

# Maschinelle Mustererkennung Teil 5

## Bildvergleiche II

Von Gilbert Brands

### Einführung

Im letzten Teil der Serie haben wir nach Bildern mit gleichem Inhalt in einer größeren Menge von Bildern gesucht, wobei die Bilder in unterschiedlichen Formaten und Bearbeitungszuständen vorliegen durften. Der Inhalt musste aber in allen Bildern komplett der gleiche sein.

Ist ein Bild beschnitten, stellt es nur noch einen Ausschnitt des anderen dar. Mit den gleichen Methoden konnten wir feststellen, ob dies der Fall ist, jedoch war Voraussetzung, dass die Bilder in exakt der gleichen Auflösung vorliegen. Das Bild mit den kleineren Abmessungen muss komplett im größeren vorhanden sein.

Die Methoden funktionieren nicht mehr, wenn die Bilder zusätzlich skaliert sind, d.h. die Bildgröße nachträglich verändert wurde. Sie funktionieren auch nicht mehr, wenn das Ausschnittbild in ein anderes Bild integriert wurde, um einen unzutreffenden Eindruck zu vermitteln. Dabei könnte es sich beispielsweise um einen Kompromittierungsversuch handeln, bei dem eine Person aus einem Bild von einem harmlosen Fest in ein Bild mit kompromittierendem Inhalt kopiert wird, um ihr zu schaden. Rückt man vom Begriff des gleichen Inhalts etwas ab, führt die letzte Aufgabenstellung auch zu der Frage, wie man wiederkehrende Motive erkennen kann. Diesen Fragen wollen wir nun nachgehen, wie zuvor auch wieder mit einfachen Programmierbeispielen, die allerdings bei der komplexer werdenden Materie mehr von fertigen Funktionen des OpenCV-Paketes Gebrauch machen wird als in den vorhergehenden Teilen. Ich werde aber trotzdem so weit auf die Methodik eingehen, dass es dem interessierten Leser möglich sein wird, den einen oder anderen Versuch selbst zu programmieren

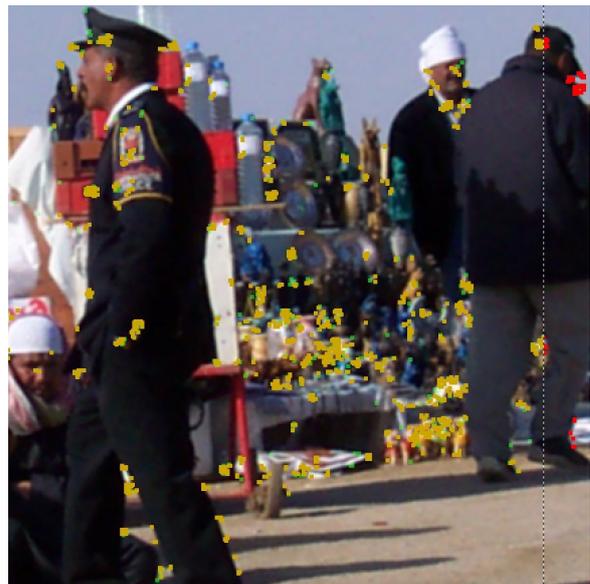
### Übereinstimmende Merkmale

Nehmen wir an, wir hätten Filteralgorithmen, die in der Lage sind, bestimmte Pixelmuster zu erkennen. Genauer formuliert: die Algorithmen würden die Pixel eines Bildes unabhängig vom Gesamthalt analysieren und Pixel mit bestimmten Umgebungen markieren. Als Ergebnis würde Bild mit vielen markierten Stellen entstehen.

Sehen wir uns dies an einem Beispiel an. Die beiden folgenden Bilder enthalten solche Markierungen, wobei das erste (linke) aus dem Referenzbild aus dem vorhergehenden Teil erzeugt wurde, das zweite aus dem Ausschnittbild (beide dargestellte Bilder sind Ausschnitte aus den Gesamtbildern):



Für das Auge ist das erst einmal verwirrend, weil sehr viele Pixel von den Algorithmen markiert wurden, wobei „sehr viele“ im Verhältnis zu der Gesamtanzahl der Pixel allerdings nur auf den primären Sinneseindruck zutrifft. Bringen wir nun beide Bilder zur Deckung,



so zeigt sich, dass die meisten markieren Pixel des einen Bildes mit denen des anderen koinzidieren. Geringe Unterschiede existieren, weil Bilder unterschiedlichen Gesamtgröße ausgewertet wurden, aber mit solchen Ausreißern muss man leben.

Die Muster, die die markierten Pixel bilden, sind das eigentlich charakteristische für die dargestellte Szene. Wir könnten die beiden Teilbilder in völlig andere Umgebungen kopieren und trotzdem würden die Algorithmen, da sie auf Pixelebene arbeiten, diese Muster – von Ausreißern abgesehen – wieder erzeugen und wir wären in der Lage, die Szenen in beiden Bildern als Kopien aus einem anderen Bild wiederzufinden.

Nun sind die Beispielbilder nur sehr kleine Ausschnitte aus den Gesamtbildern. Wie findet man die markierten Pixel des einen Bildes im anderen wieder? Man könnte natürlich einen Kreis um einen dieser Punkte schlagen, die in diesem Kreis enthaltenen weiteren markierten Punkte zählen und dann im anderen Bild versuchen, Pixel mit ähnlichen Markierungsdichten zu finden, dann die Abstände der Punkte untereinander zu bestimmen und ... der Leser bemerkt schon, dass selbst ein Computer eine Kopfschmerztablette benötigt, um das alles durch zu probieren, wobei noch gar nicht berücksichtigt ist, dass eines der Bilder anders skaliert, gedreht oder sonst wie transformiert ist. Die erste Aufgabe, solche Punkte zu finden, muss also um eine weitere, die Punkte so zu beschreiben, dass man sie woanders wiederfindet, ergänzt werden. Doch alles der Reihe nach.

So weit also der Plan. Ob und wie weit das gelingt, ist noch zu untersuchen. Die Vorgehensweise – von Pixeleigenschaften ausgehen und schrittweise zu abstrakten Begriffen zu kondensieren – entspricht auch der Arbeitsweise der Netzhaut und des Gehirns. Da die bekannterweise funktioniert, hat unser Plan gewisse Erfolgsaussichten.

## Einfache Filter

Wie sehen nun diese Filter aus, die zu den obigen Markierungen führen? Diejenigen aus Teil 3 (Gesichtserkennung) können wir nicht übernehmen, weil sie Informationen aus größeren Bildbereichen extrahieren, hier aber ausdrücklich Eigenschaften einzelner Pixel angesprochen sind.

Auch wenn die Bilder oben sehr viele bunte Markierungen enthalten, sind es im Verhältnis zur Gesamtzahl der Pixel nur relativ wenige. Wir können/müssen davon ausgehen, dass eine solche Informationsreduktion nicht in einem Schritt gelingt, sondern eine Filter-Kaskade notwendig ist. Voraussichtlich muss jeder Filter einige Parametereinstellungen erlauben, um ein optimales Ergebnis zu erreichen. Und wir werden die Skalierung von Bildern berücksichtigen müssen: man kann nicht davon ausgehen, dass ein markiertes Pixel in einem Bild der Abmessungen 1000x500 für die Filter ohne weiteres die gleichen Eigenschaften besitzt wie ein ortsgleiches Pixel in einem Bild mit 6000x3000 Pixeln: wenn ein Filter auf einen Wertunterschied zwischen benachbarten Pixeln reagiert, fällt dieser im kleineren Bild ungleich stärker aus als im größeren gleichen Inhalts.



Die für die Versuche benötigten verschiedenen Versionen eines Bildes können im Testprogramm mit OpenCV-Funktionen erzeugt werden. Die wesentlichen sind

```
dest = cv.resize(img, (cols, rows))           # Größenänderung

rows, cols = img.shape                       # Verschiebung
M = np.float32([[1, 0, 100], [0, 1, 50]])
dst = cv.warpAffine(img, M, (cols, rows))

M = cv.getRotationMatrix2D(((cols-1)/2.0, (rows-1)/2.0), 90, 1)
dst = cv.warpAffine(img, M, (cols, rows))     # Drehung
```

Es gibt auch weitere Transformationen, die wir aber vorläufig nicht berücksichtigen. Manipulationen am Farbbild kann man der Einfachheit halber auch mit einem Bildverarbeitungsprogramm vornehmen.

## Der Gauss-Filter

Eine häufige Störung in Bildern ist Rauschen: die Pixelwerte schwanken aus irgendwelchen Gründen und täuschen Strukturen vor, die tatsächlich gar nicht vorhanden sind. Dies kann sich bei der Suche nach Merkmalen störend auswirken: der Algorithmus findet ein markantes Merkmal, das jedoch in einem anderen Bild des gleichen Motivs nicht zu finden ist, weil die Region weniger gestört ist. Es bietet sich daher an, zunächst dieses Rauschen zu beseitigen. Neben Störungen werden dabei vielleicht auch einige tatsächliche, aber nur sehr schwach ausgeprägte Merkmale beseitigt, was aber auch in unserem Sinn ist: die voraussichtliche Anzahl von Merkmalspixeln ist ohnehin groß und eine Elimination von einigen schwach ausgeprägten macht die weitere Auswertung einfacher.

Zur Beseitigung von Rauschen ersetzt man jedes Pixel im Bild durch einen Mittelwert, der mit seinen nächsten Nachbarn erzeugt wird. Das Prinzip sei durch ein kleines Beispielprogramm verdeutlicht. Zunächst wird eine Filtermatrix erzeugt, die die Gewichte enthält, mit denen die Nachbarn in die Mittelwertbildung eingehen. Sodann wird ein entsprechend großer Bildbereich mit diesen Werten multipliziert und aus der Summe der Matrixelemente der neue Mittelwert berechnet:

```
Filtermatrix: [[0.25 0.5 0.25]
               [0.5 1. 0.5 ]
               [0.25 0.5 0.25]]

Bildbereich:  [[ 98  95 101]
               [104 120 107]
               [144 145 144]]

Faltung roh:  [[ 24.5  47.5  25.25]
               [ 52.   120.   53.5 ]
               [ 36.   72.5  36.  ]]

Summe Faltung: 467.25, Faktor: 4.0
Neues Pixel:   116 (statt 120)
```

**Programm:**

=====

```
gs = np.array(...)  
px = img[col-1:col+2, row-1:row+2]  
res[col, row] = int(np.sum(gs*px)/np.sum(gs))
```

Das wesentliche Merkmal dieser Faltungsmatrix, wie sie technisch genannt ist, ist die Symmetrie. Man kann sie als 3\*3-Matrix um das neu zu berechnende Pixel herum legen, aber auch andere Formate wie 5\*5 oder 7\*7 sind gebräuchlich. Die Werte werden oft mit Hilfe mathematischer Funktionen berechnet, auf die wir hier aber nicht näher eingehen müssen. OpenCV stellt hierfür eine einfache Funktion zur Verfügung:

```
gray_soft = cv.GaussianBlur(gray, (3, 3), 0)
```



Als Ergebnis erscheint ein unscharfes Bild, weil die Übergänge verwischt werden. Diese sind an Kanten natürlich besonders groß, weshalb die Kanten im Differenzbild hervortreten. Die alleine genügen allerdings noch nicht, um Bilder vergleichen zu können. Trotzdem verbessern wir zunächst die Möglichkeit, Kanten zu finden.

Hinweis: sollten Sie den Filter komplett programmieren wollen, um beispielsweise zu ermitteln, um welchen Faktor die optimierten Algorithmen schneller sind, achten Sie darauf, dass sie die neu berechneten Werte nicht in das Originalbild eintragen dürfen, da sonst die Werte der anderen Pixel verfälscht werden. Legen Sie daher eine Kopie des Bildes an.

## Sobel-Filter

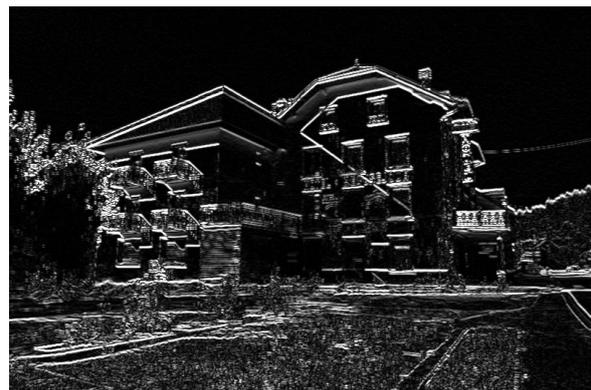
Anstelle der zuvor eingesetzten Filtermatrix betrachten wir die Matrix

Filtermatrix:  $\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$

In diese Neuberechnung geht der alte Wert des neu berechneten Pixels gar nicht ein, dafür eine Reihe von Nachbarn mit positiven, die andere mit negativen Faktoren. In einer einheitlichen Fläche kommt als Mittelwert Null heraus, aber an einer Kante mit stark unterschiedlichen Werten auf beiden Seiten des Pixels große positive oder negative Werte.

```
...  
[ 0 -3 -41 ... -14 -19 0]  
[ 0 -37 -70 ... -6 -5 0]  
[ 0 -56 -78 ... -2 0 0]
```

Überführt man diese in Absolutwerte, bekommt man ein besseres Kantenbild als mit dem Gaussfilter, allerdings mit der oben abgebildeten Matrix nur mit waagrechten Kanten. Für senkrechte Kanten muss man mit einer zweiten Matrix arbeiten, in der die 2. Spalte Nullen aufweist.



Mischt man beide Bilder, erhält man ein Bild, dass alle Kanten enthält.



```
grad_x = cv.Sobel(gray, ddepth, 1, 0, ksize=3)
```

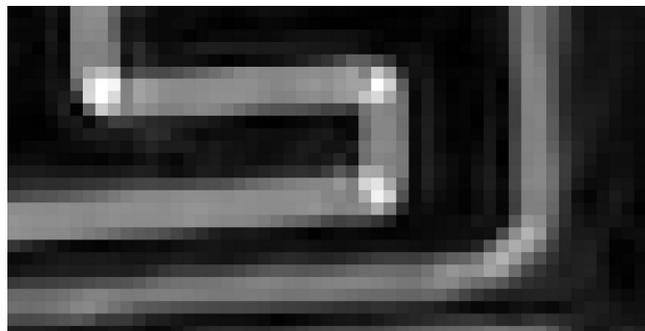
```
grad_y = cv.Sobel(gray, ddepth, 0, 1, ksize=3)
abs_grad_x = cv.convertScaleAbs(grad_x)
abs_grad_y = cv.convertScaleAbs(grad_y)
grad = cv.addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0)
```

## Komplexe Filter

Es existieren noch eine Reihe weiterer Faltungsmatrizen, mit denen man bestimmte Bildeffekte realisieren kann. Aber auch, wenn es optisch ansprechend aussieht und auch schon Einiges an Informationen fortgefallen ist, ist festzustellen: die Kanten nützen recht wenig als einfache Bildcharakteristiken, da eine Kante aus vielen Pixeln besteht. Wenn wir allerdings die Ecken im Bild identifizieren können, sieht es besser aus. Der Buchstaben „L“ lässt sich anhand seiner 6 Ecken recht gut charakterisieren.



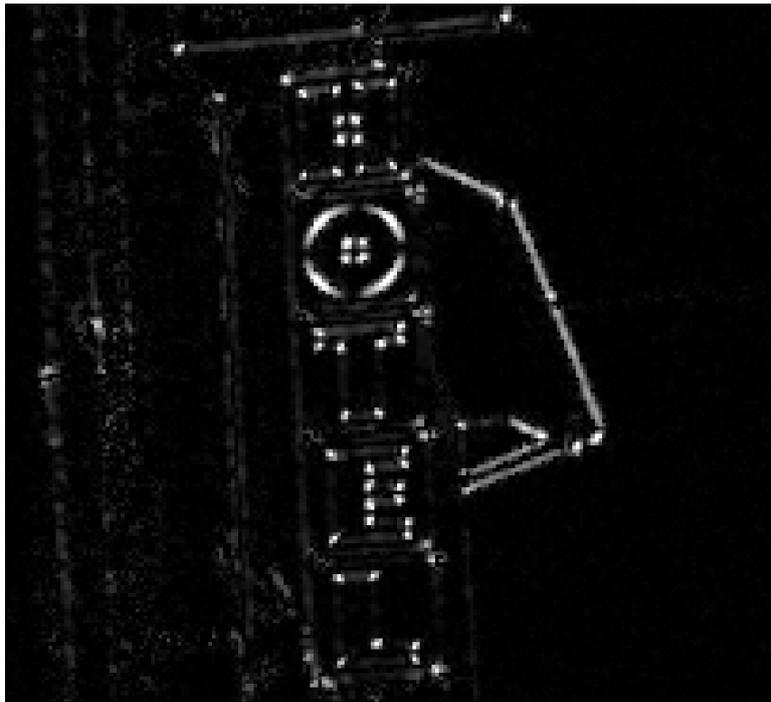
Allerdings wird die Filterung nun etwas schwieriger, was an einem Bildausschnitt klar wird.



Die weißen Pixel in der Mitte der Knicke sind die gesuchten Eckenpunkte, und würde man die Skalierung ändern, könnte die Rundung unten Rechts ebenfalls noch als Ecke durchgehen.

## Denkbare Filteransätze

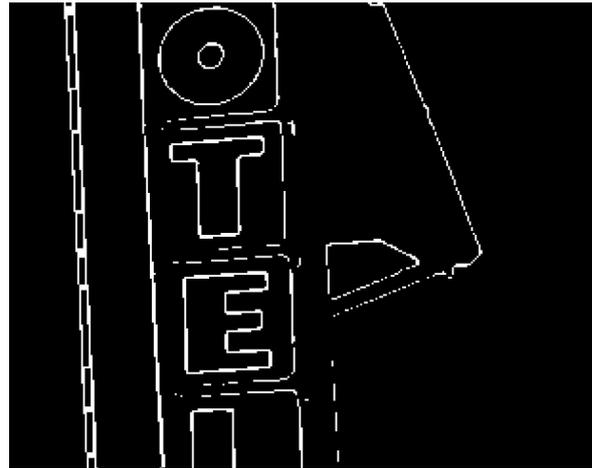
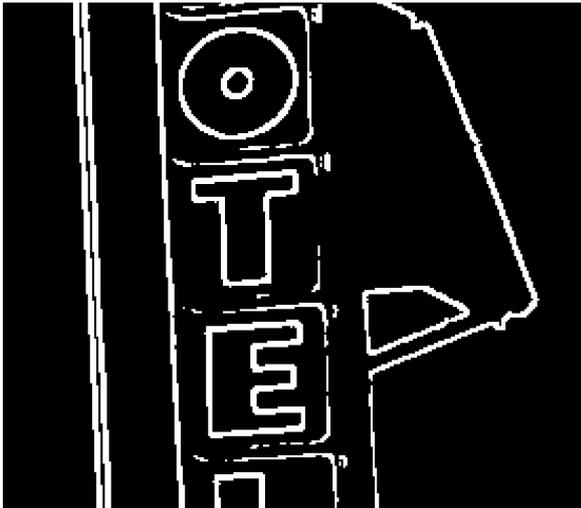
Kanten sind mathematisch gesprochen die Orte der größten Gradienten der Pixelwerte im Bild. Ecken sind die Punkte, in denen sich die Richtung der Gradienten abrupt ändert. Der Sobel-Filter ermittelt die Gradienten in X- und in Y-Richtung getrennt. Der X-Filter ist blind für Y-Gradienten, entsprechend der Y-Filter für X-Gradienten. Kombinieren wir die beiden Filterergebnisse durch eine UND-Operation, die nur dort Werte übrig lässt, wo beide Filter etwas gesehen haben, finden wir im Bereich des oben dargestellten Bildes



Die Ecken der Buchstaben kommen hier – abgesehen vom „O“ – schon recht deutlich zum Vorschein. Man könnte diese Punkte mit Hilfe einer Maske finden, die ähnlich wie die Gauss-Maske aufgebaut ist, aber die Umgebungspunkte mit Faktoren  $>1$  gewichtet. Ist der so bestimmte neue Wert größer als der alte, gibt es weitere Pixel  $\neq 0$  in der Umgebung und der Punkt ist keine isolierte Ecke. Es sei dem Leser überlassen, das auszuprobieren.

Allerdings ist eine Filterung in X- und Y-Richtung nicht geeignet, Ecken in Diagonalen zu finden. Für Diagonale lassen sich natürlich auch Gradientenmasken erzeugen und entsprechend bearbeiten. Wer hier üben möchte, hat also einige Möglichkeiten, der Fantasie freien Raum zu lassen.

Insgesamt hat die Erfahrung allerdings gezeigt, dass man auf diesem Weg keine allgemein befriedigenden Ergebnisse bei der Bildverarbeitung erhält. Eine andere Option besteht darin, das Kantenbild durch Wahl eines geeigneten Schwellwertes in ein Binärbild zu überführen (über den Schwellwert lässt sich einstellen, wie viele Punkte man letztlich finden kann) und die noch recht breiten Linien bis auf Pixelbreite 1 zu erodieren:

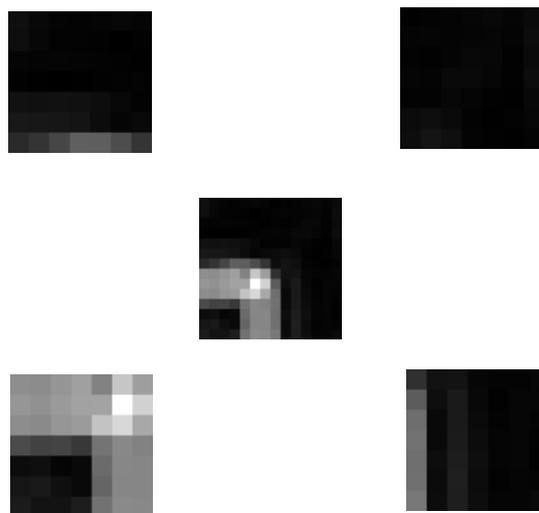


Passenden Algorithmen, die nicht zu viele Pixel entfernen (wie im rechten Beispiel), sind in OpenCV enthalten und basieren auf Masken, die ein Entfernen eines Pixels nur dann erlauben, wenn bestimmte andere Umgebungspixel belegt sind (wie solche Masken aussehen, sollte sich der Leser übungshalber einmal überlegen). Ein Eckpunkt zeichnet sich dann dadurch aus, dass er in einer 3\*3-Matrix der einzige Punkt  $\neq 0$  ist.

Solche Verfahren sind in der technischen Bildverarbeitung, in der es darauf ankommt, in wiederkehrenden einfachen Umgebungen bestimmte Sachverhalte zu erkennen, im Einsatz, da sie recht effizient sind. Für die allgemeine Bildverarbeitung sind sie aber nur bedingt geeignet.

## Harris-Ecken-Detektor

Für die Entwicklung eines effektiven Verfahrens zur Eckenfilterung kommt man nicht um eine gehörige Portion Mathematik herum, die ich natürlich keinen zumuten möchte. Das Grundprinzip lässt sich etwa so darstellen:



Um jeden Punkt wird ein größerer Bereich ausgewählt und in 4 Teilbereiche zerlegt. Summiert man die Pixelwerte der Teilbereiche auf, erhält man beispielsweise:

|        |       |      |      |                     |
|--------|-------|------|------|---------------------|
| [80,   | 1454, | 141, | 64]  | # Ecke am E         |
| [1252, | 496,  | 289, | 289] | # Bogen unter dem E |
| [72,   | 71,   | 760, | 657] | # Kante des E       |
| [94,   | 1123, | 154, | 756] | # Diagonale Kante   |
| [18,   | 27,   | 28,  | 5]   | # leeres Gebiet     |

Wie leicht festzustellen ist, zeichnen sich Ecken durch 3 schwach besetzte Felder aus, Kanten durch 2 und gleichmäßige Flächen durch 4 gleiche Felder. Durch Vorgabe der Verhältnisse lässt sich die Zahl der gefundenen Ecken optimieren.

Dies ist natürlich eine vereinfachte Darstellung, da nicht mit den Pixelwerten, sondern mit den verschiedenen Gradienten gearbeitet werden muss. Im Programmteil ist ein Beispiel vorhanden, dass der Sache deutlich näher kommt (Erläuterung siehe „Skaleninvariante ...“).

```
# compute first derivatives
dx = cv2.filter2D(image, ddepth=-1, kernel=Sx)
dy = cv2.filter2D(image, ddepth=-1, kernel=Sy)

# Gaussian Filter
A = cv2.filter2D(dx*dx, ddepth=-1, kernel=G)
B = cv2.filter2D(dy*dy, ddepth=-1, kernel=G)
C = cv2.filter2D(dx*dy, ddepth=-1, kernel=G)

# compute corner response at all pixels
return (A*B - (C*C)) - k*(A + B)*(A + B)
```

Die Wirkung der einzelnen Operationen sieht man sich am besten im Programm an.



Die optimierten Funktionen aus OpenCV liefern je nach Einstellung des Grenzwertes, ab wann ein gefundener Punkt in die Eckenzählung eingeht:



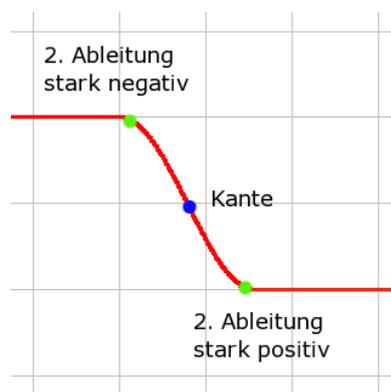
Ein anderer Ansatz setzt bei der Erklärung des Grundprinzips an und lässt eine KI (im Sinne von Teil 2 der Serie) entscheiden, ob ein Bildpunkt eine Ecke ist. Dazu zieht man einen Kreis um ein Pixel und ordnet alle Pixel innerhalb des Kreises in Form einer einfachen Liste an. Diese kann eine Größe bis zu 250 Pixeln aufweisen, was näherungsweise einem Kreis mit einem Radius ca 10 Pixeln entspricht, also in etwa dem, was in den obigen Beispielen verwendet wurde und die Erkennung einer Ecke erlaubt.

Aus einer großen Anzahl von unterschiedlichen Bildern wird nun jeweils eine größere Anzahl solcher Listen extrahiert und diese in die Kategorien Ecken und Nichtecken aufgeteilt. Anschließend wird ein neuronales Netz trainiert, zwischen den Kategorien zu unterscheiden. Da ein solches Netz, einmal trainiert, schnell und hinreichend sicher ist, lassen sich auch mit dieser Methode hinreichend viele Ecken in Bildern identifizieren.

## Skaleninvariante Merkmalsextraktion

Der Eckendetektor liefert zwar brauchbare Bildpunkte zum Vergleich von Bildern, beschränkt sich aber auf Ecken und kann in Bildern unterschiedlicher Größe zu unterschiedlichen Ergebnissen kommen. Beispielsweise kann ein Rundung in einem weniger hoch aufgelösten Bild als Ecke erscheinen.

Ein weiterer, nun allerdings mit einigem Rechenaufwand verbundener Ansatz versucht, in verschiedenen Auflösungsstufen identische charakteristische Bildpunkte zu finden. Für das Verständnis betrachten wir eine Kante genauer:



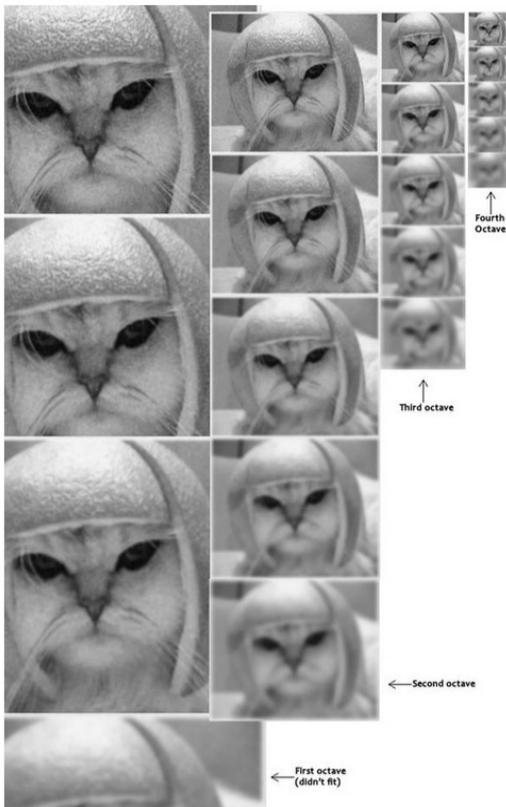
In Bildern sind Kanten nicht scharf, sondern umfassen mehrere Pixel. Insbesondere nimmt die Breite zu, wenn die Auflösung steigt oder das Bild geglättet wird. Der in allen Darstellungen gleiche und deshalb zu suchende charakteristische Kantenpunkt ist der Mittelpunkt der aufsteigenden Flanke. Mathematisch formal ist das der Punkt maximaler Steigung, den wir mit den zuvor geschilderten Mitteln versucht haben zu bestimmen, was aber eben nicht skaleninvariant ist.

Wenn ein Bild mit verschiedenen Skalierungen vorliegt, kann in jeder Version nach diesem Punkt gesucht werden. Er ist dadurch nicht nur genauer bestimmbar, er sollte in einer dieser Versionen auch einen Extremwert besitzen. Ist das nicht der Fall, besteht die Gefahr, einem Artefakt aufzusitzen, d.h. der Punkt ist gar nicht so charakteristisch, wie er sein sollte. Im Fall einer Kante, an der nur ein solcher Gradient vorliegt, tritt so etwas nicht ein, bei einem Eckpunkt, an dem mehrere Gradienten aufeinander treffen, ist das aber schon denkbar.

Um nach solchen Extrema zu suchen, werden verschiedene Bilder aus einem Ausgangsbild generiert:

- (a) Die Abmessungen des Originalbildes werden mehrfach halbiert.
- (b) Jedes dieser Bilder wird verschiedenen Gaussglättungen unterworfen

Auf diese Weise entstehen mehrere Sätze unterschiedlich scharfer Bilder. Betrachtet man dieses Sätze als Stapel, wird jeder Bildpunkt durch eine Funktion  $L(x, y, \sigma)$  beschrieben, wobei  $x, y$  die Koordinaten sind und  $\sigma$  die Ebene angibt ( $\sigma$  definiert, welche Werte in der Gauss-Filtermatrix stehen). Die Funktion heißt auch Skalenraum,



Nach den oben angestellten Überlegungen besitzt das scharfe Bild mit der geringsten Auflösung die schärfste Kante, die durch Gaussglättung und größere Auflösung systematisch verbreitert wird. Die Kombination von Maßstabänderung und Glättung sollte einen annähernd gleichmäßigen Verlauf der Verbreiterung über den gesamten Bilderstapel liefern, dessen Ebenen wir durchnummern können.

Gesucht wird nun das Maximum der Gradienten, d.h.

$$D_{k, \max}(x, y, \sigma) \approx L(x, y, (k+1) * \sigma) - L(x, y, k * \sigma)$$

Dazu wird von jeweils benachbarten Bildern wird die Differenz gebildet:



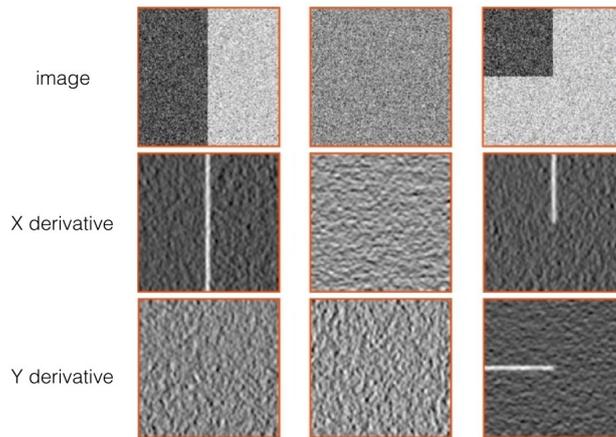
Die Bilderserie wurde für Demonstrationszwecke aufbereitet, um ein wenig Anschaulichkeit zu erzeugen. Tatsächlich beinhalten die Differenzbilder natürlich positive und negative Werte, die mit aus der mathematischen Theorie folgenden Faktoren zu versehen sind, um in der beabsichtigten Weise weiterverarbeitet werden zu können.

Bei kleiner werdender Auflösung verschwindet auch das eine oder andere Bilddetail.

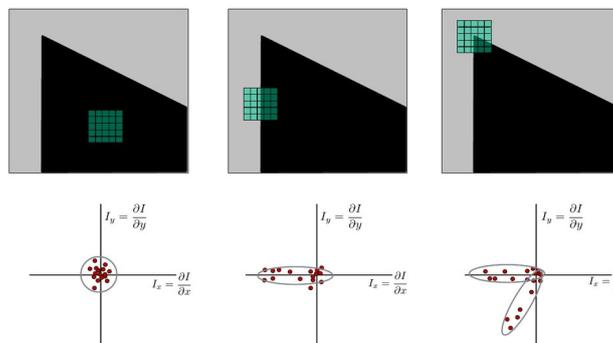


Das Finden lokaler Maxima und Minima in diesen Bildern ist reine Rechenarbeit: ein Bildpunkt ist dann ein Kandidat für einen charakteristischen Punkt, wenn er der größte oder kleinste Wert in seiner direkten Umgebung von 8 Pixeln sowie den jeweils 9 Pixeln der darüber und darunter liegenden Schärfeebene ist (es können daher nur bestimmte Ebenen hierfür heran gezogen werden). Die genaue Lage des Punktes im Sub-Pixelbereich wird mathematisch aus den Pixelwerten der verschiedenen Bildgrößen abgeleitet. Ausgeschlossen werden Pixel mit zu geringen Kontrastwerten zur Umgebung.

In der Liste dieser Punkte sind allerdings i.d.R. noch Kantenpunkte vorhanden. Bei einem Vergleich zweier Bilder können diese Punkte an unterschiedlichen Stellen identifiziert werden und sind deshalb nicht brauchbar und müssen ausgeschlossen werden. Wie wir von der Kantendetektion wissen, sind nur Punkte zu berücksichtigen, an denen ein abrupter Wechsel der Gradientenrichtungen stattfindet:



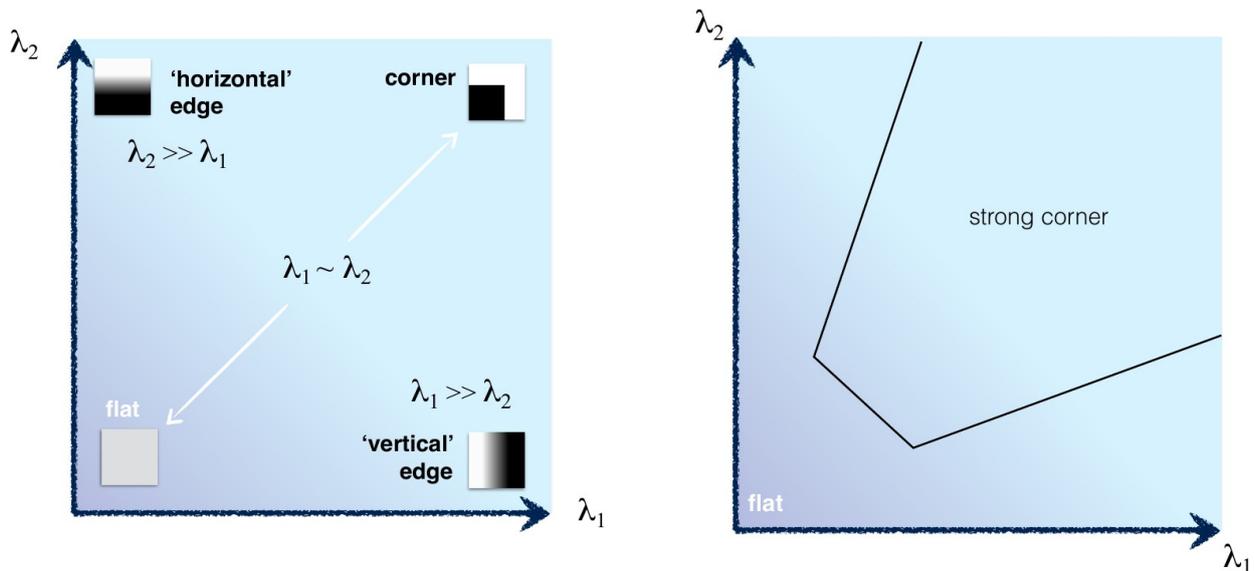
Das funktioniert gut, wenn die Kanten in X- oder in Y-Richtung verlaufen, aber nicht, wenn sie irgendwie durch den Raum verlaufen und die Ecke einen spitzen oder stumpfen Winkel besitzt, weil dann beide Gradienten  $\neq 0$  sind. Das Problem wird mathematisch gelöst, indem um den Punkt herum die getrennt erzeugten Gradientenfelder  $D_x, D_y$



zu einer Kovarianzmatrix gemischt werden:

$$A = \begin{vmatrix} \sum_P D_x D_x & \sum_P D_x D_y \\ \sum_P D_x D_y & \sum_P D_y D_y \end{vmatrix}$$

Mathematisch besitzt eine solche Matrix zwei so genannte Eigenwerte, wobei wir hier aber nicht näher darauf eingehen, was das ist und wie sie berechnet werden. Letztlich fallen aus der Rechnung zwei Zahlen  $\lambda_1, \lambda_2$  heraus, die klein sind, wenn der Punkt ein Flächenpunkt ist, von denen nur einer groß ist, wenn es sich um einen Kantenpunkt handelt und die beide im rechten Fall des obigen Bildes groß sind, wenn die Umgebung durch zwei Gradientenellipsen beschrieben wird.



Man kann so einen Bereich festlegen, in dem man von einem zuverlässig bestimmten Eckpunkt ausgehen kann, wobei die Orientierung der Kanten beliebig ist, als auch bei Drehungen des Bildes exakt wiedererkannt wird. Das Verfahren wird auch beim Harris-Eckendetektor angewandt und ist in den Rückgabewert eingearbeitet, dessen Erklärung wir hiermit nachgereicht haben.

Damit haben wir charakteristische Punkte im Bild gefunden, die allerdings als Fließkommawerte in der Form

```
(< cv2.KeyPoint 0x7f0565358630>, < cv2.KeyPoint 0x7f05653585a0>,
 < cv2.KeyPoint 0x7f05653585d0>, < cv2.KeyPoint 0x7f0565358600>, ...
```

ausgeliefert werden, da ihre Orte auf den Bruchteil eines Pixels genau berechnet wurden.

Was noch fehlt, ist eine Darstellung des Ergebnisses der Suche. Die verschieben wir auf das nächste Kapitel, weil dabei gleich noch weitere Informationen dargestellt werden können. Es sei vorab nur so viel gesagt: wer meint, dies sei nur eine genauere Eckenbestimmung, der irrt. Der skaleninvariante Algorithmus identifiziert überwiegend ganz andere Punkte, mit denen man im Primärbild beim Anschauen meist sehr wenig verbindet.

## Deskriptoren

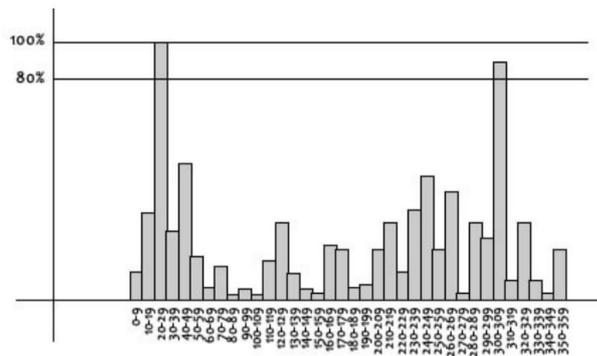
### Erzeugen

Jetzt verfügen wir zwar über charakteristische Punkte im Bild, aber wie finden wir die in anderen Bildern wieder, insbesondere auch dann, wenn die Bilder irgendwelchen Transformationen unterworfen sind?

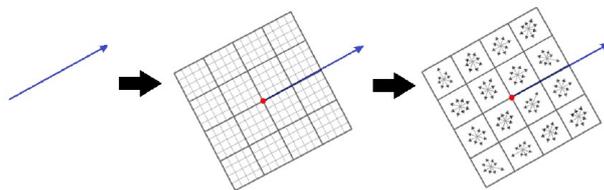
Zusätzliche Informationen müssen aus der Umgebung der gefundenen Bildpunkte generiert werden, wobei es sich allerdings um Daten handeln muss, die unabhängig von Bildanpassungen sind. Pixel-

werte kommen daher nicht in Frage, aber Gradienten sind unabhängig von den meisten Manipulationen.

Zunächst wird die Orientierung der Schlüsselpunkte ermittelt, d.h. die Richtung des größten Gradienten. Dazu legt man bildlich gesprochen eine geschnittene Torte auf den Schlüsselpunkt und ermittelt die Gradienten, die sich aus den von den „Tortenstücken“ bedeckten Pixel errechnen lassen. Die Größe hängt von der Skalierung ab. Die Orientierung ist die Richtung des größten Gradienten:



Um die dadurch definierte Achse wird ein 16\*16 Pixel großes Feld gelegt, das wiederum in 16 Felder der Größe 4\*4 zerlegt wird. Für jedes Feld werden 8 Richtungsgradienten berechnet.



Insgesamt erhält man so 128 Werte, die einen „Fingerabdruck“ oder Descriptor für den Schlüsselpunkt darstellen.

```
[[ 0.  4.  3. ... 0.  9. 31.]
 [ 0.  0.  0. ... 0. 16. 15.]
 [ 7.  1.  0. ... 3.  0.  3.]
 ...
```

Lässt man sich die Rechenergebnisse für die skaleninvariante Auswertung grafisch darstellen, erhält man:



2000\*1333 Pixel



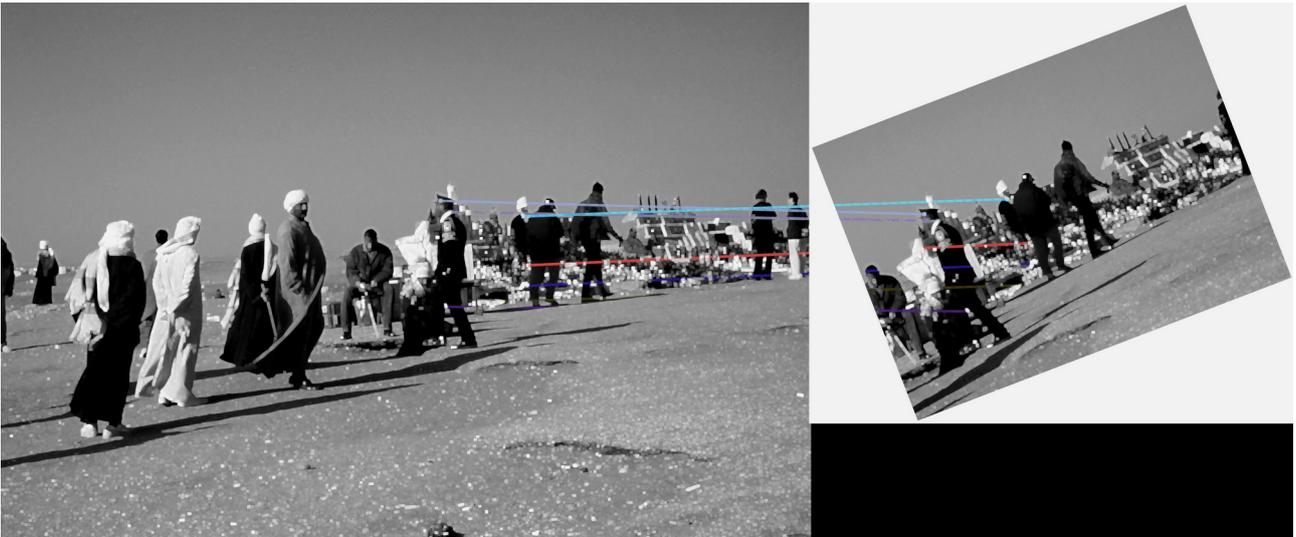
300\*200 Pixel

Die Darstellung enthält die Orientierung sowie den Bildbereich, den der Schlüsselpunkt abdeckt. Wie man unschwer erkennt, werden trotz des großen Skalenunterschiedes zwischen den Bildern (das kleiner würde man eher als „Thumbnail“ bezeichnen) die meisten Schlüsselpunkte in beiden Bildern gefunden. Die Abbildungen zeigen auch, was oben bereits erwähnt wurde: der skaleninvariante Algorithmus findet andere Bildpunkte als der Eckenalgorithmus. Von der Anschauung her wäre man sicher nicht auf die Idee gekommen, solche Punkte mitten auf einer Fläche zu plazieren.

## Bildvergleich

Für einen Vergleich erstellt man von den Bildern Listen mit Descriptoren und versucht, ähnliche Descriptoren in verschiedenen Bildern zu finden. Den Begriff „ähnlich“ kann man mathematisch unterschiedlich definieren. Eine der einfachsten Methoden besteht darin, die Descriptoren voneinander zu subtrahieren und dann abzuzählen, wie viele Positionen eine gegebene Schranke überschreiten. Für die Beispielbilder aus dem vorhergehenden Teil finden wir:





Die Zahl der gefundenen Übereinstimmungen ist hier auf 10 beschränkt, um die Darstellung übersichtlich zu halten.

Allerdings ist das noch keine Garantie, dass man tatsächlich die gleichen Bilder vor sich hat. Auch zwischen völlig verschiedenen Bildern ergeben sich natürlich Treffer:



Um letztlich eine Übereinstimmung nachzuweisen, berechnet man die Abstände zwischen den Schlüsselpunkten in den Bildern. In den beiden oberen Beispielen sind die Abstandsverhältnisse in allen Bildern gleich, im unteren Beispiel natürlich nicht.

Dieser letzte Teil ist im Beispielcode nicht mit eigenen Programmierungen hinterlegt, sondern greift komplett auf fertige OpenCV-Funktionen zurück. Soweit zum Verständnis hilfreich, sei der Leser selbst aufgefordert, weitere Versuche zu unternehmen.