$$

für ein passendes σ und ein großes N. Die Größe der Filtermaske bestimmt sich in diesem Fall über das gewählte σ. Eine Faustregel dafür lautet, dass falls $\sqrt{\hat{x}^2 + \hat{y}^2} \geq 3\sigma$ gilt, auch $g(\hat{x}, \hat{y}) \approx 0$ gilt. Ausgehend davon können wir $n = \lfloor 3\sigma \rfloor - 1$ wählen und erhalten somit die Filtermaske

$$G = \frac{1}{273} \begin{pmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{pmatrix}$$

für $\sigma = 1$, $N = 1000$ und $n = 2$. Wenden wir nun unseren Operator G auf folgendes linkes Bild an, so erhalten wir das nachstehende rechte Bild:

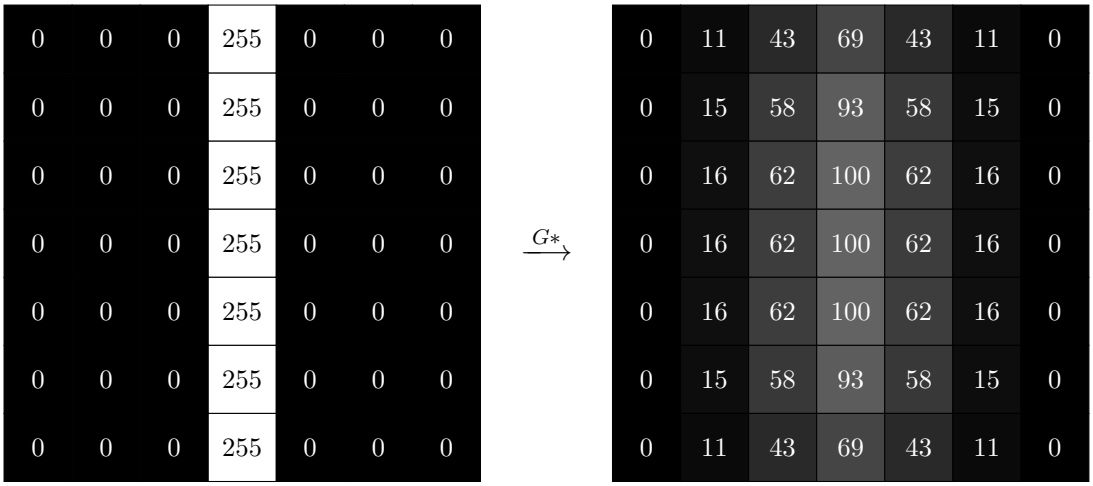

Abbildung 2.1: Die berechnete Gauß-Filtermaske in Aktion.

Dadurch, dass wir das Eingangsbild I außerhalb des Randes 0 setzen, erhalten wir keinerlei Artefakte am Rand. Als Rechenbeispiel benutzen wir jetzt das mittlere Pixel $(x, y) = (3, 3)$ aus Abbildung 2.1.

$$\begin{aligned} O(3,3) &= \sum_{i=0}^{4}\sum_{j=0}^{4} G(i,j) \cdot I(3-i+2, 3-j+2) \\ &= \sum_{i=0}^{4}\sum_{j=0}^{4} G(i,j) \cdot I(5-i, 5-j) \\ &= \frac{1}{273}\left(255 \cdot 7 + 255 \cdot 26 + 255 \cdot 41 + 255 \cdot 26 + 255 \cdot 7\right) \\ &= \frac{255}{273}\left(7 + 26 + 41 + 26 + 7\right) = 99.94 \approx 100 \end{aligned}$$

Kapitel 3

Der SUSAN Kantendetektor

In diesem Kapitel stellen wir den Kantendetektions-Algorithmus nach dem SUSAN Prinzip aus [1] vor, danach erläutern wir einige Details zur Implementation des Algorithmus in Python 3. Im dritten Abschnitt leiten wir her, warum der SUSAN-Kantendetektor funktioniert und im letzten Teil führen wir ein paar numerische Experimente durch.

3.1 Der Algorithmus

Der SUSAN Kantendetektor aus [1] besteht aus drei Großschritten:

Im ersten Schritt, den wir in Abschnitt 3.1.1 besprechen, legen wir eine Maske über jedes Pixel. Mithilfe dieser Maske berechnen wir für jedes Pixel eine Kennzahl A, die im weiteren Verlauf dieser Arbeit „Antwort“ heißt. Die lokalen Maxima von A liegen genau auf den lokalen Extrema der partiellen Ableitungen in horizontaler und vertikaler Richtung unseres Eingangsbildes I, wie wir später noch zeigen.

Der zweite Schritt in Abschnitt 3.1.2 ist die sogenannte Non-Maximum-Suppression, bei der wir mithilfe der Richtung der im ersten Schritt berechneten Antwort die Kanten im Eingangsbild I noch genauer lokalisieren, indem wir nur Züge der Kanten erhalten, die in der Antwort lokal maximal sind.

Falsch positive und falsch negative Ergebnisse sind auf mit Rauschen behafteten Digitalbildern keine Seltenheit. Aus diesem Grund gibt es noch einen dritten Ausdünnungsschritt, in welchem bei Bedarf räumlich isolierte Antworten entfernt werden und Kanten vervollständigt werden. Diesen Schritt behandeln wir näher im Abschnitt 3.1.3.

Anschließend wird im letzten Teil dieses Kapitels, Abschnitt 3.1.4 der Eckendetektor des SUSAN-Prinzips vorgestellt. Durch kleine Änderungen an den ersten zwei Schritten können so, zusätzlich zu Kanten, ebenfalls Ecken gefunden werden.

3.1.1 Das SUSAN-Prinzip

Sei I ein Eingangsbild. Um jedes Pixel im Bild legen wir eine Maske. Für unseren Zweck betrachten wir lediglich die Masken der Größe 3×3 beziehungsweise 7×7, wie sie in Abbildung 3.1 dargstellt sind.

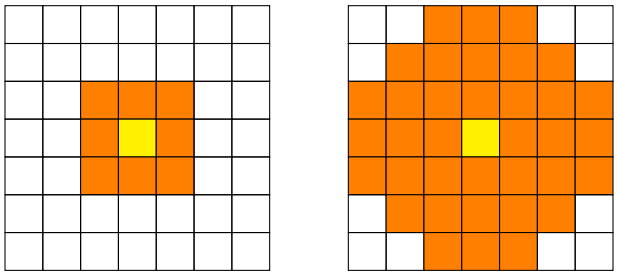

Abbildung 3.1: Die zwei Filtermasken. Links 3×3, rechts 7×7. Dabei ist das gelbe Pixel der Mittelpunkt oder Nucleus der Maske, die orangenen Pixel liegen in der Maske und die weißen Pixel außerhalb der Maske.

Die Maskenträger sind Umgebungen eines Pixels, die einen Kreis approximieren. Der 3×3 Maskenträger approximiert einen Kreis mit einem Radius von 1.4, der 7×7 Maskenträger hingegen approximiert einen kreis mit einem Radius von 3.4.

Prinzipiell finden wir beim ersten Schritt des SUSAN-Verfahrens die Anzahl $n(r)$ der zum Nucleus r, also dem Mittelpunkt der Maske, ähnlichen Pixel für jeden Nucleus r. Mit „ähnlich“ ist in diesem Sinne gemeint, dass die absolute Intensitätsdifferenz $|I(a) - I(r)|$ zwischen den beiden Pixeln a und r unter einer im Vorfeld festgelegten Grenze t liegt, wobei a ein Pixel aus dem Maskenträger ist. Die ähnlichen Pixel bezeichnen wir ferner als USAN (*univalue segment assimilating nucleus*). Die maximale Größe einer USAN ist für die 7×7 Maske 36 und für die 3×3 Maske 8. g ist der sogenannte geometrische Schwellwert und wird gewählt als maximale Größe einer USAN. Um $A(r)$ zu erhalten, subtrahieren wir die Anzahl $n(r)$ der ähnlichen Pixel von g.

Konkret führen wir folgende Rechnung durch. Für jedes Pixel r_0 in I, wobei $I(r_0)$ der Grauwert am Pixel r_0 ist, berechnen wir die Antwort

$$A(r_0) = \max\{0, g - n(r_0)\}.$$

Dabei ist n definiert als

$$n(r_0) = \sum_r c_t(r, r_0),$$

wobei wir über alle Pixel r in der Maske und r_0 selbst summieren und

$$c_t(r, r_0) = \begin{cases} 1 & \text{falls } |I(r) - I(r_0)| \leq t \\ 0 & \text{sonst} \end{cases}$$

eine Vergleichsfunktion für zwei Pixel ist. In einer USAN liegen genau diejenigen Pixel r aus der Maske, für die $c_t(r, r_0) > 0$. Wir verbleiben mit der Antwort A, auf welchem wir schon sehr gut den Effekt der Kantendetektion beobachten können.

255	255	255	255	255	0	255	255	255	255	255
255	255	255	255	255	0	255	255	255	255	255
255	255	255	255	255	0	255	255	255	255	255
255	255	255	255	255	0	255	255	255	255	255
255	255	255	255	255	0	255	255	255	255	255
255	255	255	255	255	0	255	255	255	255	255
255	255	255	255	255	0	255	255	255	255	255
255	255	255	255	255	0	255	255	255	255	255
255	255	255	255	255	0	255	255	255	255	255
255	255	255	255	255	0	255	255	255	255	255
255	255	255	255	255	0	255	255	255	255	255

170	85	85	85	170	255	170	85	85	85	170
85	0	0	0	85	212	85	0	0	0	85
85	0	0	0	85	212	85	0	0	0	85
85	0	0	0	85	212	85	0	0	0	85
85	0	0	0	85	212	85	0	0	0	85
85	0	0	0	85	212	85	0	0	0	85
85	0	0	0	85	212	85	0	0	0	85
85	0	0	0	85	212	85	0	0	0	85
85	0	0	0	85	212	85	0	0	0	85
85	0	0	0	85	212	85	0	0	0	85
170	85	85	85	170	255	170	85	85	85	170

Abbildung 3.2: Das SUSAN Prinzip (links Eingang, rechts Antwort, $t = 1$, 3×3 Maske)

Nehmen wir nun zum Beispiel ein Eingangsbild wie in Abbildung 3.2. Hier wurde der oben genannte Algorithmus angewandt. Sehen wir uns das mittlere Pixel r_0 des Eingangsbilds an und wenden den Algorithmus mit $t = 1$ an:

$$\begin{aligned} n(r_0) &= \sum_r c(r, r_0) \\ &= 0 + 1 + 0 \\ &+ 0 + 1 + 0 \\ &+ 0 + 1 + 0 = 3 \end{aligned}$$

$$A(r_0) = g - n(r_0) = 8 - 3 = 5$$

Um den Effekt hervorzuheben, wird jeder Graustufenwert mit der maximalen Antwort normiert und dann mit 255, dem Maximalwert, der für die Farbe weiß steht, multipliziert. Jeder Graustufenwert wird also mit dem Faktor $\frac{255}{\max A_{i,j}}$ multipliziert. In diesem Fall

erreicht ein Pixel r_1 am Ende der Kante den Maximalwert, da

$$\begin{aligned} n(r_1) &= \sum_r c(r, r_1) \\ &= 0 + 1 + 0 \\ &+ 0 + 1 + 0 = 2 \end{aligned}$$

$$A(r_0) = g - n(r_0) = 8 - 2 = 6$$

Wir multiplizieren also mit $\frac{255}{6}$. Somit erhalten wir an der Stelle r_0 den Wert 212 und an der Stelle r_1 den Wert 255. Zunächst einmal finden wir, dass $A(i, j)$ genau da am größten ist, wo unser Eingangsbild eine Kante hat. Auch an den Rändern des Bilds findet das SUSAN-Prinzip eine Kante. Es besteht lediglich das Problem, dass um die Kante herum eine kleine Antwort dort ist, wo im originalen Bild nur die Nähe zu einer Kante besteht. Wir schließen daraus, dass die Lokalisation der Kanten zwar durchaus funktioniert, aber verbesserungswürdig ist.

255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255
255	255	255	0	0	0	0	0	255	255	255
255	255	255	0	0	0	0	0	255	255	255
255	255	255	0	0	0	0	0	255	255	255
255	255	255	0	0	0	0	0	255	255	255
255	255	255	0	0	0	0	0	255	255	255
255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255	255	255	255

255	127	127	127	127	127	127	127	127	127	255
127	0	0	0	0	0	0	0	0	0	127
127	0	0	63	127	127	127	63	0	0	127
127	0	63	255	127	127	127	255	63	0	127
127	0	127	127	0	0	0	127	127	0	127
127	0	127	127	0	0	0	127	127	0	127
127	0	127	127	0	0	0	127	127	0	127
127	0	63	255	127	127	127	255	63	0	127
127	0	0	63	127	127	127	63	0	0	127
127	0	0	0	0	0	0	0	0	0	127
255	127	127	127	127	127	127	127	127	127	255

Abbildung 3.3: Das SUSAN Prinzip findet Ecken und hebt diese sogar in der Antwort hervor, $t = 1$, 3×3 Maske

Im Beispiel auf Abbildung 3.3 können wir beobachten, dass das SUSAN-Prinzip Ecken besonders hervorhebt. Dieser Effekt spielt eine große Rolle im SUSAN-Eckendetektor, auf den im Abschnitt 3.1.4 dieser Arbeit weiter eingegangen wird.

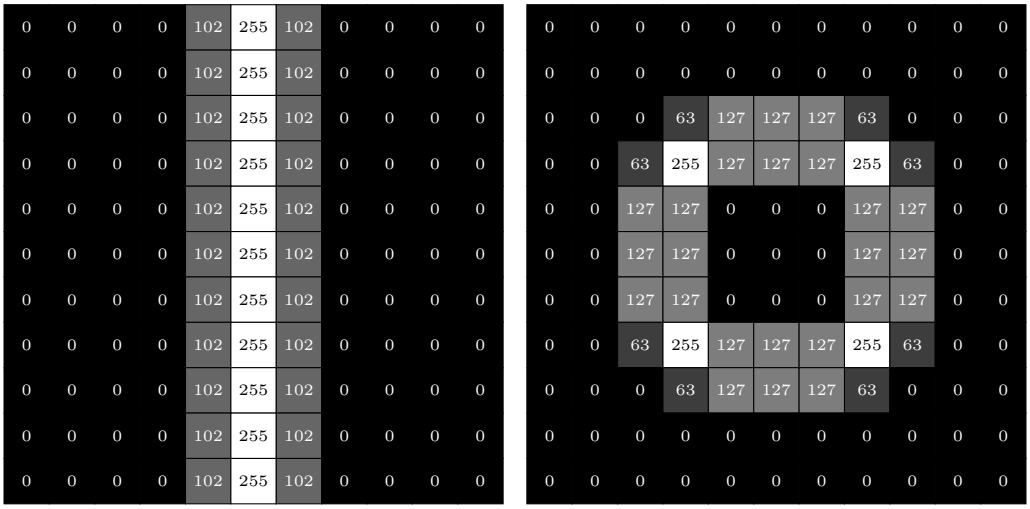

Abbildung 3.4: Randbedingung mit den Vorzeichen σ_i am Beispiel der Abbildungen 3.2 und 3.3

Für r außerhalb des Definitionsbereiches, also $r \notin [0, B-1]_{\mathbb{Z}} \times [0, H-1]_{\mathbb{Z}}$, wurde bisher $c_t(r_0, r) \equiv 0$ gewählt, da das in der Implementation leichter zu realisieren ist. Allerdings führt dies dazu, dass der Bildrand als Kante markiert wird. Diese unangepasste Anwendung lässt sich vermeiden, wenn für $r = r_0 + (d_x, d_y) = (x, y) + (d_x, d_y)$ außerhalb des Defnitionsbereichs des Bildes $c_t(r_0, r_0 + (\sigma_x d_x, \sigma_y d_y))$ gewählt wird, wobei σ_i für $i \in \{x, y\}$ jeweils ein Vorzeichen ist. Es soll jeweils gelten: Falls i nicht im Definitionsbereich ist, so setze $\sigma_i = -1$, andernfalls setze $\sigma_i = 1$. Diese Randbedingungen kommen einer Spiegelung der Pixel am Rand des Bildes gleich. Wie wir in Abbildung 3.4 sehen, wird die Antwort auf dem Rand nicht > 0, falls keine Kante auf dem Rand liegt.

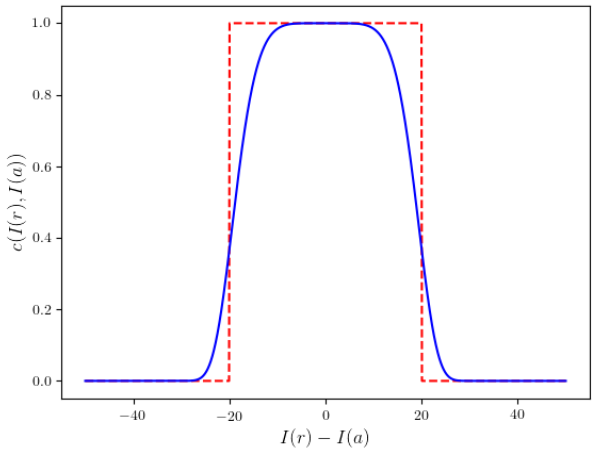

Abbildung 3.5: Die Intensitätsdifferenz aufgetragen gegen die Vergleichsfunktionen. Rot: c_t, blau: $c_{t,exp}$, $t = 10$

In der Praxis verwenden wir statt c_t oft

$$c_{t,exp}(r, r_0) = \exp\Big(- \Big(\frac{I(r) - I(r_0)}{t}\Big)^6\Big).$$

Diese Funktion wird nur annähernd 0 für große Intensitätsdifferenzen, darum wählen wir zusätzlich g als $\frac{3}{4}$ der Maskengröße. Diese Wahl von g und $c_{t,exp}$ hat den Vorteil, dass für Pixel r mit vorher kleinem $A(r)$ nun $A(r) = 0$ gilt. Dies trägt positiv zur Rauschentfernung bei. Die Wirksamkeit dieser Maßnahme ist beispielhaft an Abbildung 3.6 sichtbar.

Abbildung 3.6: Ein Testdurchlauf mit $t = 15$ und der 7×7 Maske. Ganz links ist das Originalbild. In der Mitte ist g die Maskengröße und c_t die Vergleichsfunktion. g ist rechts $\frac{3}{4}$ der Maskengröße und die Vergleichsfunktion ist $c_{t,exp}$.

Da dieser Algorithmus von den lokalen Bedingungen jedes Pixels abhängt, kann dieser Algorithmus nicht durch eine Faltung ausgedrückt werden, wie etwa eine Reihe anderer Filter, darunter der Gaussche Filter aus 2.1, aber auch der Kantendetektor von Canny (siehe 3.3).

3.1.2 Non-Maximum-Suppression

Das oben genannte Prinzip aus Abschnitt 3.1.1 findet Kanten innerhalb von Bildern, aber die Lokalisation der Kanten könnte besser sein, wie wir in der Abbildung 3.2 bereits erkennen konnten. Im Bereich der eigentlichen Kante stellen wir fest, dass die Antwort A ungleich 0 ist. Um die Kante genauer zu lokalisieren, verwenden wir das Prinzip der Non-Maximum-Suppression, bei der nur die maximale Antwort entlang einer Kante erhalten bleibt. Zu diesem Zweck berechnen wir die "Richtungëines jeden Pixels $r_0 = (x_0, y_0)$, für welches $A(x_0, y_0) \neq 0$ gilt, durch eine Fallunterscheidung. Das scheint im ersten Moment fragwürdig, da Pixel an sich keine Richtung haben, sondern die Kanten. Viel mehr ist die Richtung der Kante an der Stelle des jeweiligen Pixel gemeint. Der Einfachheit halber behalten wir uns trotzdem den Begriff der Richtung des Pixels vor. Wir unterscheiden dabei verschiedene Arten von Kanten.

Für die Fallunterscheidung zwischen den Pixeln benötigen wir das sogenannte *center*

of gravity oder Gravitationszentrum einer jeden USAN:

$$\mathrm{COG}(r_0) := \frac{\sum_r r\, c(r, r_0)}{\sum_r c(r, r_0)} = \frac{\sum_r r\, c(r, r_0)}{n(r_0)} \in \mathbb{R}^2.$$

Das $\mathrm{COG}(r_0)$ ist die gewichtete Summe der durch $c \in \{c_t, c_{t,exp}\}$ bedingt ähnlichen Pixel; also in Relation zu r_0 die Richtung, in der die meisten ähnlichen Pixel in der Maske sind. Wir betrachten zum Beispiel die Abbildung 3.7 mit der 3×3 Maske, $c = c_t$ und unter der Annahme, dass $A := (5, 2)$:

$$\begin{aligned}
\mathrm{COG(A)} &= \frac{\sum_r r\, c(r, \mathrm{A})}{\sum_r c(r, \mathrm{A})} \\
&= 0 \begin{bmatrix} 4 \\ 1 \end{bmatrix} + 0 \begin{bmatrix} 5 \\ 1 \end{bmatrix} + 0 \begin{bmatrix} 6 \\ 1 \end{bmatrix} \\
&+ 1 \begin{bmatrix} 4 \\ 2 \end{bmatrix} + 1 \begin{bmatrix} 5 \\ 2 \end{bmatrix} + 1 \begin{bmatrix} 6 \\ 2 \end{bmatrix} \\
&+ 1 \begin{bmatrix} 4 \\ 3 \end{bmatrix} + 1 \begin{bmatrix} 5 \\ 3 \end{bmatrix} + 1 \begin{bmatrix} 6 \\ 3 \end{bmatrix} = \begin{bmatrix} 5 \\ 2.5 \end{bmatrix}.
\end{aligned}$$

In Fällen wie bei dem Pixel A können wir nun die Richtung folgendermaßen bestimmen: Der Vektor zwischen dem Nucleus $\mathrm{A} := (\mathrm{A}_x, \mathrm{A}_y)$ und seinem Gravitationszentrum $\mathrm{COG(A)} := (\mathrm{COG}_x(\mathrm{A}), (\mathrm{COG}_y(\mathrm{A}))$ ist gegeben durch

$$\mathrm{COG(A)} - \mathrm{A} := \begin{bmatrix} \mathrm{COG}_x(\mathrm{A}) - \mathrm{A}_x \\ \mathrm{COG}_y(\mathrm{A}) - \mathrm{A}_y \end{bmatrix}.$$

Der nun gesuchte Vektor soll nun senkrecht auf $\mathrm{COG(A)} - \mathrm{A}$ stehen und damit die Richtung der Kante beschreiben. Es gilt:

$$\begin{bmatrix} \mathrm{COG}_y(\mathrm{A}) - \mathrm{A}_y \\ \mathrm{COG}_x(\mathrm{A}) - \mathrm{A}_x \end{bmatrix} \perp (\mathrm{COG(A)} - \mathrm{A}).$$

Wir beobachten allerdings für das Pixel C in der Abbildung 3.7 und den gleichen Bedingungen wie bei der vorherigen Berechnung, dass $\mathrm{COG(C)} = \mathrm{C}$. Da keine Distanz zwischen $\mathrm{COG(C)}$ und C ist, kann die Richtung der Kante mit dem oben genannten Verfahren nicht ermittelt werden. Bei kleineren Distanzen zwischen einem Pixel r und seinem $\mathrm{COG}(r)$ wird das Verfahren zunehmend ungenau. Stattdessen nutzen wir die Stichprobenvarianz in x- und y-Richtung folgendermaßen: Summiere für alle Pixel in der Maske $r = (x, y)$ und den

Nucleus $r_0 = (x_0, y_0)$ für $c \in \{c_t, c_{t,exp}\}$

$$Var_x(r_0) := \frac{1}{N} \sum_{r=(x,y)} (x - x_0)^2 \, c(r, r_0)$$
$$Var_y(r_0) := \frac{1}{N} \sum_{r=(x,y)} (y - y_0)^2 \, c(r, r_0),$$

wobei N die Anzahl der Pixel ist, über die iteriert wird. Die Varianz ist ein Maß dafür, wie weit die Pixel, die in der USAN liegen, durchschnittlich vom "Mittelwert"(in diesem Fall dem Nucleus r_0) in x- beziehungsweise y-Richtung abweichen. Im Falle des Pixels $\mathrm{C} = (3, 5)$ mit der 3×3 Maske und der $c = c_t$ sind die Varianzen gegeben durch

$$\begin{aligned} N\, Var_x(\mathrm{C}) = {} & (2-3)^2 \cdot 0 \; + \; (3-3)^2 \cdot 1 \; + \; (4-3)^2 \cdot 0 \\ & + (2-3)^2 \cdot 0 \; + \; (3-3)^2 \cdot 1 \; + \; (4-3)^2 \cdot 0 \\ & + (2-3)^2 \cdot 0 \; + \; (3-3)^2 \cdot 1 \; + \; (4-3)^2 \cdot 0 = 0 \end{aligned}$$

$$\begin{aligned} N\, Var_y(\mathrm{C}) = {} & (4-5)^2 \cdot 0 \; + \; (4-5)^2 \cdot 1 \; + \; (4-5)^2 \cdot 0 \\ & + (5-5)^2 \cdot 0 \; + \; (5-5)^2 \cdot 1 \; + \; (5-5)^2 \cdot 0 \\ & + (6-5)^2 \cdot 0 \; + \; (6-5)^2 \cdot 1 \; + \; (6-5)^2 \cdot 0 = 2 \end{aligned}$$

Wie im ersten Fall können wir nun die Richtung des Vektors $\frac{Var_x(\mathrm{C})}{Var_y(\mathrm{C})}$ bestimmen. Der normierende Faktor N der Stichprobenvarianz fällt dabei weg. Wir stellen uns vor, eine diagonale Kante liegt vor. Dann sehen wir, dass $Var_x(r_0) \approx Var_y(r_0)$. Ob eine „positiv diagonale“ oder "negativ diagonale"Kante vorliegt (also eine von links unten nach rechts oben oder eine von rechts unten nach links oben), entscheidet nun ein Vorzeichen, welches wir aufgrund von $Var_x > 0$ und $Var_y > 0$ nicht haben. Das Vorzeichen können wir allerdings mithilfe folgender Gleichung berechnen:

$$Var_{x,y}(r_0) = \Big(\sum_r (x - x_0)\,(y - y_0)\, c_t(r, r_0) \Big).$$

Diese „gemischte Varianz“ trägt die Information über die Richtung der diagonalen Kante im Vorzeichen. Man betrachte zum Beispiel die beiden diagonalen Kanten D und E unter

den gleichen Voraussetzungen wie C:

$$
\begin{aligned}
N\,Var_x(\mathrm{D}) &= N\,Var_x(\mathrm{E}) = 2 \\
N\,Var_y(\mathrm{D}) &= N\,Var_y(\mathrm{E}) = 2 \\
\mathrm{sgn}(Var_{x,y}(\mathrm{D})) &= \mathrm{sgn}\Big((-1)\cdot(-1) + 0\cdot 0 + 1\cdot 1\Big) = 1 \\
\mathrm{sgn}(Var_{x,y}(\mathrm{E})) &= \mathrm{sgn}\Big((-1)\cdot 1 + 0\cdot 0 + 1\cdot(-1)\Big) = -1
\end{aligned}
$$

Für jeden Nucleus gilt: Falls das COG nah am Nucleus liegt und viele Pixel in der USAN liegen, so sprechen wir vom Inter-Pixel-Fall. Der Inter-Pixel-Fall liegt nämlich vor, wenn eine Kante zwischen zwei Pixeln liegt hier benutzen wir die Berechnung der Richtung über das Gravitationszentrum. Der Intra-Pixel-Fall hingegen liegt vor, wenn ein Pixel genau auf der Kante liegt. Hier benutzen wir die Stichprobenvarianz. In der Abbildung 3.7 stehen die grün markierten Pixel jeweils für Pixel, die in der USAN liegen. Die roten Pixel liegen außerhalb der USAN. Die Mitte ist der Nucleus der Maske.

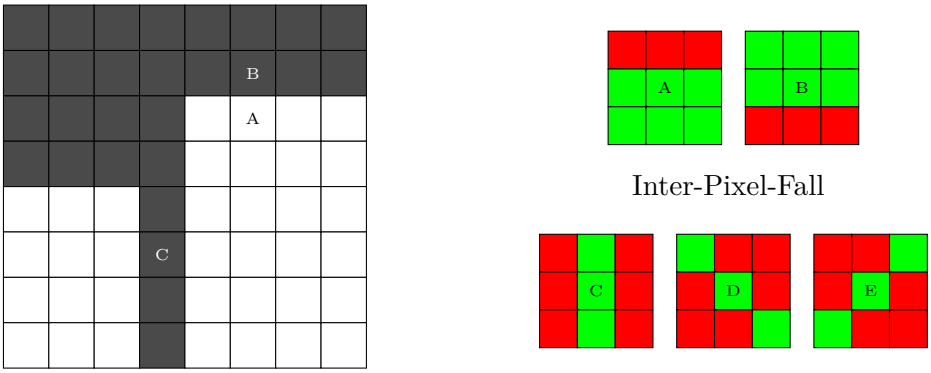

Abbildung 3.7: Inter-Pixel-Fall und Intra-Pixel-Fall für die 3×3 Maske: Die drei rechts abgebildeten USAN wurden durch Wahl des Nucleus an den mit A, B und C markierten Stellen im Eingangsbild ausgewertet. Die grün markierten Pixel stehen für Teile der USAN. Die Nuclei D und E gelten nur der weiteren Illustration von Möglichkeiten und sind nicht im Eingangsbild vorzufinden.

Zusammenfassend rechnen wir für jedes Pixel r_0 im Bild folgendermaßen die Kantenrichtung $D(r_0)$:

1. **Inter-Pixel:**

   Falls die Größe der USAN größer ist als der Maskendurchmesser und die Distanz zwischen $\mathrm{COG}(r_0)$ und $r_0 = (x_0, y_0)$ größer als 1 Pixel ist, so ist die Richtung $D(r_0)$

gegeben durch

$$D(r_0) = \begin{cases} \arctan\left(\frac{x_0 - \mathrm{COG}_x(r_0)}{y_0 - \mathrm{COG}_y(r_0)}\right) & \text{falls } \mathrm{COG}_y(r_0) \neq y_0 \\ \frac{\pi}{2} & \text{sonst} \end{cases},$$

2. **Intra-Pixel:**

   Andernfalls müssen wir die Stichprobenvarianzen der USAN folgendermaßen berechnen: Wir summieren über alle Pixel $r = (x, y)$ in der Maske und $r_0 = (x_0, y_0)$

$$N\,Var_x(r_0) := \sum_r (x - x_0)^2\, c_t(r, r_0)$$
$$N\,Var_y(r_0) := \sum_r (y - y_0)^2\, c_t(r, r_0)$$
$$\sigma := -\mathrm{sgn}\left(\sum_r (x - x_0)\,(y - y_0)\, c_t(r, r_0)\right)$$

   Dabei ergibt sich die Kantenrichtung als

$$D(r_0) = \begin{cases} \sigma \arctan \frac{Var_y(r_0)}{Var_x(r_0)} & \text{falls } Var_x(r_0) \neq 0 \text{ und } \sigma \neq 0 \\ \frac{\pi}{2} & \text{falls } \sigma = 0 \text{ und } Var_x(r_0) < Var_y(r_0) \\ & \text{oder falls } Var_x(r_0) = 0 \\ 0 & \text{sonst} \end{cases}.$$

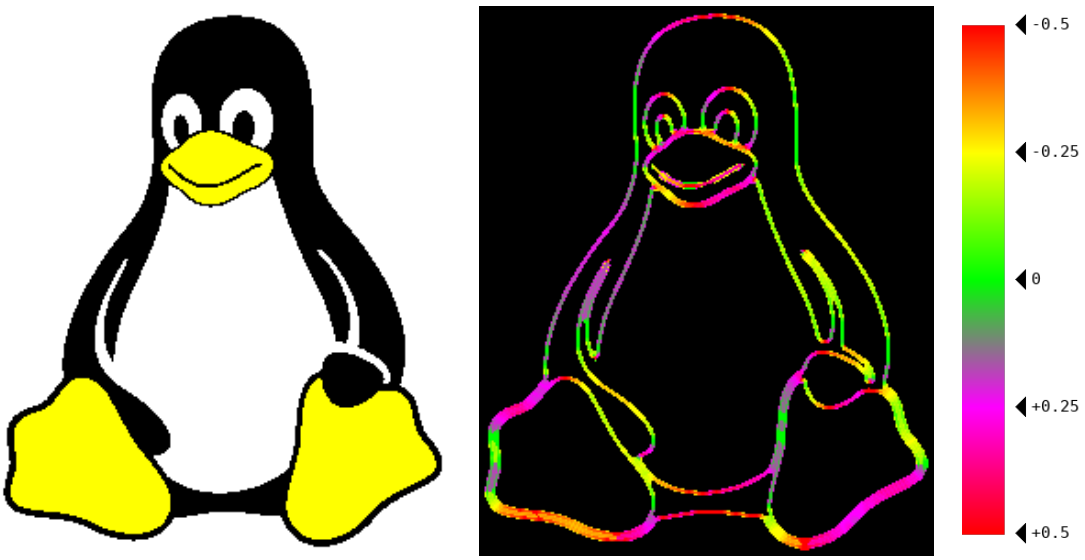

Abbildung 3.8: Ein Testdurchlauf für das Berechnen der Richtungen mit $t = 15$. Links ist das Eingangsbild. Rechts ist die Kantenrichtung für jedes Pixel abgebildet. Die Zahlenwerte können an der Skala abgelesen werden und sind als Vielfache von π angegeben.

Für Pixel außerhalb des Definitionsbereiches $r \notin [0, B-1]_{\mathbb{Z}} \times [0, H-1]_{\mathbb{Z}}$ legen wir für diese Berechnungen die gleichen Randbedingungen fest, die wir auch schon für den Hauptteil des Algorithmus in Sektion 3.1.1 festgelegt haben.

Die Berechnungen für die jeweiligen Kantenrichtungen an jedem Pixel werden in der Implementation gleichzeitig mit den Berechnungen für das SUSAN-Prinzip durchgeführt. Auf diese Weise müssen c und n an jeder Stelle nur einmal berechnet werden.

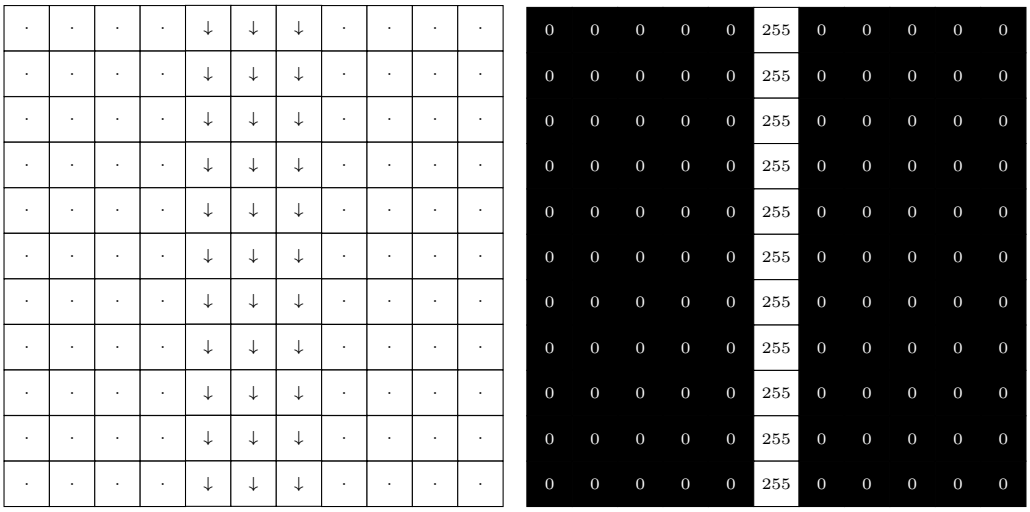

Abbildung 3.9: Eingangsbild wie in Abbildung 3.4. Links sind die Kantenrichtungen abgebildet und klassifiziert wie wir noch in Sektion 3.2.2 besprechen werden. Rechts wurde das Bild durch die Non-Maximum-Suppression wurde die Kante pixelgenau lokalisiert.

Die Richtung muss nur für diejenigen Pixel (i, j) zu bestimmt werden, für die $A(i, j) > 0$ gilt. Ist die Richtung der Kante bestimmt, so können wir die lokalen Maxima von A entlang der Richtung, die senkrecht zur Kantenrichtung steht, erhalten. Alles, was kein lokales Maximum entlang dieser Richtung ist, wird verworfen. Wir erhalten so ein neues Bild. In Sektion 3.2.2 wird darauf eingegangen, wie genau dieser Prozess implementiert wurde.

Die dort vorgestellte Prozedur kategorisiert jedes Pixel nach der generellen Richtung jeder Kante mit $\frac{\pi}{4}$ Genauigkeit. Senkrecht zu dieser Richtung wird dann nur das Maximum der direkten Nachbarschaft erhalten, der Rest wird aus dem Kantenbild entfernt (also $= 0$ gesetzt).

Die Non-Maximum-Suppression unterdrückt in unserem Beispiel tatsächlich alle diejenigen Pixel, die keine lokalen Maxima in der Richtung senkrecht zur Kante sind. Auch für größere Bilder erzielt die Non-Maximum-Suppression den erwünschten Effekt, siehe Abbildung 3.10.

Abbildung 3.10: Wie Abbildung 3.6. Links das Bild mit den Kantenrichtungen (Skala wie in Abbildung 3.8). In der Mitte das Bild nach dem prinzipiellen SUSAN-Schritt. Das Bild rechts ist nach der Non-Maximum-Suppression.

3.1.3 Ausdünnen

Da viele digitale Bilder eingangs mit Rauschen behaftet sind, ist es manchmal hilfreich, einzelne Antworten zu entfernen und dafür andere hinzuzufügen. Zu diesem Zwecke empfiehlt [1], dass man einige der Kanten ausdünnt. Dieser Vorgang wird genauer in [2] beschrieben. Es wird erneut über das ganze Bild iteriert. Jedes Pixel (i, j) mit einer Antwort $A(i, j) > 0$ wird auf die Anzahl seiner direkten Nachbarn mit $A(x, y) > 0$ überprüft. Als direkte Nachbarschaft werden die acht nächsten Pixel bezeichnet, siehe dazu die 3×3 Maske in Abbildung 3.1. Angenommen, das Pixel hat...

- **0 Nachbarn:** Entferne die Antwort des Pixels.
- **1 Nachbar:** Überprüfe, ob in einer Reichweite von 3 Pixeln eine Linie mit der gleichen Richtung existiert. Falls ja, verbinde die Pixel miteinander.
- **2 Nachbarn:** Falls das Pixel benachbart zu einer diagonalen Linie ist, entferne es. Falls das Pixel außerhalb einer sonst horizontalen oder vertikalen Linie liegt, verschiebe die Antwort des Pixels in die Lücke der vertikalen oder horizontalen Linie.
- **3 Nachbarn:** Falls die drei Nachbarn in einer Linie liegen, entferne die Antwort des Pixels.

Im Testbild der Abbildung 3.11 erkennt man die Funktionsweise in allen vier oben genannten Fällen.

Abbildung 3.12: BLABLUBBI thinout neighbors??

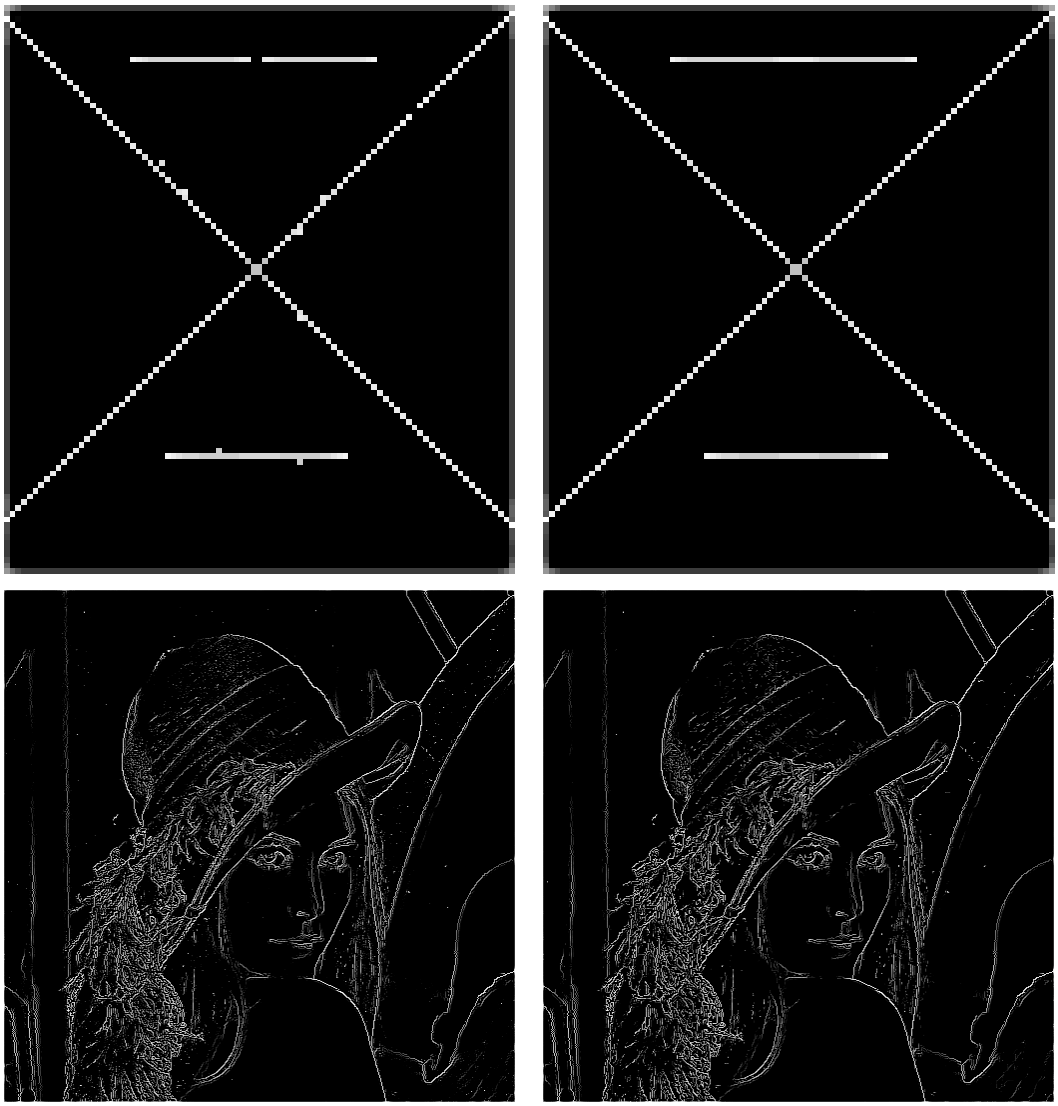

Abbildung 3.11: Links: Kantenbilder nach der Non-Maximum-Suppression. Rechts: Kantenbilder nach dem Ausdünnen.

Laut der zitierten Quelle [2] soll eigentlich auch für die Fälle mit mehr als 3 Nachbarn gesorgt werden. Allerdings ist eine Vorhersage über Nachbarschaften mit 4 oder mehr Nachbarn relativ schwierig. In Abbildung 3.12 sind ein paar Beispielhafte Fälle abgebildet.

3.1.4 Der SUSAN-Eckendetektor

In [1] wird zusätzlich ein Verfahren beschrieben, mit welchem sich durch das SUSAN-Prinzip Ecken finden lassen. Dieser Eckendetektor basiert darauf, dass der SUSAN Kantendetektor an Ecken stärkere Antworten liefert als an Kanten (siehe auch Abbildung 3.3).

Wir erinnern uns an den Algorithmus aus Sektion 3.1.1:

$$A(r_0) = \max\{0, g - n(r_0)\},\ n(r_0) = \sum_r c_t(r, r_0),\ c_t(r, r_0) = \begin{cases} 1 & \text{falls } |I(r) - I(r_0)| \leq t \\ 0 & \text{sonst} \end{cases}.$$

Um nur vergleichsweise starke Antworten zu erhalten, benutzen wir statt dem bisherigen geometrischen Schwellwert g einen niedrigeren Wert. Im Beispiel in Abbildung 3.13 wurde ein Wert von $\frac{1}{4}$ der Maskengröße verwendet.

Danach finden sich immer noch viele falsch positive Antworten. Um diese zu vermeiden, überprüft man, ob die jeweiligen Kandidaten mit einer Kante zusammenhängen. Dies gelingt durch das Erzwingen folgender Regeln für jede Ecke:

1. Die Entfernung zwischen Gravitationszentrum und Nucleus muss groß genug sein.
2. Jedes Pixel in der Maske, welches sich auf gerader Strecke zwischen Nucleus und Gravitationszentrum befindet, muss in der USAN des Nucleus enthalten sein.
3. Alle Antworten $A(i, j)$, die kein lokales Maximum in einer 5×5 Maske sind, sollen entfernt werden.

Die Entfernung zwischen Gravitationszentrum und Nucleus sichert ab, dass die Richtung der Kante sich an der Ecke ändert. Punkt zwei prüft den räumlichen Zusammenhang von Ecken und Kanten. Der dritte Punkt ist vom Prinzip gleich der Non-Maximum-Suppression.

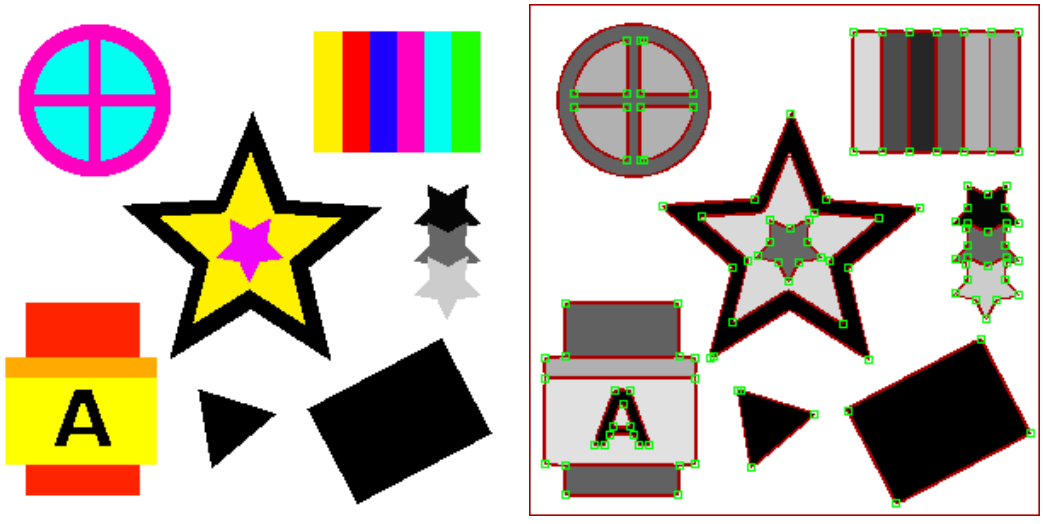

Abbildung 3.13: Links das Eingangsbild, rechts sind Kanten und Ecken markiert. Die Ecken sind grün umrahmt wohingegen die Kanten rot übergezeichnet sind. Für g wurde hier $\frac{3}{4}$ der Maskengröße verwendet. Der Mindestabstand zwischen Nucleus und Gravitationszentrum einer jeden USAN ist hier $\sqrt{(2)}$.

3.2 Implementation

Meine persönliche Implementation ist in Python 3 erfolgt. Die Software ist abhängig von den folgenden Python 3 Paketen:

- `pillow`, eine Bibiliothek für Bildverarbeitung [4],
- `numpy`, eine Bibliothek für wissenschaftliches Rechnen [5],
- `multiprocessing` eine Bibiliothek für prozessbasierte Parallelisierung [6].

Die Software ist verfügbar unter [10]. In der Klasse `Susan.py` befindet sich der vollständige Kantendetektor. Er lässt sich per `import Susan` nutzen. Dazu erzeugt man ein Objekt der `Susan`-Klasse und übergibt eine Bilddatei. Danach ruft man auf dem Objekt die Funktion `detect_edges_mp` mit dem Parameter t, dem Grenzwert aus Abschnitt 3.1.1, auf.

```
1 import Susan
2
3 t = 15
4 S = Susan("lena.png")
5 S.detect_edges_mp(t)
```

Abbildung 3.14: Nutzen der Python 3 Applikation

Es können weiterhin folgende Parameter übergeben werden:

Name	**Beschreibung**	**Datentyp**	**Standardwert**
`t`	Grenzwert für c_t	Ganzzahl, $t \in [1, 255]_{\mathbb{N}}$	-
`filename`	Ausgabepfad	Zeichenkette	`"out.png"`
`nms`	Non-Maximum-Suppression	Boolesch	`True`
`heatmap`	Ausgabe der Richtungen	Boolesch	`False`
`geometric`	Schwache Antworten unterdrücken	Boolesch	`True`
`corners`	Ecken finden	Boolesch	`False`
`overlay`	Überlagerungsbild erzeugen	Boolesch	`True`

Beim Paramater `heatmap` werden die Richtungen als Bild ausgegeben, dabei korrespondieren die Richtungen zu Farben, genau so wie wir es schon in den Abbildungen zu Abschnitt 3.1.2 sehen konnten. Falls `geometric` auf `True` gesetzt ist, so wird g als $\frac{3}{4}$ der Maskengröße gewählt. Andernfalls ist g gleich der Maskengröße, wie im Abschnitt 3.1.1. Die Option `overlay` erzeugt ein Bild, auf dem Kanten in rot nachgezeichnet und Ecken von grünen Kästen umrahmt werden, wie in Abbildung 3.13.

3.2.1 Vergleichsfunktion

Zu Gunsten der Effizienz habe ich für die Vergleichsfunktion eine Lookup-Tabelle implementiert, wie es auch in [1] nahgelegt wurde. Beim Initialisieren des `Susan`-Objekts wird ein Feld erzeugt. Das Ziel ist es, alle möglichen Werte, die c_t (siehe 3.1.1) annehmen kann, zu speichern. In unserem Fall ist c_t folgendermaßen definiert:

$$c_t(a,b) := \exp\left(-\left(\frac{I(a)-I(b)}{t}\right)^6\right).$$

Ein Pixel in einem 8-bit Graustufenbild kann $2^8 = 256$ mögliche Intensitäten annehmen, demnach braucht das Feld für die Differenz zweier solcher Graustufenintensitäten 512 Plätze. Regulär werden Felder entweder ab 0 oder 1 indiziert. In Python werden Felder ab 0 indiziert, allerdings bietet Python den Vorteil, dass man mit Index $-i$ das i-te Element von hinten abrufen kann. In der Implementation ist diese Lookup-Tabelle dank dieser speziellen Art der Indizierung einfach und elegant: Der Index $a-b$ des Feldes steht für $c_t(a,b)$.

3.2.2 Non-Maximum-Suppression

Die Non-Maximum-Suppression ist ein Vorgang, bei der jede Kante genauer lokalisiert wird. Entlang einer Kante soll immer nur die maximale Antwort erhalten bleiben. Um die Non-Maximum-Suppression durchzuführen, wird zunächst für jedes Pixel (i,j) mit $A(i,j) > 0$ die Kantenrichtung $D(i,j)$ bestimmt. Die möglichen Kantenrichtungen werden dann für einen Zwischenschritt kategorisiert. Die Kategorien sind *negativ diagonal*, *vertikal*, *positiv diagonal* und *horizontal*. Die Kategorie bestimmt, welche zwei adjazenten Pixel C für die Non-Maximum-Suppression interessant sind.

Bedingung	**Kategorie**	**Adjazente C**
$D(i,j) \leq -\frac{3}{8}\pi$	negativ diagonal	$\{(i+1,j-1), \quad (i-1,j+1)\}$
$D(i,j) > -\frac{3}{8}\pi, \quad D(i,j) \leq -\frac{1}{8}\pi$	horizontal	$\{(i-1,j), \quad (i+1,j)\}$
$D(i,j) > -\frac{1}{8}\pi, \quad D(i,j) \leq \frac{1}{8}\pi$	positiv diagonal	$\{(i+1,j+1), \quad (i-1,j-1)\}$
$D(i,j) > -\frac{3}{8}\pi$	vertikal	$\{(i,j-1), \quad (i,j+1)\}$

Für jedes Pixel (i,j) wird nun überprüft, ob $(i,j) = \max(C \cup \{(i,j)\})$ gilt. Gilt es nicht, so wird $A(i,j)$ unterdrückt.

3.2.3 Parallelisierung

Das SUSAN-Prinzip sieht vor, die Antwort $A(i,j)$ und die Kantenrichtung $D(i,j)$ immer nur mithilfe von Pixeln aus der Maske auf (i,j) zu berechnen. An dieser Stelle kann die Berechnung von A und D parallelisiert werden. In meiner Implementation liefert das Paket

`multiprocessing` die nötigen Werkzeuge, um die vorhandenen Ressourcen zusammenzuschließen und die Berechnung von A und D parallel abzuarbeiten.

Der Einfachheit halber habe ich das Bild nur der Höhe nach partitioniert. Eine Partitionierung $\mathcal{S}$ ist eine Menge von nichtnegativen ganzen Zahlen $\{S_1, S_2, ..., S_n, S_{n+1}\}$. Für alle $i \in \{1, 2, ..., n\}$ arbeitet der Job J_i die Partition $(S_i, S_{i+1}]_{\mathbb{Z}}$ ab.

Unter der stark vereinfachten Annahme, dass alle Kerne gleich viele Fließkommazahloperationen pro Zeiteinheit abarbeiten können und gleichgroße Partitionen etwa gleichviel Rechenzeit benötigen, gelte folgende Aussage: Eine Partitionierung $\mathcal{S}$ für die gilt $\#\mathcal{S} = n + 1$ ist genau dann optimal, wenn für alle Partitionen gilt

$$|\#(S_i, S_{i+1}]_{\mathbb{Z}} - \#(S_j, S_{j+1}]_{\mathbb{Z}}| \leq 1, \quad \forall i \neq j, \quad i, j \in \{1, 2, ..., n\}.$$

Angenommen der Computer, auf dem die Software läuft, besitzt n Kerne, kann also n Jobs $J_1, J_2, ..., J_n$ gleichzeitig abarbeiten, und ein Bild der Höhe H wird eingegeben. Sei $k := H \bmod n$ und $z := \left\lfloor \frac{H}{n} \right\rfloor$. Dann ist die optimale Partitionierung $\mathcal{S}$ nach der Höhe gegeben durch:

$$\begin{aligned}
S_1 &= 0 \\
S_2 &= S_1 + z + 1 \\
S_3 &= S_2 + z + 1 \\
&\vdots \\
S_{k-1} &= S_{k-2} + z + 1 \\
S_k &= S_{k-1} + z + 1 \\
S_{k+1} &= S_k + z \\
&\vdots \\
S_n &= S_{n-1} + z \\
S_{n+1} &= S_n + z = H
\end{aligned}$$

Die Jobs $J_1, J_2, ..., J_n$ arbeiten das komplette Bild ab, denn es gilt

$$\bigcup_{i=1}^{n} (S_i, S_{i+1}]_{\mathbb{Z}} = (0, H]_{\mathbb{Z}} = [1, H]_{\mathbb{Z}}$$

3.3 Vergleich mit dem Kantendetektor von Canny

In diesem Abschnitt wird der Kantendetektor von Canny (aus [9]) kurz vorgestellt und anschließend mit dem SUSAN-Kantendetektor verglichen. Der Kantendetektor von Canny basiert auf dem Prinzip der Ableitung: Ziel ist es, die Stellen der größten Änderung zu markieren. In der stetigen Analysis sind diese Stellen die Nullstellen der zweiten Ableitung. Da wir allerdings keine stetige Funktion vorliegen haben, sondern lediglich ein digitales Bild, müssen wir eine Approximation finden.

Zunächst aber wird das Bild geglättet, meist durch einen Gausschen Weichzeichner, wie er bereits in Kapitel 2.1 besprochen wurde.

Unter den Begriff Sobel-Operator fallen Filtermasken, die eine Approximation der ersten Ableitung in eine Richtung berechnen und orthogonal dazu das Bild glätten.

Mit den Sobel-Operatoren S_x und S_y lassen sich die geglätteten Approximationen der partiellen Ableitungen folgendermaßen berechnen

$$g_x = S_x * I \qquad g_y = S_y * I,$$

wobei

$$S_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \qquad S_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}.$$

Danach werden die beiden beiden gefalteten Bilder folgendermaßen addiert

$$O(i,j) = |g_x(i,j) + g_y(i,j)|.$$

Weiterhin wird, ähnlich wie beim SUSAN-Kantendetektor, eine Non-Maximum-Suppression durchgeführt. Die Kantenrichtung ist durch

$$\varphi(i,j) = \begin{cases} \arctan\left(\frac{g_y(i,j)}{g_x(i,j)}\right) & \text{falls } g_x(i,j) \neq 0 \\ \frac{\pi}{2} & \text{sonst} \end{cases},$$

bestimmt.

Zum Vergleich mit meiner Implementation des SUSAN-Kantendetektors ziehen wir nun die OpenCV-Implementation des Canny-Kantendetektors heran. Durch die Vektorisierbarkeit der Teilschritte lässt sich über Cannys Kantendetektor sagen, dass die Implementation in Python 3 schnellere Laufzeiten mit sich bringt. Nichtlineare Operationen müssen in Python 3, wie in den meisten anderen Programmiersprachen, über Schleifen geregelt werden. Diese sind verglichen langsam zu beispielsweise der Programmiersprache C. Das liegt begründet im dynamischen Charakter von Python [7].

Für lineare Probleme ist die Vektorisierung ein Weg, die Dynamik Pythons auszu-

nutzen, um Ergebnisse schnell zu berechnen. Der nichtlineare Charakter des SUSAN-Detektors lässt allerdings nur einfache Parallelisierungen zu, keine Vektorisierung.

Allerdings bringt der SUSAN-Kantendetektor einen besonders großen Vorteil mit sich. Cannys Kantendetektor findet nämlich im Gegensatz zum SUSAN-Kantendetektor keine Ecken, wie auch bereits in [1] beschrieben wurde.

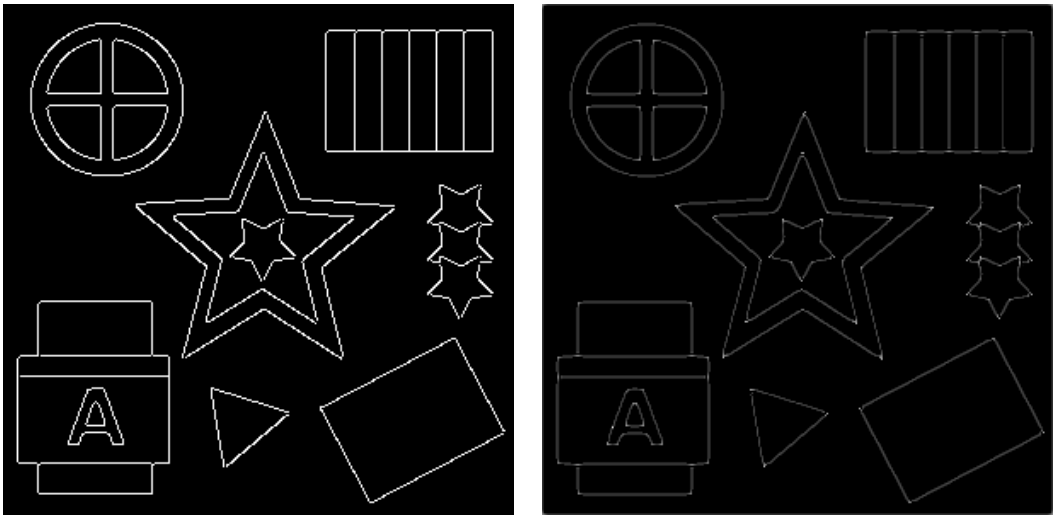

Abbildung 3.15: Links: Canny, Rechts: SUSAN.

3.4 Heuristik

Durch die folgende Heuristik aus [1] soll dargestellt werden, wie das Prinzip des SUSAN-Kantendetektors funktioniert. Die Heuristik ist nicht als Beweis zu verstehen, da für diese Heuristik Annahmen getroffen werden, die nicht auf Digitalbilder zutreffen.

Angenommen wir haben ein eindimensionales Bild. Unser Ziel ist es, genau die Stellen zu markieren, die den größten Grauwertunterschied zu ihrer Nachbarschaft haben. Das klassische Werkzeug der Analysis, um genau diese Stellen zu identifizieren, ist die Ableitung. Leiten wir eine zweimal differenzierbare, stetige Funktion I zweimal ab, so erhalten wir eine Funktion I'', dessen Nullstellen die Stellen des größten Unterschieds in I repräsentieren.

So eine Ableitung existiert allerdings nicht für Digitalbilder. Darum nehmen wir zunächst an, dass unser Bild eine solche Funktion $I : [0, B-1]_{\mathbb{R}} \to [0, 255]_{\mathbb{R}}$ ist. Implizit wird im Folgenden die Annahme getroffen, dass I eine bijektive Funktion ist. Das ist im Allgemeinen zwar nicht der Fall, allerdings liegt uns bei der digitalen Verarbeitung von Bildern im Gegensatz zur eindimensionalen Analysis immer beides vor: Die Position und die Intensität eines Pixels lassen sich hier durch geschicktes Speichern aufeinander zurückführen.

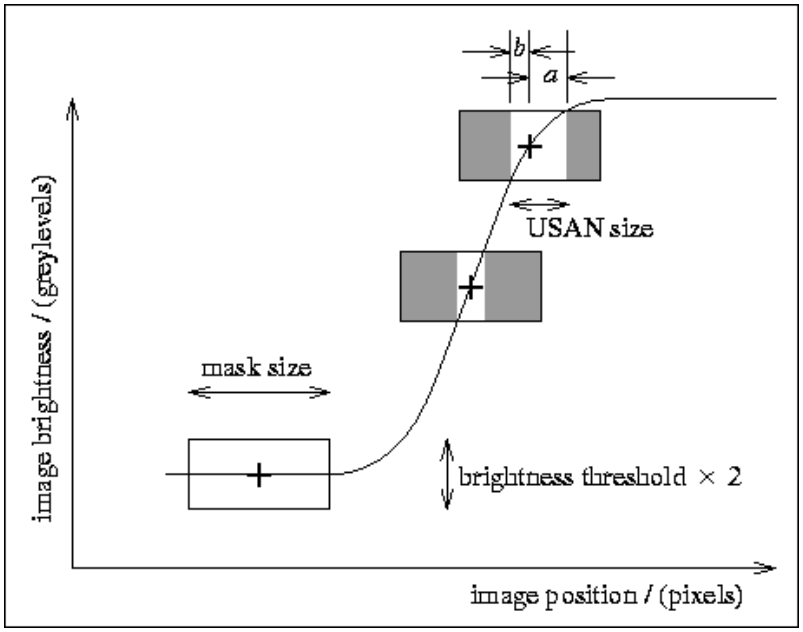

Abbildung 3.16: Das Bild I (Bild entnommen aus [1]). Das Kreuz steht für den Nucleus der USAN und die Kästen für die Maske.

Der SUSAN-Kantendetektor beruht auf folgendem Prinzip: Für einen Schwellwert t gibt es, abhängig von der spezifischen Stelle im Bild x_0 eine obere Grenze $x_0 + a(x_0)$ und eine untere Grenze $x_0 - b(x_0)$ der USAN. In der Abbildung 3.16 ist an drei verschiedenen Stellen dargestellt, wie das Fenster für verschiedene x_0 variieren kann. Dies ist in Abbildung 3.16 für x_0 am mittleren Kreuz der Fall. Dann gilt an den Grenzen:

$$I(x_0 + a(x_0)) = I(x_0) + t$$
$$I(x_0 - b(x_0)) = I(x_0) - t.$$

Hier gehen wir von der Darstellung in Abbildung 3.16 aus. Falls die Intensitätsfunktion I fallend ist, sind die Formeln entsprechend zu ändern. Das Prinzip des SUSAN-Kantendetektors lässt sich folgendermaßen formulieren: An genau den Stellen, an denen $n(x_0) := a(x_0) + b(x_0)$ ein Minimum hat, das heißt wenn die Fensterbreite am kleinsten ist, ist die Antwort $A(x_0) > 0$. Wir erhalten unter der Annahme, dass n an x_0 ein lokales Minimum erreicht:

$$n'(x_0) = a'(x_0) + b'(x_0) = 0$$

Die obigen Gleichungen können wir nach a und b umstellen. So erhalten wir

$$a(x_0) = x(I(x_0) + t) - x_0$$
$$b(x_0) = x_0 - x(I(x_0) - t),$$

wobei x die Umkehrfunktion von I ist. Damit ist $n(x_0) = x(I(x_0)+t) - x(I(x_0)-t)$ und daraus folgt

$$n'(x_0) = x'(I(x_0)+t) \cdot I'(x_0) - x'(I(x_0)-t) \cdot I'(x_0) = 0,$$

anders formuliert,

$$x'(I(x_0)+t) \cdot I'(x_0) = x'(I(x_0)-t) \cdot I'(x_0)$$

und damit

$$x'(I(x_0)+t) = x'(I(x_0)-t) \quad \Longleftrightarrow \quad x'(I(x_0+a(x_0))) = x'(I(x_0-b(x_0))),$$

falls $I'(x_0) \neq 0$, das heißt falls die Bildfunktion I nicht konstant ist. Aus der Inversenregel der Differenzialrechnung erhalten wir

$$x'(I(x_0+a(x_0))) = \frac{1}{I'(x_0+a(x_0)),}$$
$$x'(I(x_0-b(x_0))) = \frac{1}{I'(x_0-b(x_0)),}$$

und dadurch

$$I'(x_0+a(x_0)) = I'(x_0-b(x_0)),$$

falls $I'(x_0+a(x_0)) \neq 0$. Folglich ist die Approximation der zweiten Ableitung

$$I''(x_0) \approx \frac{I'(x_0+a(x_0)) - I'(x_0-b(x_0))}{a(x_0)+b(x_0)} = 0$$

Also ist die Antwort des SUSAN-Kantendetektors in x_0 maximal, wenn die zweite Ableitung der Bildfunktion $I''(x_0) = 0$ ist. Tatsächlich verwendet der SUSAN-Kantendetektor allerdings keine Ableitungen, sondern diskrete Werte. Der SUSAN-Algorithmus bestimmt lediglich die Anzahl der ähnlichen Pixel in einer Umgebung.

Abbildung 3.17: Filtermaske im 1D-Bild. Der Nucleus ist gelb markiert. Die orangenen Pixel liegen innerhalb der Maske. Die weißen Pixel liegen außerhalb der Maske.

In einem eindimensionalen Digitalbild $P : [0, B-1]_{\mathbb{Z}} \to [0, 255]_{\mathbb{Z}}$ identifizeren wir durch eine Maske wie zum Beispiel in Abbildung 3.17 die USAN der einzelnen Pixel, wie auch schon im Algorithmus in 3.1. Dabei stellen wir für jede USAN fest, wie viele Pixel r ähnlich zu dem Nucleus r_0 in der Mitte der Maske sind. Wir erinnern uns an die Definitionen aus

dem Algorithmus

$$A(r_0) = \max\{0, g - n(r_0)\},\ n(r_0) = \sum_r c_t(r, r_0),\ c_t(r, r_0) = \begin{cases} 1 & \text{falls } |I(r) - I(r_0)| \leq t \\ 0 & \text{sonst} \end{cases}.$$

Anhand eines Beispielbildes wird nun vorgeführt, wie der SUSAN-Kantendetektor eine Kante identifiziert.

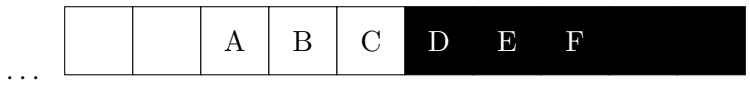

Abbildung 3.18: Die weißen Pixel haben den Grauwert 255, die schwarzen Pixel den Grauwert 0

Wir stellen fest, dass die mit A-F markierten Pixel verschiedene Anzahlen an ähnlichen Nachbarn in ihrer Maske haben.

$$n(A) = 4 \quad n(B) = 3 \quad n(C) = 2$$
$$n(D) = 2 \quad n(E) = 3 \quad n(F) = 4$$

Entsprechend finden wir eine Antwort mit $g = 4$ wie in Abbildung 3.19.

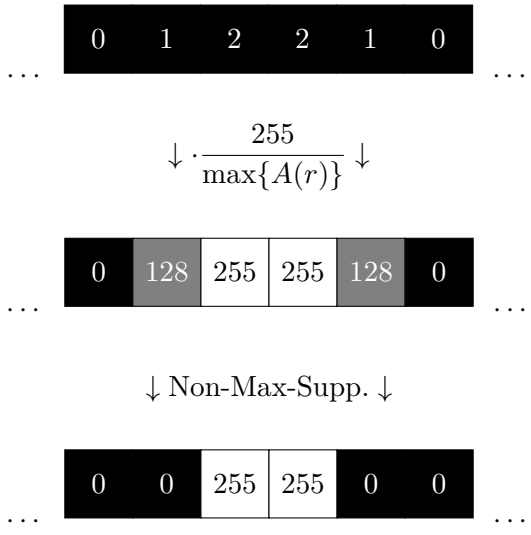

Abbildung 3.19: Antwort

Durch das SUSAN-Prinzip und die Non-Maximum-Suppression konnten wir die Kante ohne Ableitung bestimmen.

Kapitel 4

Diskussion

Der SUSAN-Kantendetektor findet Kanten auf digitalen Bildern einfach, schnell und pixelgenau, wie wir an mehreren Beispielen erkennen konnten. Im Gegensatz zum Kantendetektor von Canny eignet sich SUSAN auch zur Identifikation von Ecken.

Meine persönliche Implementation könnte trotz Einbindung von `multiprocessing` noch schneller sein, dazu würden sich einerseits GPU-gestützte Methoden sowie auch eine Implementation in einer weniger dynamischen Sprache wie zum Beispiel `C++` anbieten (siehe [7]). Die Implementation in Python 3 bietet aber den großen Vorteil der Modifizierbarkeit, was einerseits die Fehlersuche und andererseits das Experimentieren mit verschiedenen Maskenträgern und Randbedingungen erleichtert. Im Sinne einer Bachelorarbeit, welche nicht notwendigerweise einen industriellen Nutzen tragen muss, bewerte ich die Wahl der Sprache aus diesem Grund als gelungen.

Der Kantendetektor von Canny hingegen ist durch die Vektorisierbarkeit der Faltung in Python 3 mit kürzeren Laufzeiten zu realisieren. Dadurch fielen nämlich zahlreiche `for`-Schleifen weg, welche aufgrund vorher genannter dynamischer Natur von Python 3 der Flaschenhals meiner Implementation des SUSAN-Kantendetektors sind.

Der Kantendetektor von Canny basiert auf dem Prinzip der Diskretisierung der Ableitung, wohingegen der SUSAN-Kantedetektor auf einem kombinatorischen Prinzip basiert, welches in [1] nur über eine Analogie mit der eindimensionalen Analysis erklärt wurde, ähnlich, wie in Kapitel 3.4. Das Fehlen der mathematischen Gründlichkeit für eine Herleitung des SUSAN-Kantendetektors steht einer vergleichsweise einfachen Strategie des Zählens der ähnlichen Nachbarn gegenüber.

Literaturverzeichnis

[1] Stephen M. Smith, J. Michael Brady,
SUSAN - A New Approach to Low Level Image Processing,
International Journal of Computer Vision 23(1), 45-78,
1997

[2] Stephen M. Smith,
Edge Thinning Used in the SUSAN Edge Detector,
Technical Report TR95SMS5,
1995

[3] Robert Fisher, Simon Perkins, Ashley Walker, Erik Wolfart,
Gaussian Smoothing,
The Hypermedia Image Processing Reference 2,
2004

[4] Pillow: the friendly PIL fork,
`https://python-pillow.org/`

[5] NumPy,
`https://numpy.org/`

[6] multiprocessing - Process-based parallelism,
`https://docs.python.org/3/library/multiprocessing.html`

[7] Jake VanderPlas,
Why Python is Slow: Looking Under the Hood,
`https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/`,
2014

[8] Markus Konrad,
Vectorization and parallelization in Python with NumPy and Pandas,
`https://datascience.blog.wzb.eu/2018/02/02/vectorization-and-parallelization-in-python-with-numpy-and-pandas/`,
2018

[9] John Canny,
Finding lines and edges in images,
Technical Report TM-720,
Artificial Intelligence Laboratory, Massachusetts Institute of Technology,
1983

[10] Julian Lüken,
Implementation des SUSAN Kantendetektors in Python 3,
`https://gitlab.gwdg.de/julian.lueken/susan/`,
2019

Anhang A

Quellcode

```
import numpy as np
import sys
import multiprocessing as mp
from PIL import Image

np.set_printoptions(precision=3)

n_proc = mp.cpu_count()

_mask37_rad = 3.4
_mask37 = np.matrix([
                [0,0,1,1,1,0,0],
                [0,1,1,1,1,1,0],
                [1,1,1,1,1,1,1],
                [1,1,1,1,1,1,1],
                [1,1,1,1,1,1,1],
                [0,1,1,1,1,1,0],
                [0,0,1,1,1,0,0]
        ], dtype='?')

_mask9_rad = 1.4
_mask9 = np.matrix([
        [1,1,1],
        [1,1,1],
        [1,1,1]
], dtype="?")

class Susan:
        # default mask with 37 neighbors per pixel
        # sets initial mask, file path and comparison function
        def __init__(self, path, mask = "mask37", compare = "exp_lut"):
                self.load(path)

                if mask == "mask37":
                        self._set_mask(_mask37)
                        self.mask_rad = _mask37_rad
                if mask == "mask9":
                        self._set_mask(_mask9)
                        self.mask_rad = _mask9_rad

                self._has_lut = True
                if compare == "naive":
                        self.compare = self._compare_naive
                if compare == "exp":
                        self.compare = self._compare_exp
                if compare == "exp_lut":
                        self.compare = self._compare_exp_lut
                        self._exp_lut = np.zeros(1024)
                        self._has_lut = False

        def _set_path(self, path):
                self.path = path

```

```
54          def _set_image(self, img):
55                  self.img = img
56
57          def _set_mask(self, mask):
58                  self.mask = mask
59                  self.center = (self.mask.shape[0]//2, self.mask.shape[1]//2)
60
61                  self.nbd_size = mask.sum()-1
62                  if self.nbd_size <= 1:
63                          print("Error: Mask should have more than one item (has %d)" % self.nbd_size)
64                          sys.exit(0)
65
66                  self._init_nbd()
67
68          # loads image into susan module
69          def load(self, path, space = "L"):
70                  self._set_path(path)
71
72                  try:
73                          #self.img = cv2.imread(path, 0)
74                          self.img = np.array(Image.open(self.path).convert(space), dtype="i")
75
76                  except:
77                          print("Error: Cannot open", self.path)
78                          sys.exit(0)
79
80                  self.height = self.img.shape[0]
81                  self.width      = self.img.shape[1]
82
83          # get indices from mask
84          def _init_nbd(self):
85                  self.mask_nbd = []
86                  for k in range(self.mask.shape[0]):
87                          for l in range(self.mask.shape[1]):
88                                  if self.mask[k,l] == 1:
89                                          x = k-self.center[0]
90                                          y = l-self.center[1]
91                                          self.mask_nbd.append((x,y))
92
93
94          # compare functions
95          def _compare_naive(self, img, a, b, t):
96                  if np.abs(img[a] - img[b]) <= t:
97                          return 1
98                  return 0
99
100         def _compare_exp(self, img, a, b, t):
101                 return np.exp(-((img[a] - img[b])/t)**6)
102
103         def _init_lut(self, t):
104                 for c in range(-511, 512):
105                         self._exp_lut[c] = np.exp(-(c/t)**6)
106                 self._has_lut = True
107
108         def _compare_exp_lut(self, img, a, b, t):
109                 return self._exp_lut[img[a]-img[b]]
110
111         # make array flat
112         def __flatten(self, A):
113                 return A.flatten()
114
115         # turn flat array back into matrix of image size
116         def __unflatten(self, A):
117                 uf = np.zeros((self.height, self.width))
118                 for i in range(self.height):
119                         for j in range(self.width):
120                                 uf[i,j] = A[i*self.width + j]
121                 return uf
122
123         # main function for usan
124         # computes usan area, usan value and gradient
125         # for one chunk of height of total size (chunkend - chunkstart) * imagewidth
126         # note that (i,j) = r_0 and (x,y) = r
127         def _nbd_compare_mp(self, start, end, t, geometric):
128                 diam = np.ceil(2*self.mask_rad)                 # mask diameter
```

```
129
130                 for i in range(start, end):
131                         for j in range(self.width):
132                                 usan_area               = 0                     # sum over all comparisons in
                                        ↪ usan
133
134                                 i_cog                   = 0                     # center of gravity (vertical
                                        ↪ position)
135                                 j_cog                   = 0                     # center of gravity (horizontal
                                        ↪  position)
136
137                                 i2_intra                = 0                     # second moment of usan value (
                                        ↪ vertical position)
138                                 j2_intra                = 0                     # second moment of usan value (
                                        ↪ horizontal position)
139                                 ij_intra                = 0                     # second moment of usan value (
                                        ↪ sign)
140
141                                 # calculate center of gravity and usan value
142                                 for r in self.mask_nbd:
143                                         x = i+r[0]
144                                         y = j+r[1]
145
146                                         # flip boundaries if necessary
147                                         if x < 0 or x >= self.height-1:
148                                                 x = i-r[0]
149                                         if y < 0 or y >= self.width-1:
150                                                 y = j-r[1]
151
152                                         curr = self.compare(self.img, (i,j), (x,y), t)
153
154                                         if curr != 0:
155                                                 usan_area = usan_area + curr
156
157                                                 i_cog += x * curr
158                                                 j_cog += y * curr
159
160                                                 i2_intra += (r[0]**2) * curr
161                                                 j2_intra += (r[1]**2) * curr
162                                                 ij_intra += r[0] * r[1] * curr
163
164                                 i_cog = i_cog / usan_area
165                                 j_cog = j_cog / usan_area
166
167
168
169                                 self.i_cogs[i*self.width+j] = i_cog
170                                 self.j_cogs[i*self.width+j] = j_cog
171
172                                 # get direction for non max suppression
173                                 direction = 2   # 'no edge' marker
174                                 if geometric - usan_area > 0:
175                                         self.dist_from_cog[i*self.width+j] = np.sqrt((i_cog - i)**2 + (j_cog -
                                                ↪ j)**2)
176
177                                         # inter pixel case
178                                         if usan_area > diam and self.dist_from_cog[i*self.width+j] >= 1:
179                                                 if j_cog != j:
180                                                         direction = np.arctan((i-i_cog)/(j-j_cog))
181                                                 else:
182                                                         direction = np.pi/2
183
184                                         # intra pixel case
185                                         elif i2_intra != 0:
186                                                 phi = (j2_intra/i2_intra)
187                                                 if ij_intra == 0:
188                                                         if j2_intra > i2_intra:
189                                                                 direction = np.pi/2
190                                                         else:
191                                                                 direction = 0
192                                                 else:
193                                                         direction = -np.sign(ij_intra) * np.arctan(phi)
194
195                                         else:
196                                                 direction = np.pi/2
```

```
197
198
199
200                             index = i*self.width+j
201                             self.direction[index]   = direction
202                             self.response[index]    = max(0, geometric - usan_area)
203
204         # classification of direction and non max suppression
205         # keep in mind that the directional values for each pixel are stored in self.direction
206         # which is computed in the _nbd_compare_mp function
207         _orientations = np.pi*np.array([-0.375, -0.125, 0.125, 0.375])
208         def _suppress_nonmax_mp(self, start, end):
209                 for i in range(start, end):
210                         for j in range(self.width):
211                                 if self.direction[i*self.width+j] != 2:
212                                         max_here = True
213                                         index = i*self.width+j
214                                         r_curr = self.response[index]
215
216                                         di_p = 1
217                                         di_n = -1
218
219                                         dj_p = 1
220                                         dj_n = -1
221
222                                         if i + di_p >= self.height:
223                                                 di_p = -di_p
224                                         if i - di_n < 0:
225                                                 di_n = -di_n
226                                         if j + dj_p >= self.width:
227                                                 dj_p = -dj_p
228                                         if j - dj_n < 0:
229                                                 dj_n = -dj_n
230
231                                         di_p *= self.width
232                                         di_n *= self.width
233
234
235                                         # negative diagonal
236                                         if self.direction[index] > self._orientations[0] and self.direction[
                                                ↪ index] <= self._orientations[1]:
237                                                 if self.response[index+dj_p+di_n] > r_curr or self.response[
                                                        ↪ index+dj_n+di_p] > r_curr:
238                                                         max_here = False
239
240                                         # vertical
241                                         elif self.direction[index] > self._orientations[1] and self.direction[
                                                ↪ index] <= self._orientations[2]:
242                                                 if self.response[index+dj_p] > r_curr or self.response[index+
                                                        ↪ dj_n] > r_curr:
243                                                         max_here = False
244
245                                         # positive diagonal
246                                         elif self.direction[index] > self._orientations[2] and self.direction[
                                                ↪ index] <= self._orientations[3]:
247                                                 if self.response[index+dj_p+di_p] > r_curr or self.response[
                                                        ↪ index+dj_n+di_n] > r_curr:
248                                                         max_here = False
249
250                                         # horizontal
251                                         else:
252                                                 if self.response[index+di_p] > r_curr or self.response[index+
                                                        ↪ di_n] > r_curr:
253                                                         max_here = False
254
255                                         # apply nonmax suppression
256                                         if not max_here:
257                                                 self.response[index] = 0
258
259
260         # heat map visualization for self.direction
261         _cmap = np.array([
262                                 [255,   0,              0],
263                                 [255,   255,    0],
264                                 [0,             255,    0],
```

```
                                [255,   0,              255],
                                [255,   0,              0]
                        ], dtype='i')

        def _caffine(self, phi):
                x = phi/np.pi+0.5
                l = len(self._cmap)-1

                h = x*l

                if int(h) < l:
                        return np.uint8((1-h%1) * self._cmap[int(h)] + (h%1) * self._cmap[int(h)+1])

                return self._cmap[-1]

        def __make_heatmap(self, filename):
                A = np.linspace(-np.pi/2,np.pi/2,100)
                for a in A:
                        self._caffine(a)

                O = np.array([[[0]*3]*self.width]*self.height, dtype="i")

                for i in range(self.height):
                        for j in range(self.width):
                                if self.direction[i*self.width+j] == 2:
                                        O[i,j] = [0,0,0]
                                else:
                                        O[i,j] = self._caffine(self.direction[i*self.width+j])

                self.save(O, filename + ".png")



        # thinning algorithm - this needs to be thoroughly improved.
        _direct_neighbors = np.array([
                                                [-1,-1],[-1, 0],[-1, 1],
                                                [ 0,-1],        [ 0, 1],
                                                [ 1,-1],[ 1, 0],[ 1, 1]
        ])

        def __approx(self, a, b, eps):
                return a + eps > b and a - eps < b

        def _thinout(self, start, end):
                # maximum reach of line completion
                maxlen = 3

                # rotational slack
                eps = np.pi/32

                for i in range(start, end):
                        for j in range(1,self.width-1):
                                if self.response[i*self.width+j] > 0:
                                        neighbor_count = 0
                                        dx = []
                                        dy = []
                                        for r in self._direct_neighbors:
                                                x = i+r[0]
                                                y = j+r[1]

                                                # filter direct neighbors with response
                                                if x >= 0 and x < self.height and y >= 0 and y < self.width:
                                                        if self.response[x*self.width+y] > 0:
                                                                neighbor_count += 1
                                                                dx.append(r[0])
                                                                dy.append(r[1])


                                        ### cases for number of neighbors ###

                                        # probably a false positive
                                        if neighbor_count == 0:
                                                self.response[i*self.width+j] = 0

```

```
340
341                                     # try line completion if only one neighbor
342                                     elif neighbor_count == 1:
343                                             if dx[0] != 0 and dy[0] != 0:
344                                                     if self.response[(i+2*dx[0])*self.width+j] > 0 and self
                                                            ↪ .response[i*self.width+(j+2*dy[0])] > 0:
345                                                             self.response[i*self.width+j] = 0
346
347                                             if i >= maxlen+1 and i < self.height-maxlen-1 and j >= maxlen+1
                                                    ↪  and j < self.width-maxlen-1:
348                                                     # create array to mark pixels for line completion. will
                                                            ↪  only complete line if direction of the "other
                                                            ↪ side" matches.
349                                                     inbtwn = np.zeros(maxlen+2, dtype="i")
350                                                     linecnt = 0
351                                                     for k in range(len(inbtwn)):
352                                                             inbtwn[k] = (i-k*dx[0])*self.width+(j-k*dy[0])
353
354
355                                                     # rotational slackness
356                                                     for k in range(1,maxlen+2):
357                                                             if inbtwn[k] < self.width*self.height and
                                                                    ↪ inbtwn[k] > 0:
358                                                                     if self.response[inbtwn[k]] > 0 and
                                                                            ↪ self.__approx(self.direction[
                                                                            ↪ inbtwn[k]], self.direction[
                                                                            ↪ inbtwn[0]], eps):
359                                                                             linecnt = k
360
361                                                     # draw line
362                                                     for k in range(1, linecnt):
363                                                             self.response[inbtwn[k]] = 0.5*self.response[
                                                                    ↪ inbtwn[0]] + 0.5*self.response[inbtwn[
                                                                    ↪ linecnt]]
364
365
366                                     elif neighbor_count == 2:
367                                             if dx[0] + dx[1] != 0 and dy[0] + dy[1] != 0:
368                                                     if self.__approx(self.direction[(dx[0]+i)*self.width+(
                                                            ↪ dy[0]+j)], self.direction[(dx[1]+i)*self.width
                                                            ↪ +(dy[1]+j)], eps):
369                                                             self.response[i*self.width+j] = 0
370
371                                     elif neighbor_count == 3:
372                                             if (dx[0] == dx[1] and dx[1] == dx[2]) or (dy[0] == dy[1] and
                                                    ↪ dy[1] == dy[2]):
373                                                     self.response[i*self.width+j] = 0
374
375
376     _fivebyfive_neighbors = np.array([
377                                                             [-2,-1],[-2, 0],[-2, 1],
378                                             [-2,-1],[-1,-1],[-1, 0],[-1, 1],[-1, 2],
379                                             [-2, 0],[ 0,-1],                [ 0, 1],[ 0, 2],
380                                             [-2, 1],[ 1,-1],[ 1, 0],[ 1, 1],[ 1, 2],
381                                                             [ 2,-1],[ 2, 0],[ 2, 1],
382     ])
383     def _detect_corners(self, start, end, delta_g):
384             # get corners from edge detection response
385             for i in range(start, end):
386                     for j in range(self.width):
387
388                             # if pixel is too close to cog, it's not a corner
389                             if self.dist_from_cog[i*self.width+j] <= np.sqrt(2):
390                                     continue
391
392                             # check if pixels on the way to center of gravity lie in the USAN, only then we
                                    ↪  have a corner candidate
393                             flag = True
394
395                             i_dist = int(np.round(self.i_cogs[i*self.width+j],0)) - i
396                             j_dist = int(np.round(self.j_cogs[i*self.width+j],0)) - j
397
398                             maxdist = max(i_dist, j_dist)
399                             for k in range(2, maxdist+1):
400                                     x = int(np.round(i_dist/k,0))
```

```
401                                     y = int(np.round(j_dist/k,0))
402
403                                     if self.response[(x+i)*self.width+(y+j)] == 0:
404                                             flag = False
405
406                             if flag:
407                                     self.corners[i*self.width+j] = max(0, self.response[i*self.width+j] -
                                            ↪ delta_g)
408
409
410             # suppress everything that is not a local maximum in a 5x5 area
411             for i in range(start, end):
412                     if i >= 1 and i < self.height-1:
413                             for j in range(self.width):
414                                     if self.corners[i*self.width+j] > 0:
415                                             for r in self._fivebyfive_neighbors:
416                                                     x = i+r[0]
417                                                     y = j+r[1]
418
419                                                     if x >= 2 and x < self.height-2 and y >= 2 and y < self
                                                            ↪ .width-2:
420                                                             if self.corners[i*self.width+j] <= self.corners
                                                                    ↪ [x*self.width+y]:
421                                                                     self.corners[i*self.width+j] = 0
422                                                                     break
423
424
425
426
427
428
429
430     def _overlay(self, filename, corners = False):
431             O = np.array([[[0]*3]*self.width]*self.height, dtype="i")
432             for i in range(self.height):
433                     for j in range(self.width):
434                             O[i,j,0] = self.img[i,j]
435                             O[i,j,1] = self.img[i,j]
436                             O[i,j,2] = self.img[i,j]
437
438             for i in range(self.height):
439                     for j in range(self.width):
440                             if self.response[i*self.width+j] != 0:
441                                     O[i,j] = [(self.response[i*self.width+j]+2*255)/3,0,0]
442
443             # Overlay for corner detection (to be fixed)
444             if corners:
445                     for i in range(self.height):
446                             for j in range(self.width):
447                                     v = self.corners[i*self.width+j]
448                                     if v > 0 and i < self.height-2 and j < self.width-2 and i > 2 and j >
                                            ↪ 2:
449                                             color = [0,255,0]
450                                             O[i-2,  j-2]    = color
451                                             O[i-1,  j-2]    = color
452                                             O[i,    j-2]    = color
453                                             O[i+1,  j-2]    = color
454                                             O[i+2,  j-2]    = color
455
456                                             O[i-2,  j-1]    = color
457                                             O[i+2,  j-1]    = color
458
459                                             O[i-2,  j]              = color
460                                             O[i+2,  j]              = color
461
462                                             O[i-2,  j+1]    = color
463                                             O[i+2,  j+1]    = color
464
465                                             O[i-2,  j+2]    = color
466                                             O[i+2,  j+2]    = color
467
468                                             O[i-2,  j+2]    = color
469                                             O[i-1,  j+2]    = color
470                                             O[i,    j+2]    = color
471                                             O[i+1,  j+2]    = color
```

```
472                                             O[i+2,  j+2]    = color
473
474             self.save(O, filename + ".png")
475
476
477         def __execute_and_wait(self, jobs):
478                 for job in jobs:
479                         job.start()
480                 for job in jobs:
481                         job.join()
482
483         def __execute_and_save(self, filename, jobs, chunks):
484                 self.__execute_and_wait(jobs)
485                 R = self.__unflatten(self.response)/max(self.response)*255
486                 self.save(R, filename+".png")
487
488
489
490
491         def detect_edges_mp(self, t, filename, geometric = True, nms = True, thin = False, heatmap = True,
                ↪ overlay = True, corners = False):
492                 self.response           = mp.Array('d', self.width*self.height)
493                 self.direction          = mp.Array('d', self.width*self.height)
494
495                 self.corners            = mp.Array('d', self.width*self.height)
496
497                 self.i_cogs             = mp.Array('d', self.width*self.height)
498                 self.j_cogs                     = mp.Array('d', self.width*self.height)
499                 self.dist_from_cog      = mp.Array('d', self.width*self.height)
500
501                 if not self._has_lut:
502                         self._init_lut(t)
503
504                 if geometric:
505                         g = .75*self.nbd_size
506                 else:
507                         g = self.nbd_size
508
509                 n_proc = mp.cpu_count()                 # number of cores
510                 chunk_size = (self.height)//n_proc
511                 remainder = (self.height)%n_proc
512
513                 # find appropriate chunking
514                 pivot = 0
515                 chunks = np.uint16(np.zeros(n_proc+1))
516                 chunks[0] = 0
517                 for i in range(n_proc):
518                         if remainder > 0:
519                                 pivot += chunk_size+1
520                                 remainder -= 1
521                         else:
522                                 pivot += chunk_size
523                         chunks[i+1] = pivot
524
525                 jobs = [mp.Process(
526                                 target = self._nbd_compare_mp,
527                                 args = (chunks[i], chunks[i+1], t, g))
528                                 for i in range(len(chunks)-1)
529                 ]
530
531                 # No non-max suppression
532                 self.__execute_and_save(filename+"_raw", jobs, chunks)
533
534                 # Directional heatmap, basically direction of gradient of the edges
535                 if heatmap:
536                         A = self.__make_heatmap(filename+"_heat")
537
538                 # Non-max suppression
539                 if nms:
540                         jobs = [mp.Process(
541                                         target = self._suppress_nonmax_mp,
542                                         args = (chunks[i], chunks[i+1]))
543                                         for i in range(len(chunks)-1)
544                         ]
545                         self.__execute_and_save(filename+"_nonmax_supp", jobs, chunks)
```

```
546
547                 # Thinning (not done yet)
548                 if thin:
549                         jobs = [mp.Process(
550                                         target = self._thinout,
551                                         args = (chunks[i], chunks[i+1]))
552                                         for i in range(len(chunks)-1)
553                         ]
554                         self.__execute_and_save(filename+"_thinned", jobs, chunks)
555
556                 # SUSAN corner detection
557                 if corners:
558                         delta_g = 0.25 * self.nbd_size
559                         jobs = [mp.Process(
560                                         target = self._detect_corners,
561                                         args = (chunks[i], chunks[i+1], delta_g))
562                                         for i in range(len(chunks)-1)
563                         ]
564                         self.__execute_and_wait(jobs)
565
566                 # Overlay for edge detection
567                 if overlay:
568                         self._overlay(filename+"_overlay", corners)
569
570
571
572
573
574         def save(self, r, filename = "a.png"):
575                 """
576                 Saves image referenced in the ConstantDenoiser object.
577
578                 Parameters
579                 ----------
580                 filename: String, optional
581                         Path, to which the image will be saved.
582                         Will save to ./a.png upon not specifying a String
583
584                 """
585                 try:
586                         #cv2.imwrite(filename, np.uint8(r))
587                         a = Image.fromarray(np.uint8(r))
588                         a.save(filename)
589
590                         print("Saved␣file␣to", filename)
591                 except:
592                         print("Error:␣Couldn't␣save", filename)
593                         return
```