CV/task1b — Coin Counting and Segmentation

1 Überblick

Ziel dieser Aufgabe ist es, in einem Bild Münzen zu segmentieren und anschließend zu klassifizieren. Um die Münzen im Bild zu erkennen wird ein Canny Kantendetektor sowie eine Hough Transformation verwendet. Die dadurch gefundenen Münzen werden anhand ihrer Größe klassifiziert.

(a) Eingabebild (b) Detektierte Münzen

Abbildung 1: Münzensegmentierung: Das linke Bild zeigt das Eingabebild mit nebeneinander liegenden Münzen. Im rechten Bild sieht man die segmentierten Münzen nach erfolgreichem Ausführen des Algorithmus.

Die Eingabedaten werden aus JSON-Dateien bereits automatisch eingelesen und können sofort zur Berechnung herangezogen werden. Der Inhalt dieser Aufgabe beschränkt sich einzig und allein auf den Münzensegmentierungs-Algorithmus, nicht auf etwaigen Aufwand der sich durch die Programmiersprache ergibt. Das Framework ist so aufgebaut, dass Sie die relevanten Funktionen implementieren müssen um die gewünschte Ausgabe zu erreichen.

Für die Kantendetektion werden 2 Algorithmen implementiert, der Canny-Algorithmus und die Hough Transformation. Beide Algorithmen im Zusammenspiel erlauben eine Segmentierung der Münzen. ***Wichtiger Hinweis***: Für die Lösung der Aufgabe dürfen Sie **nicht** die Funktionen `cv::Canny(...)` und `cv::HoughCircles(...)` bzw. `cv::HoughLines(...)` verwenden.

- **Canny Algorithmus** [1]

  Der Algorithmus ist in mehrere Schritte aufgeteilt.

  Im **ersten** Schritt wird das Eingabebild zu einem Graustufenbild konvertiert. Dabei wird das Bild von einem 3 Kanal RGB Bild in eines mit einem Kanal umgewandelt, in dem jeder Pixel einen Wert zwischen 0 und 255 annimmt. Abb. 2 zeigt das Originalbild und das konvertierte Bild.

(a) Eingabebild (b) Graustufenbild

Abbildung 2: Konvertierung: Das Eingabebild links wir zu einem Graustufenbild (rechts) umgewandelt.

  Im darauf folgenden **zweiten** Schritt wird ein Gaußfilter auf das Bild angewendet um etwaiges Rauschen aus dem Bild zu entfernen. Dies hilft beim weiteren Verarbeiten des Bildes und beugt dem Auftreten von Artefakten vor. Abb. 3 zeigt das Bild nach der Filterung.

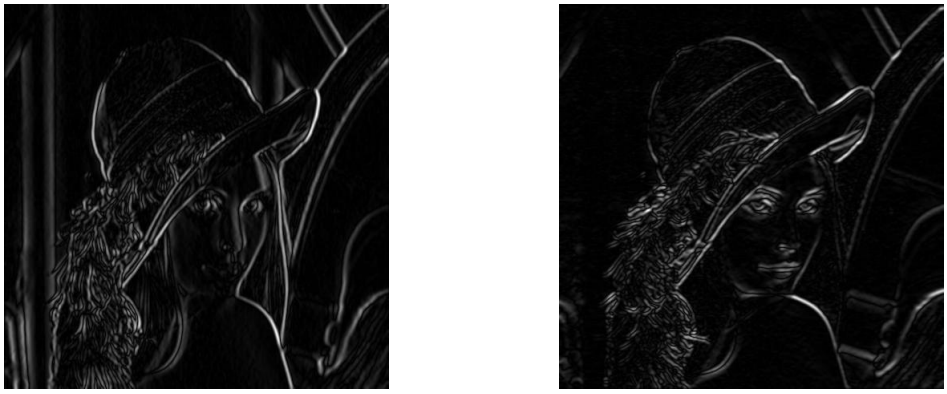
Abbildung 3: Gefiltertes Bild: Das Bild wird von Rauschen befreit, dadurch entstehen später weniger Artefakte.

  Im **dritten** Schritt wird ein Sobelfilter auf das Bild angewandt. Dadurch werden

die Gradienten G_x, G_y des Bildes in x- bzw. y-Richtung berechnet. Abb. 4 zeigt die Gradientenbilder. Aus den erstellten Gradientenbildern kann durch Anwendung von

$$|\,G\,| = \sqrt{G_x^2 + G_y^2} \tag{1}$$

das Gradient Magnitude Bild G erstellt werden. Abb. 5 zeigt das Ergebnis dieses Schrittes.



(a) Gradientenbild in x-Richtung (b) Gradientenbild in y-Richtung

Abbildung 4: Gradientenbilder

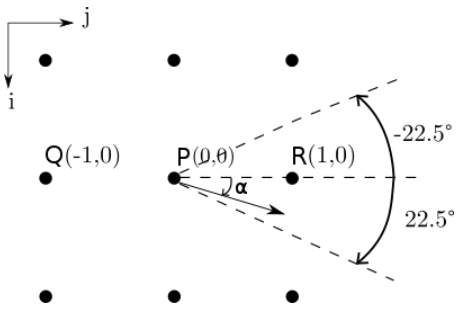
Abbildung 5: Gradient Magnitude Bild

Des Weiteren können aus den Gradientenbildern G_x, G_y die Gradientenrichungen α der einzelnen Pixel berechnet werden

$$\alpha = \angle G = \arctan\left(\frac{G_y}{G_x}\right) \tag{2}$$

Im **vierten** Schritt werden die lokalen Maxima durch eine Non-Maximum-Suppression gefunden. Ziel dieses Schrittes ist es entlang der Gradientenrichtung eines Pixels

P zu überprüfen, ob P ein lokales Maximum ist. Um dies zu erreichen wird die Gradientenrichtung α auf dem Pixels P herangezogen. Entlang dieser Richtung sollen die beiden Nachbarn R und Q gewählt werden (siehe Abb. 6a). Wenn der Wert des Punktes P kleiner ist als der Wert der Punkte Q oder R, wird dieser auf 0 gesetzt, da dieser nur ein Ausläufer des größten Wertes ist. Abb. 6b zeigt das Ausgabebild der Non-Maximum-Suppression.



(a) Klassifizierung der Gradientenwinkel α

(b) Ausgabebild der Non-Maxima-Suppression

Abbildung 6: Non-Maxima-Suppression

Der Canny-Algorithmus arbeitet mit zwei Thresholds (**Hysteresis Thresholding**). Es gibt einen oberen Threshold $\tau_{\max}$ und einen unteren Threshold $\tau_{\min}$. Alle Pixel die über Threshold $\tau_{\max}$ liegen werden als starke Kanten bezeichnet. Diese starken Kanten sind auf jeden Fall im Endergebnis enthalten. Alle Pixelwerte die unter den Threshold $\tau_{\min}$ fallen werden auf 0 gesetzt da diese sicher keine Kanten sind. Alle Pixel die zwischen $\tau_{\min}$ und $\tau_{\max}$ liegen werden als schwache Kanten bezeichnet. Diese schwachen Kanten werden nur dann als Kanten gezählt, wenn diese an eine starke Kante angrenzen. Ansonsten werden diese nicht als Kanten gezählt. Wurden alle Pixel richtig zugeordnet und der Algorithmus richtig angewandt, so entsteht dass in Abb. 7 gezeigte Bild, welches die Kanten des Eingabebildes darstellt.

- **Hough Circle Transformation**

  Dieser Algorithmus baut auf den, durch den Canny-Algorithmus, hervorgehobenen Kanten im Bild auf und kann parametrisierbare Objekte (z.B. Kreise, Geraden, etc.) erkennen. In diesem Schritt fokussieren wir uns auf die Detektion von Kreisen.

  Ein **Kreis** folgt der Kreisgleichung in der Formel (3). Wobei a und b den Kreismittelpunkt und r den Radius darstellen. Alle Punkte (x,y), die diese Gleichung erfüllen, liegen am Kreis mit Mittelpunkt (a,b) und Radius r.

$$r^2 = (x-a)^2 + (y-b)^2 \tag{3}$$

Abbildung 7: Endergebnis

Ziel der Aufgabe ist es Kreise (Münzen) im Bild zu erkennen. Hierfür wird ein 3 dimensionaler **Parameterraum** (auch Hough-Raum genannt) erschaffen, welcher für jeden Pixel im Bild, der auf einer Kante liegt, alle möglichen Kreise testet und deren Kreispixel im Parameterraum einträgt. Die Anzahl an möglichen Kreisen wird hierbei durch einen fix vorgebenden Minimal- und Maximialradius beschränkt. Im Parameterraum stellt jeder Punkt einen möglichen Kreis dar. Kommt so ein möglicher Kreis x-mal vor, so widerspiegelt dies dass x Kreise ausgehend von einem Kantenpunkt (wiederum mit den möglichen Radien) durch den Mittelpunkt dieses Kreises gehen.

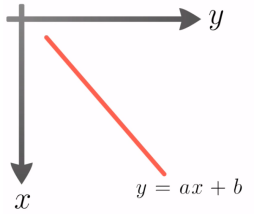
- **Hough Line Transformation**

  Diese funktioniert analog zur Circle Transformation. Anstatt Kreise zu erkennen versuchen wir hier nun Linien zu erkennen, wodurch sich einige Unterschiede im Algorithmus ergeben.

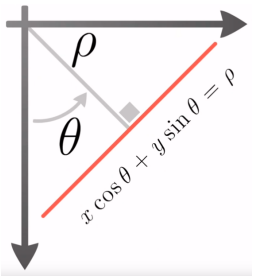
  Hierfür wird ein 2 dimensionaler **Parameterraum** erschaffen (auch Hough-Raum genannt), welcher für jeden Punkt (bzw. Pixel) im Bild, der auf einer Kante liegt, alle möglichen Parameter der Linie im Parameterraum einträgt. Somit stellt jeder Punkt im Parameterraum eine Linie dar. Kommt ein Punkt also x-mal vor, so widerspiegelt dies dass hier im Bild eine Linie auftritt die durch x Punkte geht.

  Wie erwähnt ist es die Aufgabe Linien im Bild zu erkennen. Eine **Linie** folgt der Geradengleichung $y = ax + b$. Wobei a die Steigung und b den Y-Achsenabschnitt darstellt (siehe Abb. 8a). Diese Funktion birgt jedoch das Problem dass zur Y-Achse parallele Geraden nicht dargestellt werden können. Die Lösung hierfür ist anstatt eine Linie mittels a und b zu definieren, diese mit dem Winkel θ (theta) und dem Radius ρ (rho) mit der Hesseschen Normalform darzustellen:

$$x \cdot cos(\theta) + y \cdot sin(\theta) = \rho \tag{4}$$



(a) Geradengleichung



(b) Gerade ausgedrückt durch die Hessesche Normalform.

Abbildung 8: Unterschiedliche Darstellungsformen einer Geraden.

Eine grafische Darstellung davon ist in Abb. 8b zu sehen. Unsere Aufgabe ist es die jeweils auftretenden θ und ρ Werte zu finden, welche mit hoher Wahrscheinlichkeit eine Linie darstellen (Hinweis: ρ kann auch negativ sein).

2 Aufgaben

Das Beispiel ist in mehrere Unteraufgaben gegliedert. Der Ablauf besteht aus den folgenden Schritten:

- Canny-Algorithmus:
  - Berechnen der Gradienten und der Gradientenrichtungen
  - Berechnen der Non-Maxima-Suppression
  - Berechnen des Hysteresis Thresholding
- Hough Circle Transformation:
  - Füllen der 3D Akkumulator Matrix
  - Berechnen der lokalen Maxima
  - Sortierung der lokalen Maxima
  - Berechnen der detektierten Linien aus den lokalen Maxima

Bonusaufgabe: Die Bonusaufgabe besteht darin die durch den Canny-Alorithmus und durch Hough Circle Transformation gefundenen Münzen zu klassifizieren. Dieser Teil der Aufgabe wird am besten durch die Hough Line Transformation gelöst. Es gibt keine automatische Überprüfung der Bonusaufgabe. Die Beurteilung erfolgt durch den Tutor beim Abgabegespräch. Dazu muss die Bonusaufgabe vorgezeigt werden und erklärt werden, wie diese gelöst wurde. Die gefundenen Münzen sollen anhand ihrer Größe klassifiziert werden. Dazu wird aus der JSON-Datei die Größe der verschiedenen Münzen und deren Wert eingelesen. Diese Vorgabe wird also durch das Framework zur Verfügung gestellt. Um einen Umrechnungsfaktor von Pixel auf Zentimeter berechnen zu können, wird auf jedem Bild ein rechteckiges Objekt mit bekannter Größe platziert welches mittels Hough Line Transform erkannt wird. Anhand dieses Referenzobjektes ist es möglich, einen Umrechnungsfaktor zu berechnen. Mit diesem Faktor kann dann die Größe der einzelnen gefundenen Münzen berechnet werden. Die berechnete Größe wird dann mit der tatsächlichen Größe der Münzen (die tatsächliche Größe wird aus der JSON-Datei eingelesen) verglichen. So entsteht eine Klassifizierung der einzelnen Münzen und der Wert der auf dem Eingabebild sich befindenden Münzen kann berechnet werden.

Diese Aufgabe ist mit Hilfe von OpenCV[1] 2.4.9 zu implementieren. Nutzen Sie die Funktionen, die Ihnen OpenCV zur Verfügung stellt (mit Ausnahme von `cv::Canny(...)`, `cv::HoughCircles(...)` und `cv::HoughLines(...)`) und achten Sie auf die unterschiedlichen Parameter und Bildtypen.

2.1 Berechnen der Gradienten und der Gradientenrichtungen (1 Punkt)

Dieser Teil der Aufgabenstellung ist in den Funktionen `cannyOwn(...)` und `calcAngles(...)` zu implementieren. Hierbei soll zunächst in der Funktion `cannyOwn(...)` ein Gradientenbild mit Hilfe der von OpenCV zur Verfügung gestellten Funktion der Sobel-Filterung erstellt werden. Der Sobel Filter detektiert gerichtete Änderungen im Bild (Gradienten) unter Zuhilfenahme von Ableitungen. Ein großer Gradientenwert eines Bildes deutet auf eine Änderung im Bild hin (z. B. Kanten, Farbübergänge). Für die Ableitung in x-Richtung G_x wird das Bild I mit dem Kern

$$S_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \tag{5}$$

[1] `http://opencv.org/`

gefaltet. Analog wird für die Ableitung in y-Richtung G_y das Bild I mit dem Kern

$$S_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{6}$$

gefaltet, sodass sich die Formeln

$$G_x = S_x * I, \quad G_y = S_y * I \tag{7}$$

ergeben. $*$ bezeichnet dabei die Faltungsoperation. Bestimmen Sie zuerst die jeweils *erste* Ableitung in den Richtungen x und y. Wenden Sie dazu den Sobel Filter (aus der OpenCV library) auf das Grauwertbild an. Wählen Sie für den Parameter, der die Bit-Tiefe des Ausgabebildes beschreibt, den Typ `CV_32F`.

Aus den beiden Gradientenbildern G_x und G_y kann mit Hilfe der Formel

$$|G| = \sqrt{G_x^2 + G_y^2} \tag{8}$$

das Gradient Magnitude Bild G berechnet werden. Dieses Bild wird dann für die weitere Verarbeitung verwendet.

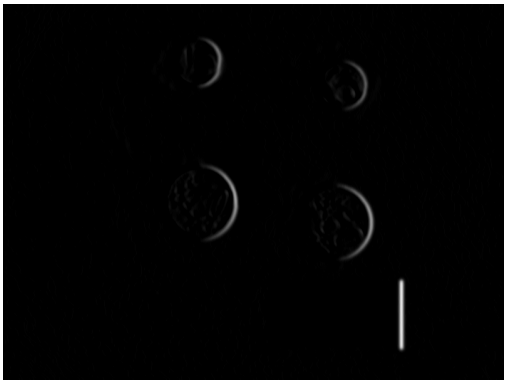
Da in einem weiteren Verarbeitungsschritt in die Orientierungen der Gradienten benötigt werden, werden diese in `calcAngles(...)` berechnet. Dazu werden wieder die bereits berechneten Gradientenbilder G_x und G_y verwendet. Um die Orientierung zu berechnen, wird der arctan verwendet

$$\alpha = \angle G = \arctan\left(\frac{G_y}{G_x}\right). \tag{9}$$

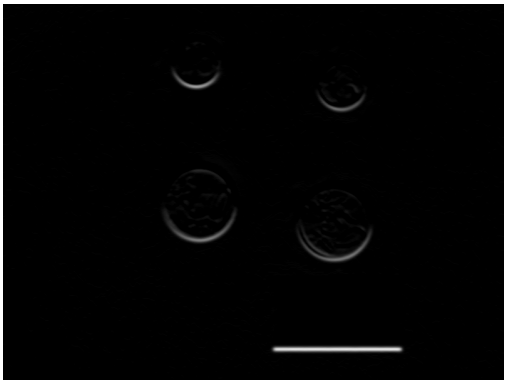
Abbildung 9 zeigt das Ergebnis der Gradienten Berechnung. Abbildung 10 zeigt das Gradient Magnitude Bild und die Orientierungen. *Hinweis:* Speichern Sie die Orientierungen in Winkelgrad und nicht in Bogenmaß. Verwenden Sie dafür `RAD2DEG`.

Hilfreiche OpenCV-Methoden:

- `cv::Sobel(...)`
- `cv::mult(...)`
- `cv::sqrt(...)`
- `std::atan2(...)`

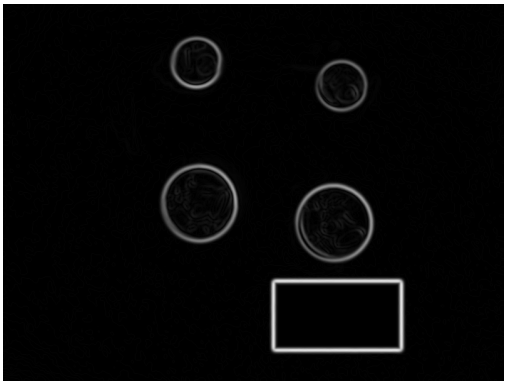


(a) Visualisierung der Gradientenmatrix in x-Richtung.

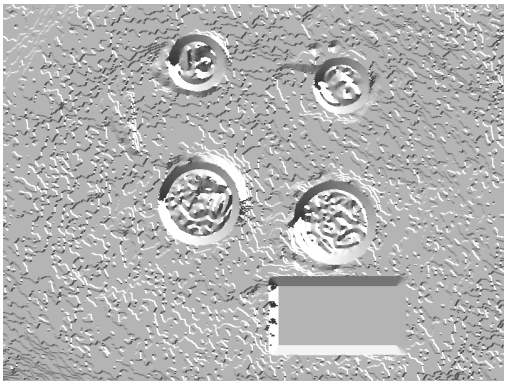


(b) Visualisierung der Gradientenmatrix in y-Richtung.

Abbildung 9: Ausgabe der Gradienten Berechnung.



(a) Visualisierung der Gradient Magnitude.



(b) Visualisierung der Gradienten Orientierungen.

Abbildung 10: Ausgabe der Gradienten Berechnung.

2.2 Berechnen der Non-Maxima-Suppression (1,5 Punkte)

Dieser Teil der Aufgabenstellung ist in der Funktion `nonMaximaSuppression(...)` zu implementieren. Hierbei sollen lokale Maxima gefunden werden. Dies führt dazu, dass im Ergebnis Bild die detektierten Kanten übrig bleiben.

Dafür wird für jeden Pixel P, an der Position (x, y), die bereits in Sektion 2.1 berechnete Richtung des Gradienten herangezogen. Zuerst werden alle Winkel die kleiner als 0° sind auf den korrespondierenden Winkel zwischen 0° und 180° gebracht.

$$\beta = \begin{cases} (\alpha + 360) \mod 180, & \text{if } \alpha < 0, \\ \alpha & \text{otherwise.} \end{cases} \tag{10}$$

Tabelle 1: Klassifizierung der Winkel

Klasse	**Winkel**	**Benachbarte Pixel in Relation zu P an der Position (x,y)**
I	$\beta <= 22.5$ **oder** $\beta > 157.5$	Q : gleiche Reihe (y), eine Spalte links ($x-1$)
		R : gleiche Reihe (y), eine Spalte rechts ($x+1$)
II	$22.5 < \beta <= 67.5$	Q : Reihe unterhalb ($y+1$), eine Spalte rechts ($x+1$)
		R : Reihe oberhalb (y - 1), eine Spalte links ($x-1$)
III	$67.5 < \beta <= 112.5$	Q : eine Reihe oberhalb ($y-1$), gleiche Spalte (x)
		R : eine Reihe unterhalb ($y+1$), gleiche Spalte (x)
IV	$112.5 < \beta <= 157.5$	Q : eine Reihe unterhalb ($y+1$), eine Spalte links ($x-1$)
		R : eine Reihe oberhalb ($y-1$), eine Spalte rechts ($x+1$)

Der berechnete Winkel β wird nun einer der 4 Klassen (horizontal, “diagonal-1”, vertikal, “diagonal-2”) anhand von Tabelle 1 zugeordnet (siehe auch Abb. 6a).

Abhängig von der Klasse des Winkels β, ergeben sich die Positionen der beiden Nachbarn Q und R (siehe Tabelle 1). Ist der Wert des Pixels Q oder R größer als der Wert des Pixels P so wird der Wert des Pixels P auf 0 gesetzt, da dieser nicht der größte Wert in der Gradientenrichtung ist.

$$\widehat{P} = \begin{cases} 0 & \text{if } (P < Q) \vee (P < R), \\ P & \text{otherwise.} \end{cases} \tag{11}$$

Das Ergebnis $\widehat{P}$ wird an der Position (x,y) ins Ausgabebild geschrieben. *Hinweis:* Da es an den Rändern zu out-of-bounds-reads kommen kann werden die äußeren Bereiche des Bildes nicht bearbeitet!

Abbildung 11 zeigt das Bild nach erfolgreicher Anwendung der Non-Maxima-Suppression.

Hilfreiche OpenCV-Methoden:

- `cv::copyTo(...)`

2.3 Berechnen der Hystere (0,5 Punkte)

Dieser Teil der Aufgabenstellung ist in der Funktion `hysteresis(...)` zu implementieren. In diesem Teil der Aufgabe werden die bereits gefundenen und verfeinerte Linien mit dem Thresholds verglichen. Dabei wird jeder Pixel P zugeordnet, ob dieser zu einer schwachen, einer starken oder zu keiner Kante gehört. Wenn der Wert des Pixels P größer ist als der

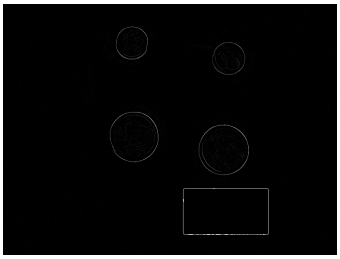


Abbildung 11: Bild nach Anwendung der Non-Maxima-Suppression.

obere Threshold $\tau_{\max}$ so ist dieser Teil einer starken Kante. Starke Kanten sind sicher Teil des Endergebnisses. Wenn der Wert kleiner ist als der untere Threshold $\tau_{\min}$ so ist dieser Pixel sicher nicht Teil einer Kante und daher auch kein Teil des Endergebnisses. Liegt der Wert des Pixels P zwischen den beiden Thresholds, so ist er Teil einer schwachen Kante.

$$P := \begin{cases} \text{starke Kante } (=255) & \text{if } P >= \tau_{\max} \\ \text{schwache Kante} & \text{if } \tau_{\min} < P < \tau_{\max} \\ \text{keine Kante } (=0) & \text{if } P < \tau_{\min} \end{cases} \tag{12}$$

Nachdem alle nicht-Kanten und starken Kanten im Bild gefunden und markiert wurden, werden schwache Kanten in Abhängigkeit ihrer Nachbarschaft iterativ bzw. rekursiv klassifiziert. Ist in der 8er Nachbarschaft eines Pixels P, welcher einer schwache Kante ist, mindestens ein Pixel der eine starke Kante bildet, so wird auch der Pixel P zu einer starken Kante (= 255). Dazu empfiehlt es sich alle bereits als starke Kanten klassifizierte Pixel zu speichern und danach deren 8er Nachbarschaft zu untersuchen ob sich in diesem Bereich eine schwache Kante befindet. Sollte dies der Fall sein, so wird die schwache Kante zu einer starken Kante und in dessen Nachbarschaft muss auch nach schwachen Kanten gesucht werden. Um dies zu erreichen kann ein iterativer Ansatz mit einem wachsenden Vektor, oder eine rekursiver Ansatz der für jeden starken Pixel eine Funktion zur Nachbarschaftsüberprüfung rekursiv aufruft gewählt werden. Am Ende dieser iterativen Schritte besteht das Ergebnis `thresh` nur aus Pixeln die entweder 0 (keine Kante) oder 255 (starke Kante) sind.

2.4 Füllen der 3D Akkumulator Matrix (2 Punkte)

Dieser Teil der Aufgabenstellung ist in der Funktion `HoughCirclesOwn(...)` zu implementieren.

Ziel dieses Schrittes ist es die **3D Akkumulator Matrix** (auch Voting Matrix genannt) zu befüllen. Diese 3D Matrix ist gespeichert als Vektor der Länge $\#rad$, gefüllt mit 2D

Akkumulator Matrix Einträgen und repräsentiert den quantisierten Parameterraum. Da die Kreisgleichung ($r^2 = (x-a)^2 + (y-b)^2$) drei Parameter hat (Radius r, und Kreismittelpunkt (a,b)), ist unsere Akkumulator Matrix drei-dimensional. Für unser Beispiel können wir annehmen, dass alle Radii der Münzen zwischen RAD_{MIN} und RAD_{MAX} fallen. Daher hat unser Vektor eine Länge von $\#rad = (RAD_{MAX} - RAD_{MIN})/rad\_quant$, wobei rad_quant die Quantisierung für den Parameter r ist. Des Weiteren nehmen wir an, dass der Kreismittelpunkt sich im Bild befinden muss. Daher haben unsere Matrizen eine Breite und Höhe von $img\_rows/spat\_quant$ bzw. $img\_cols/spat\_quant$, wobei $spat\_quant$ die Quantisierung für die Parameter a und b ist.

Um diese Matrix zu befüllen wird nun jedes Pixel P des Kantenbildes (Resultat vom Canny-Algorithmus) durchlaufen. Ist der Wert von P ungleich 0, so ist P ein Kandidat K für einen Kreis. Es werden für jeden K alle möglichen Kreise mit $\#rad$ Radii ausgetestet, und dabei deren Kreispunkte in der 3D Akkumulator Matrix eingetragen, daher der jeweils zugehörige Eintrag hochgezählt. Die 3D Akkumulator Matrix ist dabei ein Vektor mit der Länge von $\#rad$, gefüllt mit 2D Akkumulator Matrix Einträgen. Im Detail wird jeder Vektor Eintrag r durchlaufen und der jeweilige Kreisradius r_{circle} wie folgt berechnet:

$$r_{circle} = RAD_{MIN} + (r+1) \cdot rad\_quant \tag{13}$$

Dieser wird danach verwendet um die Kreispunkte zu berechnen. Zuerst werden die Eckpunkte einer Region of Interest RoI berechnet, welches um den derzeitigen Pixel K mit Koordinaten (x,y) aufgespannt wird (siehe Abb. 12 und Abb. 13).

$$\begin{aligned} y_{start} &= \lfloor \tfrac{y - r_{circle}}{spat\_quant} - 1 \rfloor \\ x_{start} &= \lfloor \tfrac{x - r_{circle}}{spat\_quant} - 1 \rfloor \\ y_{stop} &= \lceil \tfrac{y + r_{circle}}{spat\_quant} + 1 \rceil \\ x_{stop} &= \lceil \tfrac{x + r_{circle}}{spat\_quant} + 1 \rceil \end{aligned} \tag{14}$$

Danach iterieren wir über alle Pixel $P_{RoI} = (x_{RoI}, y_{RoI})$ in RoI und prüfen ob die Koordinaten valide sind (ob sie sich innerhalb der Akkumulator-Matrix befinden) und ob die Distanz $\widehat{d_{circle}}$ zum Ursprungspixel P dem aktuellen Radius r entspricht. Ist dies der Fall, so wird die Akkumulator Matrix an dieser Stelle (x_{RoI}, y_{RoI}) um eins erhöht. Schematisch ist dies in den Abbildungen 12 und 13 beschrieben. Formal berechnen wir dies folgendermaßen:

$$\begin{aligned} y_{coord} &= (y_{RoI} + 0.5) \cdot spat\_quant \\ x_{coord} &= (x_{RoI} + 0.5) \cdot spat\_quant \\ dy &= y_{coord} - y \\ dx &= x_{coord} - x \\ d_{circle} &= \sqrt{dy * dy + dx * dx} \\ \widehat{d_{circle}} &= \lfloor (d_{circle} - RAD_{MIN})/rad\_quant \rceil, \end{aligned} \tag{15}$$

wobei $\lfloor \cdot \rceil$ der Rundungsoperator ist. *Hinweis:* Die Variablen y_{coord}, x_{coord}, dy, dx, d_{circle} sollen als Datentyp `float` gespeichert werden. Die Variable $\widehat{d_{circle}}$ ist ein Integer.

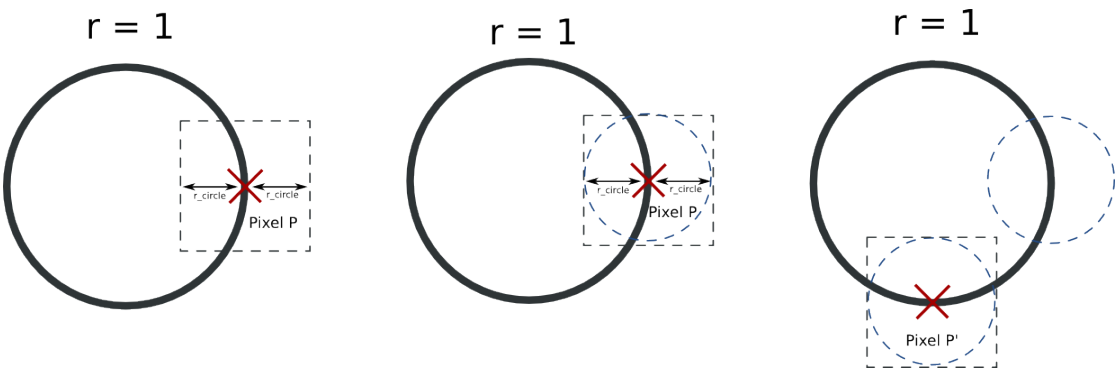


Abbildung 12: Schematische Erklärung des Algorithmus für radius = 1. Für jedes Pixel P das ungleich 0 ist wird eine r_{circle} Nachbarschaft betrachtet (Rechteck). Für jedes Pixel in diesem Rechteck, das den radius = 1 zum Pixel P hat, wird in die Akkumulator Matrix für $r = 1$ gevotet (blauer Kreis). Für ein weiteres Pixel P' wird dieser Vorgang ebenfalls wiederholt. In diesem Beispiel ergibt sich kein Schnittpunkt in der Akkumulator Matrix, daher wird hier später auch kein Kreis gefunden.

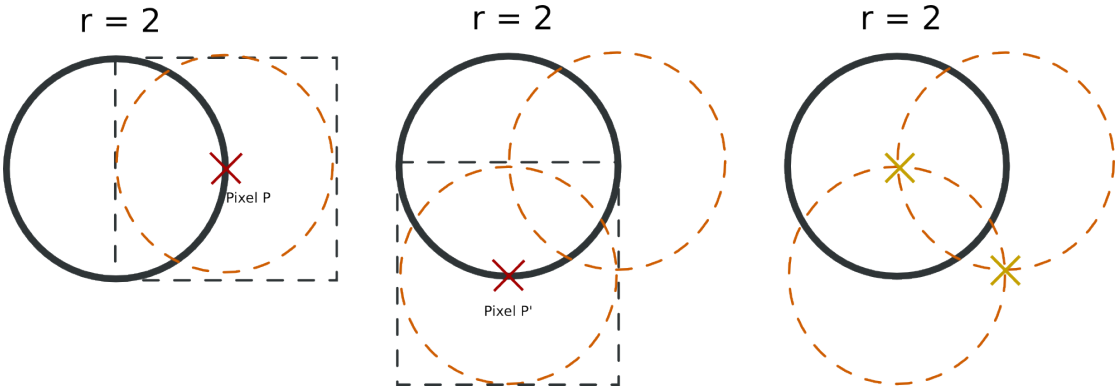


Abbildung 13: Schematische Erklärung des Algorithmus für radius = 2. Für jedes Pixel P das ungleich 0 ist wird eine r_{circle} Nachbarschaft betrachtet (Rechteck). Für jedes Pixel in diesem Rechteck, das den radius = 2 zum Pixel P hat wird in die Akkumulator Matrix für $r = 2$ gevotet (oranger Kreis). Für ein weiteres Pixel P' wird dieser Vorgang ebenfalls wiederholt. In diesem Beispiel ergeben sich zwei mögliche Schnittpunkte in der Akkumulator-Matrix (gelbe Markierungen). Diese Schnittpunkte repräsentieren gültige Kreismittelpunkte für die zwei am Kreis liegenden Punkte P und P', die vom Mittelpunkt radius = 2 entfernt sind. Wiederholt man diesen Vorgang für alle Pixel auf dem Kreis, so wird ein Schnittpunkt übrig bleiben.

Das Ergebnis einer befüllten Akkumulator Matrix wird in Abb. 14 gezeigt.

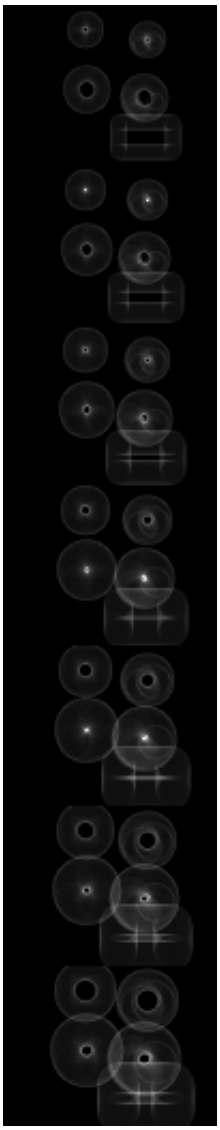


Abbildung 14: Die befüllte Akkumulator Matrix.

2.5 Berechnen der lokalen Maxima (2 Punkte)

Dieser Teil der Aufgabenstellung ist in der Funktion `HoughCirclesOwn(...)` zu implementieren.

In diesem Schritt durchlaufen wir jeden Wert A_{crt}/Index A_{crt_idx} der 3D Akkumulator

Matrix A mittels den Indices (r/y/x), die (Radius/Zeile/Spalte) entsprichen und berechnen uns dabei die lokalen Maxima. Ein lokales Maximum entspricht dem hellsten Pixel in der lokalen Nachbarschaft von A (siehe Abb. 14). Hierzu überprüfen wir zuerst ob A_{crt} größer als der Threshold ist:

$$A_{crt} > threshold \tag{16}$$

Sollte dies der Fall sein, prüfen wir die lokale 3D Nachbarschaft in einem $3 \times 3 \times 3$ Window in A, ob der gefundene Punkt auch ein lokales Maxima ist. Jeder Wert A_{crt} muss also die folgende Bedingung, laut 3D-Window, erfüllen um ein lokales Maximum zu sein:

$$\begin{aligned} &A(r,y,x) \geq A(r+dr, y+dy, x+dx), \\ &\forall dr, dy, dx \in \{-1,0,1\}^3, \end{aligned} \tag{17}$$

wobei $\{-1,0,1\}^3 = \{(-1,-1,-1),(-1,-1,0),(-1,-1,1),\ldots(1,1,1)\}$ das dreifache kartesische Produkt der Menge $\{-1,0,1\}$ ist. Hierbei wird A_{crt} mit den zugehörigen Nachbarwerten verglichen, und wenn A_{crt} größer als jeder Nachbarwert ist, so ist A_{crt} ein lokales Maximum.

Für jedes gefundene lokale Maxima merken wir uns den Wert in der Akkumulator Matrix A_{crt} und den dazugehörigen Index A_{crt_idx} indem wir diese in dem Vektor LM abspeichern. Jedes der abgespeicherten lokalen Maxima steht dabei für einen detektierten Kreis.

Hilfreiches vorgegebenes Struct:

- `CvLocalMaximum { r, x, y, accumulator_value }`

2.6 Sortierung der lokalen Maxima (1 Punkt)

Dieser Teil der Aufgabenstellung ist in der Funktion `HoughCirclesOwn(...)` zu implementieren.

Die abgespeicherten Werte in LM müssen nun absteigend nach deren Akkumulator Wert A_{crt} sortiert werden.

2.7 Berechnen der detektierten Kreise aus den lokalen Maxima (1 Punkt)

Dieser Teil der Aufgabenstellung ist in der Funktion `HoughCirclesOwn(...)` zu implementieren.

Am Schluss berechnen wir aus den vorhandenen Werten in LM die zugehörigen Kreise, die durch den Radius und den Kreismittelpunkt bestimmt sind (siehe Formel (3)). Dabei wird die Anzahl an Kreisen, welche wir zurückgeben, durch den Parameter 'int `circlesMax`' eingeschränkt. Wir durchlaufen maximal $circlesMax$ Werte von LM, wobei lm das derzeitige lokale Maximum in LM ist und berechnen uns die Kreise im originalen Koordinatensystem aus den lokalen Maximas im quantisierten Koordinatensystem mit folgender Formel:

$$\begin{aligned} r &= lm_r \\ x &= lm_x \\ y &= lm_y \\ center_x &= (x+0.5)\cdot spat\_quant \\ center_y &= (y+0.5)\cdot spat\_quant \\ radius &= (r+0.5)\cdot rad\_quant + RAD_{MIN} \end{aligned} \tag{18}$$

Die Ergebnisse sollen in den Vektor `circles` gespeichert werden, wobei $center_x$ der erste, $center_y$ der zweite und $radius$ der dritte Eintrag eines `cv::Vec3f` Elements ist.

3 Bonusaufgaben (3 Punkte)

Für die Bonusaufgabe muss zuerst die Funktion `HoughLinesOwn(...)` implementiert werden um Linien im Bild erkennen zu können, welche danach zum Klassifizieren der jeweiligen Münzen in der Funktion `classifyCircles(...)` zu verwenden ist. Hier ist Eigeninitiative gefragt um das Rechteck des Maßstabs aus den erkannten Linien herauszurechnen, um somit die Größe der Münzen im Bild bestimmen zu können.

3.1 Vorberechnen der benötigten Werte für Hough Transform

Dieser Teil der Aufgabenstellung ist in der Funktion `HoughLinesOwn(...)` zu implementieren.

Zuerst berechnen wir die Anzahl an Winkeln $\#\theta$ und die Anzahl an Radien $\#\rho$. Dabei verwenden wir folgende Notation und Parameter:

θ_q : Eingabeparameter 'float theta' der HoughLinesOwn-Funktion, der angibt wie stark der Winkelbereich quantisiert werden soll.

ρ_q : Eingabeparameter 'float rho' der HoughLinesOwn-Funktion, der angibt wie stark der Distanzbereich quantisiert werden soll.

img : Eingabebild

$\lfloor x \rceil$: runde x zum nächsten Integer Wert

$$\#\theta = \left\lfloor \frac{\pi}{\theta_q} \right\rceil \#\rho = \left\lfloor \frac{(img_{width} + img_{height}) \cdot 2 + 1}{\rho_q} \right\rceil \tag{19}$$

Um möglichst effektiv pro Pixel, welcher auf einer Kante liegt, alle θ Werte zu durchlaufen, berechnen wir im Vorhinein alle zugehörigen $cos(\theta)$ und $sin(\theta)$, und multiplizieren diese bereits vorab mit $1/\rho_q$ um sie schlussendlich in die jeweiligen Arrays θ_{cos} und θ_{sin} zu speichern. Dies erspart uns später, je nach Anzahl an Linien, eine große Anzahl an wiederholten Berechnungen. Formal schauen die Einträge folgendermaßen aus:

$$\theta_{sin_i} = sin(\theta_q \cdot i) \cdot \frac{1}{\rho_q} \tag{20}$$

$$\theta_{cos_i} = cos(\theta_q \cdot i) \cdot \frac{1}{\rho_q} \tag{21}$$

$$0 \leq i < \#\theta. \tag{22}$$

Hilfreiche OpenCV-Methoden:

- `cvRound(...)`

3.2 Füllen der 2D Akkumulator Matrix

Dieser Teil der Aufgabenstellung ist in der Funktion `HoughLinesOwn(...)` zu implementieren.

In diesem Teil der Aufgabe wird jedes Pixel P des Kantenbildes (Resultat vom Canny-Algorithmus) durchlaufen. Ist der Wert von P ungleich 0, so ist P ein Kandidat K für eine Linie. Somit werden für jeden K alle $\#\theta$ Winkel ausgetestet, um alle möglichen Linien die durch diesen Punkt (mit den gewählten Parametern) laufen zu beachten, und in einer 2D **Akkumulator Matrix** (auch Voting Matrix genannt), der jeweils zugehörige Eintrag hochgezählt. Angenommen wir wollen die Akkumulator Einträge für den i-ten Winkel berechnen. Der jeweils zugehörige Index für die Voting Matrix wird mit folgender Formel berechnet:

$$r = \lfloor (x \cdot \theta_{cos_i} + y \cdot \theta_{sin_i}) + d \rceil. \tag{23}$$

Wobei x und y die X- und die Y-Koordinate des Pixels im Bild darstellen. d wiederum ist die Diagonale des Bildes und wird verwendet um die ρ-Werte alle in den positiven Bereich zu shiften und somit den Arrayzugriff zu erleichtern. d wird mittels

$$d = (\#\rho - 1)/2 \tag{24}$$

berechnet.

Das Ergebnis einer befüllten Akkumulator Matrix wird in Abb. 15 gezeigt.

Abbildung 15: Visualisierung der befüllten 2D Akkumulator-Matrix.

3.3 Berechnen der lokalen Maxima

Dieser Teil der Aufgabenstellung ist in der Funktion `HoughLinesOwn(...)` zu implementieren.

In diesem Schritt durchlaufen wir jeden Wert A_{crt} der Akkumulator Matrix A mittels den Indices (r/n) was (Zeile/Spalte) entspricht und berechnen uns dabei die lokalen Maxima. Die Höhe von A ist $\#\rho$ und die Breite $\#\theta$. Ein lokales Maximum entspricht dem hellsten Pixel in der lokalen Nachbarschaft von A (siehe Abb. 15). Hierzu durchlaufen wir A mit einem 'Cross-Window' (siehe Abb. 16) und suchen uns somit die lokalen Maxima im Ergebnisraum. Jeder Wert A_{crt} (entspricht A(r+1,n+1)) muss also die folgende Bedingung,

laut 'Cross-Window', erfüllen um ein lokales Maxima zu sein.

$$\begin{aligned} &A(r+1,n+1) > A(r+1,n) \wedge A(r+1,n+1) > A(r+1,n+2) \\ &\wedge A(r+1,n+1) > A(r,n+1) \wedge A(r+1,n+1) > A(r+2,n+1) \end{aligned} \tag{25}$$

Hierbei wird A_{crt} mit den zugehörigen Nachbarwerten verglichen.

Zusätzlich gilt ein Wert erst als lokales Maxima, sobald dieser größer dem gewählten Threshold ist.

$$A(r+1,n+1) > threshold \tag{26}$$

Für jedes noch relevante lokale Maxima (oberhalb Threshold) merken wir uns den Wert in der Akkumulator Matrix A_{crt} und den dazugehörigen Index A_{crt_idx} indem wir diese in dem Vektor LM abspeichern. Jedes der abgespeicherten lokalen Maxima steht dabei für eine erkannte Linie.

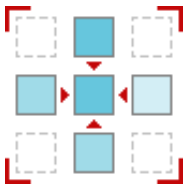


Abbildung 16: Cross-Window

Hilfreiches vorgegebenes Struct:

- `CvLocalMaximum { r, x, y, accumulator_value }`

3.4 Sortierung der lokalen Maxima

Dieser Teil der Aufgabenstellung ist in der Funktion `HoughLinesOwn(...)` zu implementieren.

Die abgespeicherten Werte in LM müssen nun absteigend nach deren Akkumulator Wert A_{crt} sortiert werden.

3.5 Berechnen der detektierten Linien aus den lokalen Maxima

Dieser Teil der Aufgabenstellung ist in der Funktion `HoughLinesOwn(...)` zu implementieren.

Am Schluss berechnen wir aus den vorhandenen Werten in LM die zugehörigen Linien in der hessischen Normalform (siehe Formel (4)). Dabei wird die Anzahl an Linien, welche wir zurückgeben eingeschränkt durch den Parameter '`int linesMax`'. Wir durchlaufen maximal $linesMax$ Werte von LM und berechnen uns die Linien mit folgender Formel:

$$\begin{aligned} line_\rho &= (lm_r - \tfrac{\#\rho - 1}{2}) \cdot \rho_q \\ line_\theta &= lm_n \cdot \theta_q, \end{aligned} \tag{27}$$

wobei $line_\rho$ das erste Element und $line_\theta$ das zweite Element des `cv::Vec2f` sind und lm der aktuelle Eintrag im LM Vektor ist.

Hilfreiche OpenCV-Methoden:

- `cvFloor(...)`

3.6 Kategorisierung der Kreise

Diese Funktion ist in *classifyCircles(...)* zu implementieren. Diese Funktion übernimmt die detektierten Kreise, detektierten Linien und einen Vektor, der die Eigenschaften der Münzen, die zu detektieren sind (Durchmesser in mm (diameter) und Wert (value)) speichert. Ziel der Funktion ist es alle Münzen im übergebenen Vektor zu klassifizieren und deren Wert (value) zu akkumulieren und zurückzugeben.

Um dies zu erreichen kann angenommen werden, dass die längste Seite des Rechtecks 40mm lang ist. Dadurch können die gemessenen Pixel-Durchmesser der Münzen normalisiert werden und diese dann Kategorisiert werden.

Hier ist Eigeninitiative gefragt um das Rechteck des Maßstabs aus den erkannten Linien herauszurechnen, um somit die Größe der Münzen im Bild bestimmen zu können.

4 Ein- und Ausgabeparameter

Folgende Parameter sind in den Konfigurationsdateien angegeben:

- Eingabe - Ausgewählter Bild-Datensatz (e.g. coins): `data_selected`
- Eingabe - Pfad zu den Bild-Datensätzen: `data_path`
- Ausgabe - Pfad zum Ausgabe-Ordner: `out_directory`
- Ausgabe - Dateiendung: `out_filetype`
- Ausgabe - Voranstehende Nullen bei den Filenamen: `out_filename_number_zero_filled`
- Ausgabe - Anzahl an Stellen der Zahlen (gefüllt mit Nullen) bei den Filenamen: `out_filename_number_width`
- Ausgabe - Name der auszugebenden Bilder: `out_filenames`

5 Programmgerüst

Die folgende Funktionalität ist in dem vom ICG zur Verfügung gestellten Programmgerüst bereits implementiert und muss von Ihnen nicht selbst programmiert werden:

- Die Konfigurationsdatei (JSON) wird vom Programmgerüst gelesen.
- Lesen des Eingabebildes und der Eingabeparameter
- Iteratives Ausführen der einzelnen Funktionen
- Schreiben der Ausgabebilder in die dafür vorgesehenen Ordner

6 Abgabe

Die Aufgaben bestehen jeweils aus mehreren Schritten, die zum Teil aufeinander aufbauen, jedoch unabhängig voneinander beurteilt werden. Dadurch ist einerseits eine objektive Beurteilung sichergestellt und andererseits gewährleistet, dass auch bei unvollständiger Lösung der Aufgaben Punkte erzielt werden können.

Wir weisen ausdrücklich darauf hin, dass die Übungsaufgaben von jedem Teilnehmer eigenständig gelöst werden müssen. Wenn Quellcode anderen Teilnehmern zugänglich gemacht wird (bewusst oder durch Vernachlässigung eines gewissen Mindestmaßes an Datensicherheit), wird das betreffende Beispiel bei allen Beteiligten mit 0 Punkten bewertet, unabhängig davon, wer den Code ursprünglich erstellt hat. Ebenso ist es nicht zulässig, Code aus dem Internet, aus Büchern oder aus anderen Quellen zu verwenden. Es erfolgt sowohl eine automatische als auch eine manuelle Überprüfung auf Plagiate.

Die Abgabe der Übungsbeispiele und die Termineinteilung für die Abgabegespräche erfolgt über ein Webportal. Die Abgabe erfolgt ausschließlich über das Abgabesystem. Eine Abgabe auf andere Art und Weise (z.B. per Email) wird nicht akzeptiert. Der genaue Abgabeprozess ist im TeachCenter beschrieben.

Die Tests werden automatisch ausgeführt. Das Testsystem ist zusätzlich mit einem Timeout von 7 Minuten versehen. Sollte Ihr Programm innerhalb dieser Zeit nicht beendet werden, wird es vom Testsystem abgebrochen. Überprüfen Sie deshalb bei Ihrer Abgabe unbedingt die Laufzeit Ihres Programms.

Da die abgegebenen Programme halbautomatisch getestet werden, muss die Übergabe der Parameter mit Hilfe von entsprechenden Konfigurationsdateien genauso erfolgen wie bei den einzelnen Beispielen spezifiziert. Insbesondere ist eine interaktive Eingabe von Parametern nicht zulässig. Sollte aufgrund von Änderungen am Konfigurationssystem die Ausführung der abgegebenen Dateien mit den Testdaten fehlschlagen, wird das Beispiel mit 0 Punkten bewertet. Die Konfigurationsdateien liegen im JSON-Format vor, zu deren Auswertung steht Ihnen `rapidjson` zur Verfügung. Die Verwendung ist aus dem Programmgerüst ersichtlich.

Jede Konfigurationsdatei enthält zumindest einen Testfall und dessen Konfiguration. Es ist auch möglich, dass eine Konfigurationsdatei mehrere Testfälle enthält, um gemeinsame Parameter nicht mehrfach in verschiedenen Dateien spezifizieren zu müssen. In manchen Konfigurationsdateien finden sich auch einstellbare Parameter, die in Form eines `select` Feldes vorliegen. Diese sollen die Handhabung der Konfigurationsdateien erleichtern und ein einfaches Umschalten der Modi gewährleisten.

Es steht Ihnen frei, z.B. zu Testzwecken eigene Erweiterungen zu implementieren. Stellen Sie jedoch sicher, dass solche Erweiterungen in Ihrem abgegebenen Code deaktiviert sind, damit ein Vergleich der abgegebenen Arbeiten mit unserer Referenzimplementierung möglich ist.

Die Programmgerüste, die zur Verfügung gestellt werden, sind unmittelbar aus unserer Referenzimplementierung abgeleitet, indem nur jene Teile entfernt wurden, die dem Inhalt der Übung entsprechen. Die Verwendung dieser Gerüste ist nicht zwingend, aber Sie ersparen sich sehr viel Arbeit, wenn Sie davon Gebrauch machen.

Literatur

[1] John Canny. A Computational Approach to Edge Detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, 8(6):679–698, 1986.