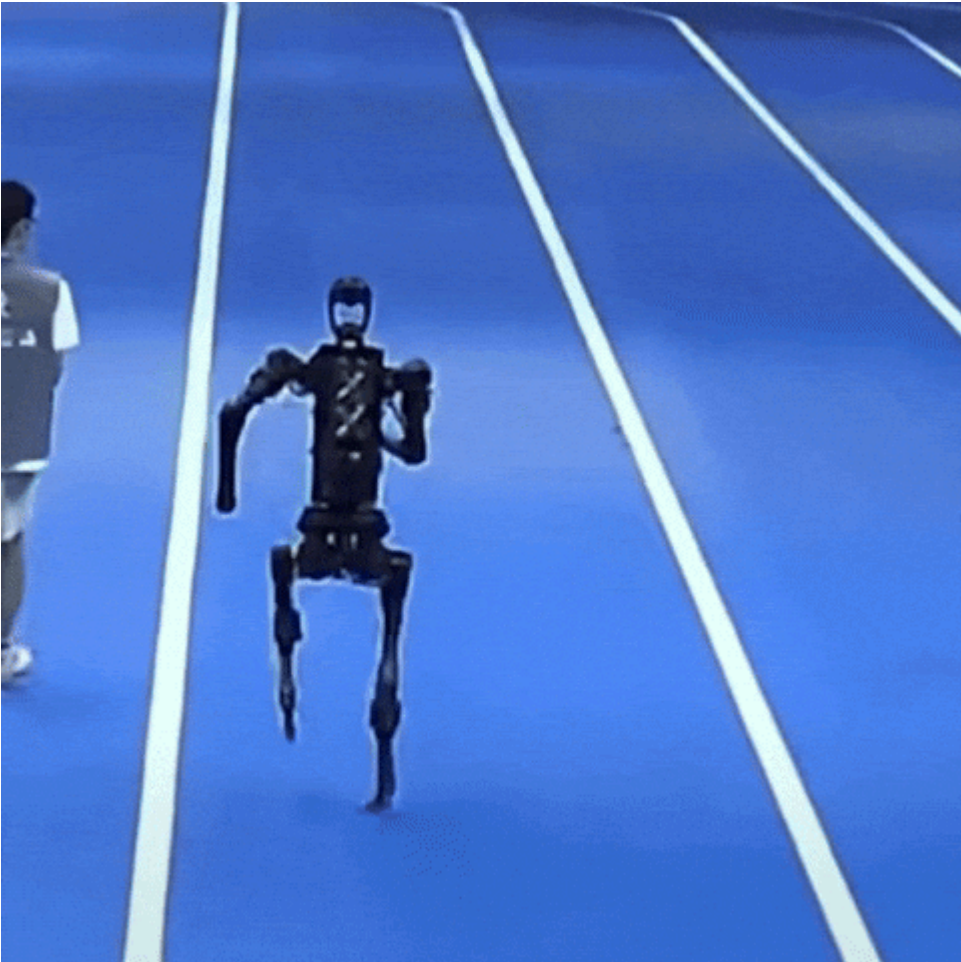


Bildverarbeitung GrundlagenRos2

Parameter	Kursinformationen
Veranstaltung:	Robotik Projekt
Semester	Wintersemester 2024/25
Hochschule:	Technische Universität Freiberg
Inhalte:	Grundlagen der Bildverarbeitung für mobile Roboter
Link auf GitHub:	https://github.com/TUBAF-lfl-LiaScript/VL_SoftwareprojektRobotik/blob/master/05_Bildverarbeitung/05_Bildverarbeitung.md
Autoren	Sebastian Zug & Claude.ai



Zielstellung der heutigen Veranstaltung

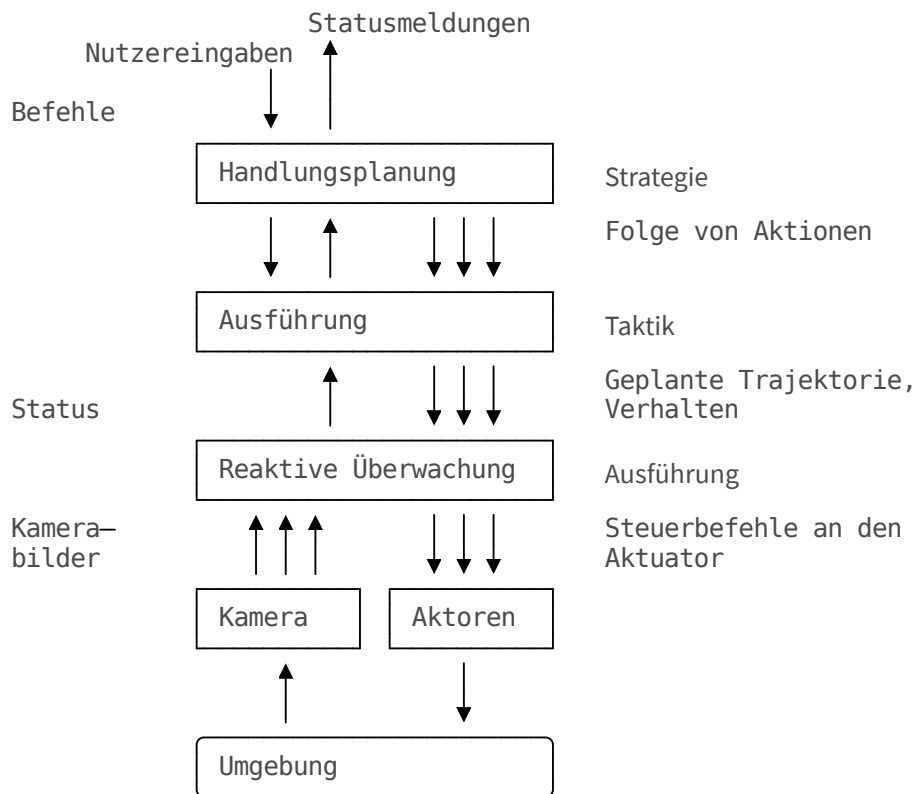
- Verständnis digitaler Bildrepräsentation und Farbräume
 - Kennenlernen grundlegender Bildverarbeitungsoperationen
 - Filterung und Rauschunterdrückung
 - Kantenerkennung als Basis für Objektdetektion
 - Einführung in Kameramodelle (Pinhole-Modell)
 - Integration von Bildverarbeitung in ROS 2 mit OpenCV
-

Motivation: Warum Bildverarbeitung in der Robotik?

Willkommen zur sechsten Vorlesung! Bisher haben wir verschiedene Sensoren kennengelernt – von der Odometrie über IMU bis hin zu Lidar und GPS. Heute fokussieren wir uns auf einen der informationsreichsten Sensoren: die Kamera. Kameras liefern uns nicht nur Distanzinformationen, sondern auch Farbe, Textur und semantische Informationen über unsere Umgebung.



Kameras als Sensoren in der Roboterarchitektur



Kamerabilder müssen verarbeitet werden, bevor sie für Regelung und Planung nutzbar sind. Diese Verarbeitungskette – von Rohbildern zu extrahierten Features – ist das Thema dieser Vorlesung.

Was macht Bildverarbeitung komplex?

Herausforderungen bei der visuellen Wahrnehmung

- **Hohe Datenrate:** Eine VGA-Kamera (640×480 Pixel) bei 30 fps erzeugt ~27 MB/s
- **Beleuchtungsvarianz:** Sonnenlicht vs. künstliches Licht, Schatten, Reflexionen
- **Perspektive und Verzerrung:** 3D-Welt → 2D-Bild, Objekte erscheinen unterschiedlich groß
- **Rauschen:** Sensorrauschen, insbesondere bei schlechten Lichtverhältnissen
- **Echtzeitanforderungen:** Roboter müssen schnell reagieren (< 100 ms)
- **Dynamische Umgebung:** Bewegte Objekte, wechselnde Szenen

Beispiel: Autonome Fahrzeuge müssen Fußgänger, Verkehrsschilder und andere Fahrzeuge in Echtzeit erkennen – bei Tag, Nacht, Regen und Gegenlicht!

Anwendungen in der mobilen Robotik

Anwendung	Bildverarbeitungstechnik	Beispiel
Hinderniserkennung	Kantendetektion, Disparitätskarten	Roboter weicht Objekten aus
Objekterkennung	Feature-Extraktion, Deep Learning	Erkennung von Personen, Tieren, Autos
Spurhaltung	Linienerkennung (Hough- Transformation)	Line-Following Roboter
Verkehrszeichenerkennung	Farb- und Formanalyse, CNN	Autonome Fahrzeuge
Visuelle Odometrie	Feature-Tracking über Bildsequenzen	SLAM (Simultaneous Localization and Mapping)
Landmarken-Navigation	Template Matching, QR-Codes, ArUco-Marker	Roboter navigiert zu bekannten Zielen

Blick nach vorn - Bilder in ROS2

Schauen wir uns mal die Bilddatenstruktur in ROS2 an. Hier das Message-Format für Bilder in ROS2:

https://docs.ros2.org/foxy/api/sensor_msgs/msg/Image.html

```
# This message contains an uncompressed image
# (0, 0) is at top-left corner of image
#std_msgs/Header header # Header timestamp should be acquisition time of image
# # Header frame_id should be optical frame of camera
# # origin of frame should be optical center of camera
# # +x should point to the right in the image
# # +y should point down in the image
# # +z should point into to plane of the image
# # If the frame_id here and the frame_id of the CameraInfo
# # message associated with the image conflict
# # the behavior is undefined
#uint32 height # image height, that is, number of rows
```

```
#uint32 width # image width, that is, number of columns
# The legal values for encoding are in file src/image_encodings.cpp
# If you want to standardize a new string format, join
# ros-users@lists.ros.org and send an email proposing a new encoding.
#string encoding # Encoding of pixels -- channel meaning, ordering, size
# # taken from the list of strings in include/sensor_msgs/image_encodings.h
#uint8 is_bigendian # is this data bigendian?
#uint32 step # Full row length in bytes
#uint8[] data # actual matrix data, size is (step * rows)

std_msgs/msg/Header header
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

Welche Bedeutung haben die einzelnen Felder?

Und wie wenden wir das Ganze an?

```
sudo apt install ros-${ROS_DISTRO}-v4l2-camera
sudo apt install v4l-utils
v4l2-ctl --list-devices
```



```
run v4l2_camera v4l2_camera_node --ros-args -p video_device:=/dev/video0
```



```
ros2 topic echo /image_raw
ros2 topic info /image_raw
ros2 topic hz /image_raw
```



Der Bilddatenstrom kann mit `rqt_image_view` oder `rviz2` visualisiert werden.

```
ros2 run rqt_image_view rqt_image_view
```



Bisweilen ist das Ganze etwas wackelig – je nach Kamera und Treiber. Hier hilft dann eine sorgfältige Treiberwahl und ggf. Anpassung der Parameter im `v4l2_camera_node`.

Lassen Sie uns die Charakteristik eines Bildes näher betrachten und ableiten, wie wir am Ende der Veranstaltung eine Gesichtserkennung in ROS2 implementieren können.

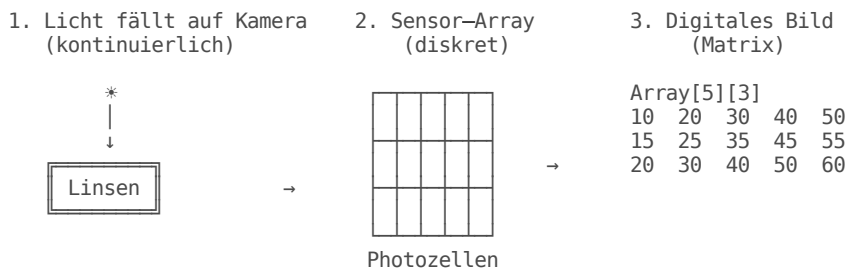
- Bilddarstellung
- Farbräume
- Grundlegende Bildoperationen
- Bilder unter ROS2 mit OpenCV verarbeiten

Grundlagen digitaler Bilder



Was ist ein Bild?

Der Weg vom Licht zum Pixel:



Zwei fundamentale Schritte:

- 1. Räumliche Diskretisierung (Sampling):** Kontinuierliche Szene → Pixelgitter - Auflösung bestimmt, wie viele Pixel (z.B. 640×480) - Je höher die Auflösung, desto mehr Details
- 2. Quantisierung:** Analog-Digital-Wandlung der Helligkeit - Typisch: 8 Bit → 256 Helligkeitsstufen (0–255) - Je mehr Bit, desto feiner die Abstufungen

Ein Bild ist eine 2D-Matrix von **Pixeln** (Picture Elements). Jeder Pixel hat:

- **Position:** Koordinaten (x, y) im Bild
- **Intensität/Farbe:** Ein oder mehrere Zahlenwerte

Was bedeutet "Auflösung" in der Praxis?

Je größer die Auflösung, desto mehr Details können erfasst werden – aber auch desto mehr Daten müssen verarbeitet werden!

Auflösung	Pixel	Graustufenbild	RGB-Bild	Typische Anwendung
QVGA	320 × 240	75 KB	~225 KB	Embedded Systems
VGA	640 × 480	300 KB	~900 KB	Webcams, einfache Roboter
HD (720p)	1280 × 720	900 KB	2.7 MB	Action-Kameras
Full HD	1920 × 1080	2 MB	~6 MB	Hochwertige Kameras
4K	3840 × 2160	8 MB	~24 MB	High-End-Anwendungen

Datenrate bei 30 fps (Full HD RGB):

$$\text{Datenrate} = 1920 \times 1080 \times 3 \times 30 \text{ fps} \approx 186 \text{ MB/s}$$

Für Echtzeitverarbeitung ist oft eine Reduzierung der Auflösung oder Kompression notwendig!

Trade-off für Robotik:

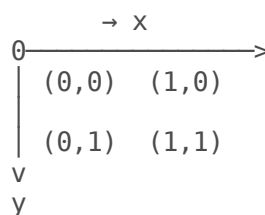
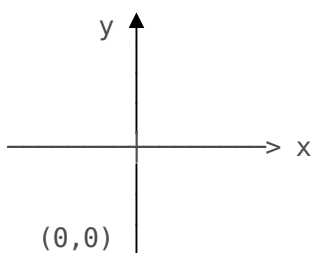
Auflösung	Vorteile	Nachteile	Robotik-Anwendung
Niedrig (320×240)	Schnelle Verarbeitung Wenig Speicher	Wenig Details	Line-Following, Farberkennung
Mittel (640×480)	Guter Kompromiss	Moderater Aufwand	Navigation, Objekterkennung
Hoch (1920×1080+)	Viele Details Präzise Messungen	Langsam Viel Speicher	Inspektion, Vermessung

Faustregel: Nutze die **niedrigste Auflösung**, die für deine Aufgabe ausreicht! Diese Festlegung ist in der Praxis aber nicht einfach und erfordert Erfahrung.

Achtung: Bildkoordinaten \neq mathematische Koordinaten!

Mathematisches System

Bildverarbeitungs-System (Standard)



Warum? Historisch: Elektronenstrahl-Röhren zeichneten von oben nach unten!

Konsequenzen für Code:

```
# Pixel oben links
top_left = image[0, 0]

# Pixel unten rechts (für 100×100 Bild)
```



```

bottom_right = image[99, 99]

# ⚠ Häufiger Fehler: Verwechslung von Bild-Koordinaten (x, y) und Array-Index [Zeile, Spalte]!
x, y = 50, 30 # Position im Bild: x=50 (horizontal), y=30 (vertikal)

pixel = image[x, y] # FALSCH! (liest Zeile 50, Spalte 30 statt Zeile 30, Spalte 50)
pixel = image[y, x] # RICHTIG! (liest Zeile 30, Spalte 50)

# Merkhilfe: Array-Index ist [Zeile, Spalte], also [y-Koordinate, x-Koordinate]

```

Faustregel: `image[y, x]` statt `image[x, y]` – weil Arrays `[Zeile, Spalte]` verwenden!

Das folgende Beispiel zeigt die Generierung eines einfachen Graustufenbildes mit NumPy und OpenCV.

Graustufenbilder haben einen Helligkeitswert pro Pixel:

- Typisch: 8 Bit pro Pixel → Wertebereich 0–255
- 0 = Schwarz, 255 = Weiß
- Dateigröße: Breite × Höhe × 1 Byte

gray_image.py

```

1 import numpy as np
2 import cv2
3
4 # Erzeuge ein 100x100 Graustufenbild
5 gray_image = np.ones((100, 100), dtype=np.uint8) * 255
6 gray_image[25:75, 25:75] = 128 # Graues Quadrat in der Mitte
7
8 print(f"Bildgröße: {gray_image.shape}") # (100, 100)
9 print(f"Datentyp: {gray_image.dtype}")  # uint8
10 print(f"Wertebereich: {gray_image.min()} - {gray_image.max()}")
11
12 cv2.imwrite('gray_image.png', gray_image)

```

```

Waking up execution server ...
This may take up to 30 seconds ...
Please be patient ...
.....

```

RGB-Farbbilder haben drei Kanäle (Rot, Grün, Blau):

- Typisch: $3 \times 8 \text{ Bit} = 24 \text{ Bit pro Pixel}$
- Jeder Kanal: 0–255
- Dateigröße: Breite \times Höhe \times 3 Bytes

Red Channel (0-255)	Green Channel (0-255)	Blue Channel (0-255)
255 200 150 ...	100 80 60 ...	50 30 20 ...
200 180 140 ...	90 70 50 ...	40 25 15 ...
...

Kombiniert zu RGB-Bild
(Höhe \times Breite \times 3 Kanäle)

color_image.py

```
# RGB-Bild in OpenCV (Achtung: OpenCV nutzt BGR!)
color_image = cv2.imread('image.jpg') # Lädt als BGR
print(color_image.shape) # z.B. (480, 640, 3) = Höhe  $\times$  Breite  $\times$  Kanäle<!--
-->

# Konvertierung BGR  $\rightarrow$  RGB
rgb_image = cv2.cvtColor(color_image, cv2.COLOR_BGR2RGB)
cv2.imwrite('gray_image.png', color_image)
```

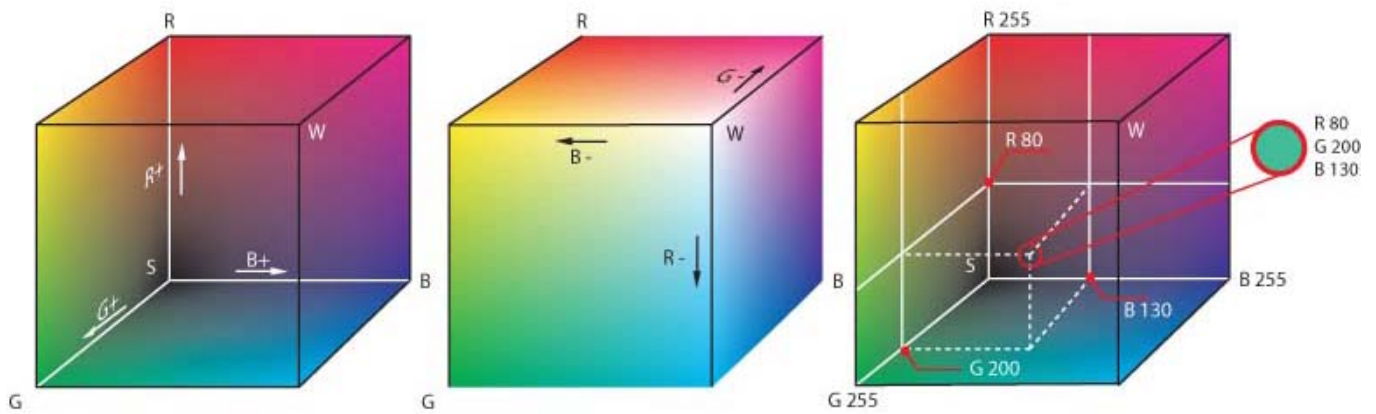
Wichtig: OpenCV verwendet standardmäßig **BGR** statt RGB! Grund: historische Konvention.

Farbräume

Farbräume definieren, wie Farbinformationen repräsentiert werden. Die Wahl des richtigen Farbraums kann die Bildverarbeitung erheblich vereinfachen.

RGB Farbraum

RGB (Red, Green, Blue) – der Standard-Farbraum für Kameras und Displays



RGB Farbraumwürfel CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=13754>

Eigenschaften:

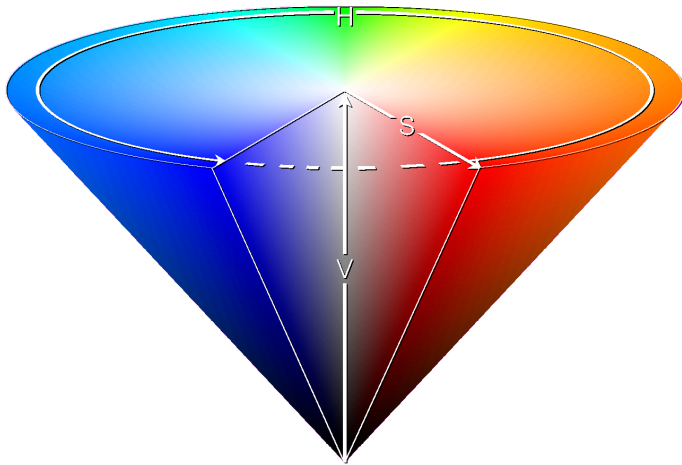
- ✓ Intuitiv für Displays und Kameras
- ✓ Direkt von Hardware unterstützt
- ✗ Nicht intuitiv für Farbauswahl (z.B. "hellblau"?)
- ✗ Empfindlich gegenüber Beleuchtungsänderungen

```
# RGB-Wert für verschiedene Farben
red   = (255, 0, 0)
green = (0, 255, 0)
blue  = (0, 0, 255)
yellow = (255, 255, 0) # Rot + Grün
white  = (255, 255, 255)
black  = (0, 0, 0)
```

HSV Farbraum

HSV (Hue, Saturation, Value) – intuitiver Farbraum für Objekterkennung

- Hue (Farbton): 0–179° (in OpenCV: 0–179 statt 0–360)
- Saturation (Sättigung): 0–255 (0 = Grau, 255 = reine Farbe)
- Value (Helligkeit): 0–255 (0 = Schwarz, 255 = hell)



HSV Farbraum von (3ucky3all) - Uploaded to en:File:HSV cone.png, CC BY-SA 3.0,
<https://commons.wikimedia.org/w/index.php?curid=943857>

Vorteile für Robotik:

- ☒ Trennung von Farbe (Hue) und Helligkeit (Value)
- ☒ Robuster gegenüber Beleuchtungsänderungen
- ☒ Ideal für farbbasierte Objekterkennung

Beispiel: Erkennung eines roten Balls



Farbtonskala Von Kalan - Eigenes Werk, Gemeinfrei, <https://commons.wikimedia.org/w/index.php?curid=1609109>

```
import cv2
import numpy as np

# Bild einlesen und zu HSV konvertieren
image = cv2.imread('ball.jpg')
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

# Rot hat zwei Bereiche im Farbkreis (0-10° und 170-180°)
# Format: [Hue (Farbton), Saturation (Sättigung), Value (Helligkeit)]

# Bereich 1: Rot am Anfang des Farbkreises (0-10°)
lower_red1 = np.array([0, 100, 100])    # Minimums: H=0°, S=100, V=100
upper_red1 = np.array([10, 255, 255])   # Maximums: H=10°, S=255, V=255

# Bereich 2: Rot am Ende des Farbkreises (170-180°)
lower_red2 = np.array([170, 100, 100])
upper_red2 = np.array([180, 255, 255])

# Masken erstellen
mask1 = cv2.inRange(hsv, lower_red1, upper_red1)
```

```
mask2 = cv2.inRange(hsv, lower_red2, upper_red2)
mask = cv2.bitwise_or(mask1, mask2)

# Rote Bereiche im Bild finden
result = cv2.bitwise_and(image, image, mask=mask)
```

Wie funktioniert die Erkennung?

Die Tabelle zeigt, welche Rottöne erkannt werden und warum:

Farbe	Hue	Saturation	Value	Wird erkannt?
Kräftiges Rot	5°	200	200	✅ Ja (Bereich 1: 0-10°, S≥100, V≥100)
Dunkles Rot	175°	150	80	❌ Nein (Value < 100, zu dunkel)
Rosa (helles Rot)	0°	50	255	❌ Nein (Saturation < 100, zu wenig Farbe)
Leuchtendes Rot	2°	255	255	✅ Ja (Bereich 1: perfektes Rot)
Orange	15°	200	200	❌ Nein (Hue > 10°, außerhalb Bereich)

Warum zwei Bereiche? Rot liegt an beiden Enden des Farbkreises (0° und 180°). Daher benötigen wir zwei Masken, die mit `bitwise_or` kombiniert werden.

Vergleich: RGB vs. HSV

Eigenschaft	RGB	HSV
Intuitivität	Schwierig für Menschen	Intuitiv (Farbton, Sättigung)
Beleuchtungsrobustheit	Empfindlich	Robuster (H unabhängig von V)
Farbsegmentierung	Kompliziert	Einfach (nur H-Kanal prüfen)
Hardware-Unterstützung	Nativ	Konvertierung notwendig
Rechenaufwand	Gering	Konvertierung erforderlich
Anwendung Robotik	Grundlegende Operationen	Objekterkennung nach Farbe

Faustregel:

- RGB: Wenn Farbe nicht wichtig ist oder Rechenleistung begrenzt
- HSV: Wenn Objekte nach Farbe erkannt werden sollen

Graustufenkonvertierung

Oft ist Farbe nicht notwendig → Konvertierung zu Graustufen spart Speicher und Rechenzeit

Methode 1: Einfacher Durchschnitt (nicht empfohlen)

$$\text{Gray} = \frac{R + G + B}{3}$$

Methode 2: Gewichteter Durchschnitt (Standard)

$$\text{Gray} = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

Gewichtung reflektiert die menschliche Wahrnehmung: Grün erscheint uns heller als Blau

loadImage.py



```
1 import cv2
2 import numpy as np
3
4 import urllib.request
5
6 def load_image_from_url(url):
7     resp = urllib.request.urlopen(url)
8     image_data = resp.read()
9     image_array = np.asarray(bytearray(image_data), dtype=np.uint8)
10    image = cv2.imdecode(image_array, cv2.IMREAD_COLOR)
11    return image
```

grayscale.py



```
1 import cv2
2 import numpy as np
3 from loadImage import load_image_from_url
4
5 image = load_image_from_url('https://r4r.informatik.tu-freiberg.de/co
    /images/size/w960/2022/08/karl_kegel_bau1.jpg')
6
7 # OpenCV Graustufenkonvertierung
8 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
9
10 # Oder manuell (weniger effizient):
11 gray_manual = 0.299 * image[:, :, 2] + 0.587 * image[:, :, 1] + 0.114 * i
   [:, :, 0]
12
13 cv2.imwrite('gray_image.png', gray)
14 cv2.imwrite('gray_image_manual.png', gray_manual)
```

.....

Vorteile Graustufen:

- ⚡ 3× weniger Speicher
- ⚡ Schnellere Verarbeitung
- ✓ Ausreichend für viele Algorithmen (Kantenerkennung, optischer Fluss)

Grundlegende Bildoperationen

Bevor wir komplexe Algorithmen anwenden, schauen wir uns grundlegende Operationen an, die das Fundament der Bildverarbeitung bilden. Diese können sich auf einzelne Pixel, Pixelgruppen oder das gesamte Bild beziehen.

Punktoperationen

Punktoperationen verändern jeden Pixel unabhängig von seinen Nachbarn

1. Helligkeitsanpassung

$$I_{\text{out}}(x, y) = I_{\text{in}}(x, y) + \text{offset}$$

Addition verschiebt alle Werte gleichmäßig in Richtung des Weißen (255).

2. Kontrastanpassung

$$I_{\text{out}}(x, y) = \alpha \cdot I_{\text{in}}(x, y)$$

Addition verschiebt alle Werte gleichmäßig, verändert aber nicht den Abstand!

3. Invertierung

$$I_{\text{out}}(x, y) = 255 - I_{\text{in}}(x, y)$$

loadImage.py



```
1 import cv2
2 import numpy as np
3
4 import urllib.request
5
6 def load_image_from_url(url):
7     resp = urllib.request.urlopen(url)
8     image_data = resp.read()
9     image_array = np.asarray(bytearray(image_data), dtype=np.uint8)
10    image = cv2.imdecode(image_array, cv2.IMREAD_COLOR)
11    return image
```



```

1 import cv2
2 import numpy as np
3 from loadImage import load_image_from_url
4
5 image = load_image_from_url('https://r4r.informatik.tu-freiberg.de/co
    /images/size/w960/2022/08/karl_kegel_bau1.jpg')
6
7 brighter = cv2.add(image, 50) # Automatische Sättigung bei 255
8 brighter_np = np.clip(image + 50, 0, 255).astype(np.uint8)
9 inverted = cv2.bitwise_not(image)
10
11 cv2.imwrite('gray_image.png', image)
12 cv2.imwrite('gray_image_processed.png', inverted)

```

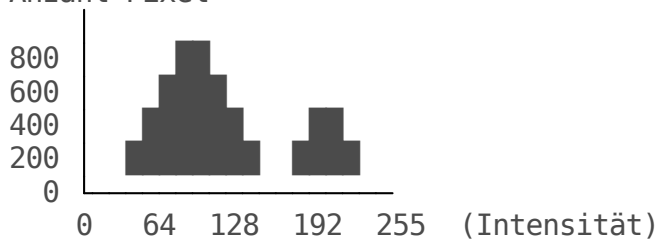
.....

Probieren Sie verschiedene Parameter aus!

Histogramme

Ein **Histogram** zeigt die Verteilung der Pixelintensitäten

Anzahl Pixel



Helles Bild: Verteilung rechts
 Dunkles Bild: Verteilung links
 Guter Kontrast: Breite Verteilung

.

loadImage.py



```
1 import cv2
2 import numpy as np
3
4 import urllib.request
5
6 def load_image_from_url(url):
7     resp = urllib.request.urlopen(url)
8     image_data = resp.read()
9     image_array = np.asarray(bytearray(image_data), dtype=np.uint8)
10    image = cv2.imdecode(image_array, cv2.IMREAD_COLOR)
11    return image
```

pixelanpassung.py



```
1 import cv2
2 import numpy as np
3 from matplotlib import pyplot as plt
4 from loadImage import load_image_from_url
5
6 image = load_image_from_url('https://r4r.informatik.tu-freiberg.de/co
    /images/size/w960/2022/08/karl_kegel_bau1.jpg')
7
8 # OpenCV Graustufenkonvertierung
9 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
10 cv2.imwrite('gray_image.png', gray)
11
12 plt.hist(gray.ravel(),256,[0,256]);
13 plt.savefig('histogram.png')
```

.....

Histogramm-Equalization – verbessert Kontrast automatisch

```
# Nur für Graustufenbilder
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
equalized = cv2.equalizeHist(gray)

# Für Farbbilder: Jeden Kanal einzeln
b, g, r = cv2.split(image)
b_eq = cv2.equalizeHist(b)
g_eq = cv2.equalizeHist(g)
r_eq = cv2.equalizeHist(r)
equalized_color = cv2.merge([b_eq, g_eq, r_eq])
```



Anwendung: Verbesserung von Bildern bei schlechten Lichtverhältnissen (z.B. Nachtsicht)

Schwellwertoperationen (Thresholding)

Binärisierung – Umwandlung in Schwarz-Weiß-Bild

Eingangsbild (Graustufen)	Schwellwert $T=128$	Ausgangsbild (Binär)
50 100 150 200	→	0 0 255 255
80 120 160 180		0 0 255 255
30 90 170 220		0 0 255 255

$$I_{\text{out}}(x, y) = \begin{cases} 255 & \text{wenn } I_{\text{in}}(x, y) > T \\ 0 & \text{sonst} \end{cases}$$

Einfaches Thresholding

```
# Fester Schwellwert
ret, binary = cv2.threshold(gray, 128, 255, cv2.THRESH_BINARY)
#
#
#
# Threshold)
#
#
```

4. Thresholding-Typ
3. Maximalwert (für Pixel
2. Schwellwert (Threshold
1. Eingangsbild (Graustu

Adaptives Thresholding – Schwellwert variiert lokal

pseudocode.txt

```
# Für Pixel an Position (x, y):
T_lokal(x,y) = Mittelwert(Umgebung_11x11) - C

# Dann Vergleich:
if pixel(x,y) > T_lokal(x,y):
    output = 255
else:
    output = 0
```

```
# Besser bei ungleichmäßiger Beleuchtung
adaptive = cv2.adaptiveThreshold(
    gray,
    255,          # Zielwert für Pixel > Schwellwert
    cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
    cv2.THRESH_BINARY,
    blockSize=11, # Größe der lokalen Umgebung
```

```
    C=2          # Konstante, die vom Mittelwert abgezogen wird
)
```

Anwendung: Dokumentenscanning, QR-Code-Erkennung, Kontursegmentierung

loadImage.py



```
1 import cv2
2 import numpy as np
3
4 import urllib.request
5
6 def load_image_from_url(url):
7     resp = urllib.request.urlopen(url)
8     image_data = resp.read()
9     image_array = np.asarray(bytearray(image_data), dtype=np.uint8)
10    image = cv2.imdecode(image_array, cv2.IMREAD_COLOR)
11    return image
```

pixelanpassung.py



```
1 import cv2
2 import numpy as np
3 from matplotlib import pyplot as plt
4 from loadImage import load_image_from_url
5
6 image = load_image_from_url('https://github.com/TUBAF-IfI-LiaScript
    /VL_SoftwareprojektRobotik/blob/master/05_Bildverarbeitung/images
    /QR_example.png?raw=true')
7
8 # OpenCV Graustufenkonvertierung
9 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
10
11 # Einfaches Thresholding
12 ret, binary = cv2.threshold(gray, 128, 255, cv2.THRESH_BINARY)
13
14 # Adaptives Thresholding
15 adaptive = cv2.adaptiveThreshold(
16     gray,
17     255,
18     cv2.ADAPTIVE_THRESH_MEAN_C,
19     cv2.THRESH_BINARY,
20     blockSize=91,
21     C=3
22 )
23 cv2.imwrite('adaptive_image.png', adaptive)
24 cv2.imwrite('binary_image.png', binary)
```

Morphologische Operationen

Morphologie arbeitet auf binären Bildern und ändert Objektformen

Erosion – verkleinert Objekte, entfernt kleine Störungen

Original (1 = weiß)

```
0 0 0 0 0
0 1 1 1 0
0 1 1 1 0
0 1 1 1 0
0 0 0 0 0
```



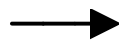
Nach Erosion

```
0 0 0 0 0
0 0 0 0 0
0 0 1 0 0
0 0 0 0 0
0 0 0 0 0
```

Dilatation – vergrößert Objekte, schließt Lücken

Original

```
0 0 0 0 0
0 1 1 1 0
0 1 1 1 0
0 1 1 1 0
0 0 0 0 0
```



Nach Dilatation

```
0 1 1 1 0
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
0 1 1 1 0
```

```
kernel = np.ones((5,5), np.uint8)
```

```
# Erosion: Entfernt Rauschen
```

```
eroded = cv2.erode(binary, kernel, iterations=1)
```

```
# Dilatation: Schließt Lücken
```

```
dilated = cv2.dilate(binary, kernel, iterations=1)
```

```
# Opening: Erosion gefolgt von Dilatation (entfernt kleine Objekte)
```

```
opening = cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel)
```

```
# Closing: Dilatation gefolgt von Erosion (schließt Löcher)
```

```
closing = cv2.morphologyEx(binary, cv2.MORPH_CLOSE, kernel)
```



Anwendungen:

- Rauschentfernung in binären Bildern
- Trennung verbundener Objekte
- Schließen von Lücken in Konturen

Filterung und Rauschunterdrückung

Reale Kamerabilder enthalten immer Rauschen – verursacht durch Sensorrauschen, Kompression oder Übertragungsfehler. Filterung ist essenziell für robuste Bildverarbeitung.



Verrauschtes Bild in dunkler Umgebung von Lugiadoom at English Wikipedia - Transferred from en.wikipedia to Commons by Mo7amedsalim using CommonsHelper.

Arten von Bildrauschen

Gaußsches Rauschen – Normalverteiltes Rauschen

- Ursache: Thermisches Rauschen im Sensor, schlechte Beleuchtung
- Modell: $I_{\text{noisy}}(x, y) = I(x, y) + \mathcal{N}(0, \sigma^2)$

Salt-and-Pepper-Rauschen – zufällige schwarze/weiße Pixel

- Ursache: Defekte Pixel, Übertragungsfehler
- Erscheinung: Vereinzelte weiße (255) und schwarze (0) Pixel

Original Bild		Gaußsches Rauschen		Salt-and-Pepper	
100 100 100		105 98 102		105 98 0	←
100 100 100	→	97 103 99	→	97 103 99	
100 100 100		101 96 104		101 255 104	

<<<<<< HEAD

=====

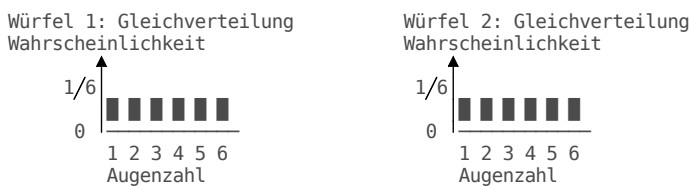
>>>>>> e317a44b8ad346132fdb19b3df5ce59f0bdc182

Wiederholung Faltungsoperationen

Was ist Faltung? Die Faltung (Convolution) ist eine mathematische Operation, die zwei Funktionen kombiniert, um eine dritte zu erzeugen. In der Bildverarbeitung verwenden wir sie, um Filter anzuwenden.

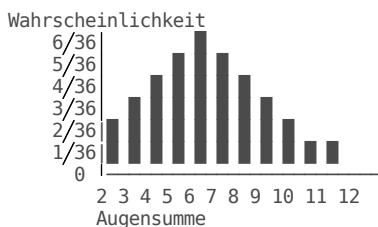
Intuitive Erklärung mit Würfeln:

Stellen Sie sich vor, Sie werfen zwei faire Würfel. Welche Augensummen sind wie wahrscheinlich?



Welche Verteilung ergibt sich für die **Summe** der beiden Würfel?

Die 7 ist am wahrscheinlichsten ($6/36 = 1/6$)! Warum? Es gibt 6 Kombinationen: (1,6), (2,5), (3,4), (4,3), (5,2), (6,1)



Berechnung der Faltung (Beispiel für Summe = 7):

$$P(\text{Summe} = 7) = \sum_{i=1}^6 P(\text{Würfel}_1 = i) \cdot P(\text{Würfel}_2 = 7 - i)$$

Verallgemeinert für zwei Verteilungen P bedeutet das

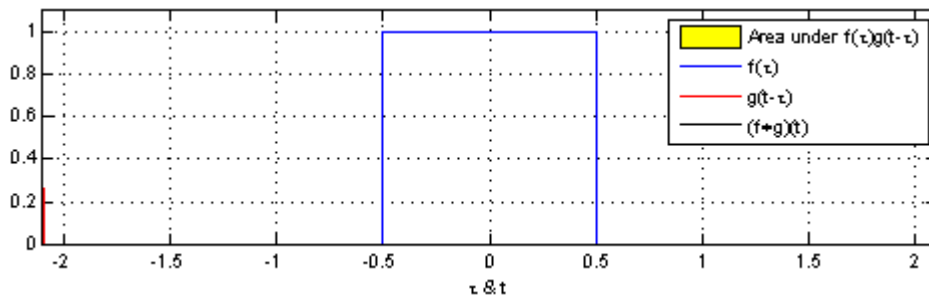
$$P * P(n) = \sum_k P(k) \cdot P(n - k)$$

Beispiel: $P * P(7) = P(1)P(6) + P(2)P(5) + P(3)P(4) + P(4)P(3) + P(5)P(2) + P(6)P(1)$

$$\text{Beispiel: } P * P(3) = P(1)P(2) + P(2)P(1) = 2 \cdot \frac{1}{6} \cdot \frac{1}{6} = \frac{2}{36}$$

wobei * die Faltungsoperation ist.

Wenn wir uns das ganze im kontinuierlichen Fall anschauen, erhalten wir das Integral über den Flächen der beiden Kurven.



Von Convolutionofboxsignalwithitself.gif: Brian Ambergderivative work: Tinos (talk) - Convolutionofboxsignalwithitself.gif, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=11003835>

Analogie zur Bildverarbeitung:

Würfelbeispiel	Bildverarbeitung
Würfel 1 (Verteilung)	Eingangsbild (Pixelwerte)
Würfel 2 (Verteilung)	Filter-Kernel (Gewichte)
Summenverteilung	Gefiltertes Bild
Aufsummieren aller Kombinationen	Pixel × Kernel aufsummieren

Kernaussage: Faltung kombiniert zwei Signale/Verteilungen zu einem neuen. In der Bildverarbeitung kombinieren wir Pixelwerte mit Filter-Gewichten!

Lineare Filter: Faltung (Convolution)

Konzept der Faltung bei der Bildverarbeitung: Wir erweitern das Konzept auf 2D-Bilder.

Eingangsbild		Kernel (3×3)		Ausgangsbild I_{out}
10 20 30 40 50		1/9 1/9 1/9		0 0 0 0 0
15 25 35 45 55	*	1/9 1/9 1/9	=	0 25 35 45 55
20 30 40 50 60		1/9 1/9 1/9		0 30 40 50 60
25 35 45 55 65				...

Für jeden Pixel: Ausgabe = Summe(Eingabe × Kernel)

Mathematisch:

$$I_{out}(x, y) = \sum_{i=-k}^k \sum_{j=-k}^k I_{in}(x + i, y + j) \cdot K(i, j)$$

Für die Zelle [1,1] im obigen Beispiel ergibt sich also mit einem Mittelwert-Filter

- $I_{out}(1, 1) = (10 + 20 + 30 + 15 + 25 + 35 + 20 + 30 + 40) / 9 = 25$
- $I_{out}(1, 2) = (20 + 30 + 40 + 25 + 35 + 45 + 30 + 40 + 50) / 9 = 35$
- usw.

Problematisch an den Rändern: Hier fehlen Nachbapixel. Lösungen:

1. Ignorieren (kleineres Ausgangsbild)
2. Auffüllen mit Nullen (schwarzer Rand)
3. Spiegeln der Ränder

Was kann ich damit anfangen? Die Faltung als Grundkonzept ermöglicht sehr viele Basisanwendungen wie Glättung, Kantenerkennung, Schärfung, etc.

Box-Filter (Durchschnittsfilter) – Einfache Glättung

```
# 5×5 Box-Filter
kernel_box = np.ones((5,5), np.float32) / 25
smoothed = cv2.filter2D(image, -1, kernel_box)

# Oder direkt:
smoothed = cv2.blur(image, (5,5))
```



Gaußscher Filter

Gaußfilter – Glättung mit Gaußscher Gewichtung

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

Eigenschaften:

- Zentrale Pixel werden stärker gewichtet als äußere
- Parameter σ (Sigma) steuert die Stärke der Glättung
- Besser als Box-Filter: weniger Artefakte

3×3 Gaußkernel-Approximation:

```
1/16 · [ 1  2  1 ]
       [ 2  4  2 ]
       [ 1  2  1 ]
```

```
# Gaußscher Weichzeichner
gaussian = cv2.GaussianBlur(image, (5,5), sigmaX=1.0)

# Größerer Kernel = stärkere Glättung
strong_blur = cv2.GaussianBlur(image, (15,15), sigmaX=3.0)
```

Beispiel: Gauß-Filter zur Rauschreduktion

```
1 import cv2
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 # Erstelle künstliches Bild mit geometrischen Formen
6 image = np.zeros((300, 300), dtype=np.uint8)
7 cv2.rectangle(image, (50, 50), (150, 150), 200, -1)
8 cv2.circle(image, (225, 225), 50, 180, -1)
9 cv2.line(image, (50, 250), (150, 200), 220, 3)
10
11 # Füge Gaußsches Rauschen hinzu (normalerweise passiert das in der Ka
12 noise = np.random.normal(0, 25, image.shape).astype(np.int16)
13 noisy_image = np.clip(image.astype(np.int16) + noise, 0, 255).astype(
14     .uint8)
15
16 # Wende Gauß-Filter an
17 gaussian_filtered = cv2.GaussianBlur(noisy_image, (5, 5), sigmaX=1.5)
18 strong_filtered = cv2.GaussianBlur(noisy_image, (15, 15), sigmaX=3.0)
19
20 # Visualisierung
21 fig, axes = plt.subplots(2, 2, figsize=(10, 10))
22 axes[0, 0].imshow(image, cmap='gray')
23 axes[0, 0].set_title('Original (ohne Rauschen)')
24 axes[0, 0].axis('off')
```

```

25
26 axes[0, 1].imshow(noisy_image, cmap='gray')
27 axes[0, 1].set_title('Mit Gaußschem Rauschen')
28 axes[0, 1].axis('off')
29
30 axes[1, 0].imshow(gaussian_filtered, cmap='gray')
31 axes[1, 0].set_title('Gauß-Filter (5×5, σ=1.5)')
32 axes[1, 0].axis('off')
33
34 axes[1, 1].imshow(strong_filtered, cmap='gray')
35 axes[1, 1].set_title('Starker Gauß-Filter (15×15, σ=3.0)')
36 axes[1, 1].axis('off')
37
38 plt.tight_layout()
39 plt.savefig('gaussian_filter_example.png')

```

.....

Anwendungen:

- Rauschreduktion vor Kantendetektion
- Hintergrund-Unschärfe (Depth-of-Field-Effekte)
- Vorbereitung für Feature-Detektion

Median-Filter

Median-Filter – Nicht-linearer Filter, ersetzt Pixel durch Median der Umgebung

Eingangsbild (3×3 Umgebung)		Sortierte Werte		Ausgabe
10 20 30		[0] [4]		[8]
15 255 40	→	10 15 20 20 30 30 40 50 255	→	30 (Median)
20 30 50				

Salt-and-Pepper (255) wird ignoriert!

```

# Median-Filter mit 5×5 Kernel
median = cv2.medianBlur(image, ksize=5)

```



Vorteile:

- Sehr effektiv gegen Salt-and-Pepper-Rauschen
- Erhält Kanten besser als Gaußfilter
- Rechenaufwändiger als lineare Filter

Vergleich:

Filter	Gaußsches Rauschen	Salt-and-Pepper	Kantenerhalt	Rechenaufwand
Gaußfilter	Sehr gut	Mittel	Mittel	Mittel
Median	Gut	Sehr gut	Gut	Hoch

Bilaterale Filterung

Problem: Gaußfilter verwischen auch Kanten!

Wenn wir einen Gaußfilter anwenden, wird jeder Pixel mit seinen Nachbarn gemittelt – **unabhängig davon, ob es sich um eine Kante handelt**. Das führt dazu, dass scharfe Objektgrenzen verschwimmen.

Der bilaterale Filter ist ein **intelligenter Filter**, der zwei Fragen stellt:

1. **Wie nah ist der Nachbar?** (räumliche Distanz) – wie beim Gaußfilter
2. **Wie ähnlich ist die Helligkeit?** (Helligkeitsdifferenz) – **NEU!**

Nur wenn **BEIDE** Bedingungen erfüllt sind, wird der Nachbarpixel zum Mittelwert beigetragen.

Jeder Nachbarpixel bekommt ein **Gewicht**, das von zwei Faktoren abhängt:

$$\text{Gewicht} = \underbrace{G_{\text{räumlich}}(\text{Abstand})}_{\text{Wie beim Gaußfilter}} \times \underbrace{G_{\text{Helligkeit}}(\text{Helligkeitsdiff.})}_{\text{NEU!}}$$

Komplett:

$$I_{\text{out}}(x, y) = \frac{1}{W} \sum_{i,j} I(i, j) \cdot \underbrace{G_{\sigma_s}(\|p - q\|)}_{\text{Differenz Abstand}} \cdot \underbrace{G_{\sigma_r}(|I(p) - I(q)|)}_{\text{Differenz Helligkeit}}$$

Parameter der zugehörigen Funktion:

Parameter	Bedeutung	Effekt
<code>d</code>	Durchmesser der Umgebung	Wie groß ist die betrachtete Nachbarschaft? (z.B. 9 = 9×9)
<code>sigmaColor</code>	Helligkeits-Toleranz	Wie unterschiedlich dürfen Pixel sein, um noch gemittelt zu werden?
		Kleiner Wert (z.B. 10): Nur fast identische Pixel → Kanten bleiben sehr scharf
		Großer Wert (z.B. 150): Auch unterschiedliche Pixel → Kanten verschwimmen
<code>sigmaSpace</code>	Räumliche Toleranz	Wie stark werden entfernte Pixel gewichtet?

Typische Werte:

```
# Standard-Einstellung (gute Kantenerhaltung)
bilateral = cv2.bilateralFilter(image, d=9, sigmaColor=75, sigmaSpace=75)

# Starke Kantenerhaltung (z.B. für Cartoon-Effekt)
bilateral = cv2.bilateralFilter(image, d=9, sigmaColor=20, sigmaSpace=20)

# Mehr Glättung, weniger Kantenerhaltung
bilateral = cv2.bilateralFilter(image, d=9, sigmaColor=150, sigmaSpace=150)
```

Beispiel: Vergleich Gauß-Filter vs. Bilateraler Filter

```
1 import cv2
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 # Erstelle künstliches Bild mit scharfen Kanten
6 image = np.zeros((300, 300), dtype=np.uint8)
7 cv2.rectangle(image, (30, 30), (130, 130), 200, -1)
8 cv2.rectangle(image, (150, 150), (270, 270), 180, -1)
9 cv2.circle(image, (80, 220), 40, 220, -1)
10
11 # Füge Gaußsches Rauschen hinzu
12 noise = np.random.normal(0, 20, image.shape).astype(np.int16)
13 noisy_image = np.clip(image.astype(np.int16) + noise, 0, 255).astype(
    .uint8)
14
15 # Wende verschiedene Filter an
16 gaussian = cv2.GaussianBlur(noisy_image, (9, 9), sigmaX=2.0)
17 bilateral = cv2.bilateralFilter(noisy_image, d=9, sigmaColor=75, sigma
    =75)
18
19 # Visualisierung
20 fig, axes = plt.subplots(2, 2, figsize=(10, 10))
21
22 axes[0, 0].imshow(image, cmap='gray')
23 axes[0, 0].set_title('Original (ohne Rauschen)')
24 axes[0, 0].axis('off')
25
26 axes[0, 1].imshow(noisy_image, cmap='gray')
27 axes[0, 1].set_title('Mit Rauschen')
28 axes[0, 1].axis('off')
29
30 axes[1, 0].imshow(gaussian, cmap='gray')
31 axes[1, 0].set_title('Gauß-Filter (Kanten verwischt)')
32 axes[1, 0].axis('off')
33
34 axes[1, 1].imshow(bilateral, cmap='gray')
35 axes[1, 1].set_title('Bilateraler Filter (Kanten erhalten)')
36 axes[1, 1].axis('off')
37
38 plt.tight_layout()
39 plt.savefig('gaussian_filter_example.png')
```

.....

Beobachtung: Der bilaterale Filter reduziert das Rauschen in homogenen Bereichen, erhält aber die scharfen Kanten zwischen den Objekten, während der Gauß-Filter alle Kanten verwischt.

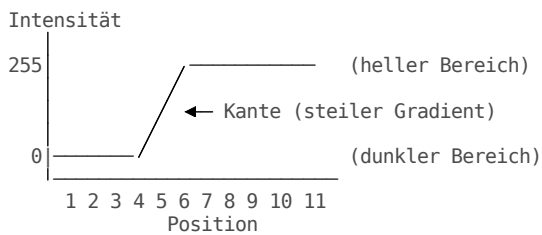
Anwendung: Hautglättung in Portrait-Fotografie, Rauschunterdrückung bei Kantenerhalt

Kantenerkennung

Kanten sind Bereiche im Bild, wo sich die Helligkeit stark ändert. Sie markieren Objektgrenzen und sind fundamental für Objekterkennung, Segmentierung und Feature-Extraktion.

Definition: Eine Kante ist ein starker Intensitätsgradient

Helligkeitsprofil über eine Kante:



Mathematisch: Erste Ableitung des Bildes

$$\nabla I = \begin{bmatrix} \frac{\partial I}{\partial x} \\ \frac{\partial I}{\partial y} \end{bmatrix}$$

Kantenstärke (Magnitude):

$$|\nabla I| = \sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2}$$

Sobel-Operator

Sobel-Filter berechnet den Gradienten in x- und y-Richtung

Sobel-Kernel für x-Richtung (vertikale Kanten):

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$



Berechnungsbeispiel: Schritt für Schritt

Gegeben sei ein kleiner Bildausschnitt (5×5 Pixel) mit einem vertikalen Helligkeitsübergang:

Eingangsbild (Grauwerte):

100	100	100	50	50
100	100	100	50	50
100	100	100	50	50
100	100	100	50	50
100	100	100	50	50

← Vertikale Kante zwischen Spalte 3 und 4

Schritt 1: Sobel-X-Filter anwenden (vertikale Kanten detektieren)

Wir berechnen den Gradientenausschnitt für den Pixel in der Mitte (Position [1,1]):

Gx-Kernel: $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ * Bildausschnitt um [1,1]: $\begin{bmatrix} 100 & 100 & 100 \\ 100 & 100 & 100 \\ 100 & 100 & 100 \end{bmatrix}$

Berechnung (elementweise Multiplikation und Summierung):

$$\begin{aligned} &> G_x = (-1 \cdot 100) + (0 \cdot 100) + (+1 \cdot 100) & (1) \\ &> + (-2 \cdot 100) + (0 \cdot 100) + (+2 \cdot 100) & (2) \\ &> + (-1 \cdot 100) + (0 \cdot 100) + (+1 \cdot 100) & (3) > \\ &> = -100 + 0 + 100 - 200 + 0 + 200 - 100 + 0 + 100 & (4) \\ &> = 0 & (5) \end{aligned}$$

Für Position [2,3] (direkt auf der Kante):

Gx-Kernel: $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ * Bildausschnitt um [1,3]: $\begin{bmatrix} 100 & 100 & 50 \\ 100 & 100 & 50 \\ 100 & 100 & 50 \end{bmatrix}$

$$\begin{aligned} &> G_x = (-1 \cdot 100) + (0 \cdot 100) + (+1 \cdot 50) & (6) \\ &> + (-2 \cdot 100) + (0 \cdot 100) + (+2 \cdot 50) & (7) \\ &> + (-1 \cdot 100) + (0 \cdot 100) + (+1 \cdot 50) & (8) > \\ &> = -100 + 0 + 50 - 200 + 0 + 100 - 100 + 0 + 50 & (9) \\ &> = -200 \text{ (starke vertikale Kante!) } & (10) \end{aligned}$$

Ok, wir können also mit dem Sobel-X-Kernel vertikale Kanten erkennen. Jetzt machen wir das gleiche für horizontale Kanten mit dem Sobel-Y-Kernel.

Sobel-Kernel für y-Richtung (horizontale Kanten):

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$



Implementierung:

```
# Gradienten in x- und y-Richtung
sobelx = cv2.Sobel(gray, cv2.CV_64F, dx=1, dy=0, ksize=3)
sobely = cv2.Sobel(gray, cv2.CV_64F, dx=0, dy=1, ksize=3)

# Kantenstärke (Magnitude)
magnitude = np.sqrt(sobelx**2 + sobely**2)
magnitude = np.uint8(np.clip(magnitude, 0, 255))

# Kantenrichtung
angle = np.arctan2(sobely, sobelx) * 180 / np.pi
```

Canny-Kantendetektor

Canny-Algorithmus – der Goldstandard für Kantendetektion

Schritte:

1. **Rauschunterdrückung:** Gaußfilter
2. **Gradientenberechnung:** Sobel-Operator
3. **Non-Maximum Suppression:** Dünne Kanten (1 Pixel breit)
4. **Hysterese-Schwelwertbildung:** Zwei Schwellwerte (low, high)

Hysterese-Schwelwert:

- Pixel > `high_threshold` → **starke Kante**
- Pixel zwischen `low` und `high` → **schwache Kante** (nur behalten, wenn mit starker Kante verbunden)
- Pixel < `low_threshold` → **verwerfen**

```
# Canny-Kantendetektor
edges = cv2.Canny(
    gray,
    threshold1=50,    # Lower threshold
    threshold2=150,   # Upper threshold
    apertureSize=3    # Sobel-Kernel-Größe
)
```

Parameter-Tuning:

- Kleines `threshold2` → viele Kanten (auch Rauschen)
- Großes `threshold2` → nur starke Kanten
- Verhältnis `threshold2 : threshold1` typisch 2:1 bis 3:1

loadImage.py

```

1 import cv2
2 import numpy as np
3
4 import urllib.request
5
6 def load_image_from_url(url):
7     resp = urllib.request.urlopen(url)
8     image_data = resp.read()
9     image_array = np.asarray(bytearray(image_data), dtype=np.uint8)
10    image = cv2.imdecode(image_array, cv2.IMREAD_COLOR)
11    return image

```

grayscale.py

```

1 import cv2
2 import numpy as np
3 from loadImage import load_image_from_url
4
5 image = load_image_from_url('https://r4r.informatik.tu-freiberg.de/co
    /images/size/w960/2022/08/karl_kegel_bau1.jpg')
6
7 # OpenCV Graustufenkonvertierung
8 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
9
10 edges = cv2.Canny(
11     gray,
12     threshold1=50,    # Lower threshold
13     threshold2=150,   # Upper threshold
14     apertureSize=3    # Sobel-Kernel-Größe
15 )
16
17 cv2.imwrite('edges.png', edges)

```

.....

Anwendungen:

- Objektkonturen finden
- Spurerkennung (Lane Detection)
- Basis für Hough-Transformation (Linien-/Kreiserkennung)

Vergleich Kantendetektor

Methode	Geschwindigkeit	Kantenqualität	Rauschrobustheit	Anwend
Sobel	Sehr schnell	Mittel	Mittel	Echtzeit, Szenen
Canny	Mittel	Sehr gut	Sehr gut	Objekte, SLAM, N

Faustregel:

- **Sobel:** Wenn Geschwindigkeit wichtig ist
- **Canny:** Wenn Qualität und Robustheit wichtig sind

Integration in ROS 2

https://docs.ros2.org/foxy/api/sensor_msgs/msg/Image.html

```
# This message contains an uncompressed image
# (0, 0) is at top-left corner of image
#std_msgs/Header header # Header timestamp should be acquisition time of image
# # Header frame_id should be optical frame of camera
# # origin of frame should be optical center of camera
# # +x should point to the right in the image
# # +y should point down in the image
# # +z should point into to plane of the image
# # If the frame_id here and the frame_id of the CameraInfo
# # message associated with the image conflict
# # the behavior is undefined
#uint32 height # image height, that is, number of rows
#uint32 width # image width, that is, number of columns
# The legal values for encoding are in file src/image_encodings.cpp
# If you want to standardize a new string format, join
# ros-users@lists.ros.org and send an email proposing a new encoding.
#string encoding # Encoding of pixels -- channel meaning, ordering, size
# # taken from the list of strings in include/sensor_msgs/image_encodings.h
#uint8 is_bigendian # is this data bigendian?
#uint32 step # Full row length in bytes
#uint8[] data # actual matrix data, size is (step * rows)
```

```
std_msgs/msg/Header header
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

cv_bridge – Die Brücke zwischen ROS und OpenCV

Problem: ROS-Messages \neq OpenCV-Bilder

Lösung: `cv_bridge` konvertiert zwischen beiden Formaten

```
from cv_bridge import CvBridge
import cv2
from sensor_msgs.msg import Image

class ImageProcessor(Node):
    def __init__(self):
        super().__init__('image_processor')
        self.bridge = CvBridge()

        # Subscriber für Kamera-Topic
        self.subscription = self.create_subscription(
            Image,
            '/camera/image_raw',
            self.image_callback,
            10
        )

        # Publisher für verarbeitetes Bild
        self.publisher = self.create_publisher(
            Image,
            '/camera/image_processed',
            10
        )

    def image_callback(self, msg):
        try:
            # ROS Image → OpenCV Image
            cv_image = self.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr')

            # Bildverarbeitung
            gray = cv2.cvtColor(cv_image, cv2.COLOR_BGR2GRAY)
            edges = cv2.Canny(gray, 50, 150)

            # OpenCV Image → ROS Image
            output_msg = self.bridge.cv2_to_imgmsg(edges, encoding='mono8')
```

```

output_msg = cv2.cvtColor(edges, cv2.COLOR_GRAY2RGB)
output_msg.header = msg.header # Zeitstempel übernehmen!

# Publizieren
self.publisher.publish(output_msg)

except Exception as e:
    self.get_logger().error(f'Fehler: {e}')

```

Wichtig:

- `imgmsg_to_cv2()` → ROS zu OpenCV
- `cv2_to_imgmsg()` → OpenCV zu ROS
- Zeitstempel beibehalten: `output_msg.header = msg.header`

Praktisches Beispiel: Farbbasierte Objekterkennung

Im Projektordner finden Sie unter `FaceDetectionNode.py` ein vollständiges Beispiel für die farbbasierte Objekterkennung mit ROS 2 und OpenCV.

```

FaceDetection/
├── src/face_detector/                                # ROS2 Python-Paket
│   ├── face_detector/
│   │   ├── face_detector_node.py                    # Hauptnode für Gesichtserkennung
│   │   └── __init__.py
│   ├── launch/
│   │   └── face_detection.launch.py                # Launch-File für komplettes System
│   ├── package.xml                                  # Paket-Metadaten
│   └── setup.py                                     # Python Setup
├── install/                                          # Build-Artefakte (generiert)
├── build/                                           # Build-Cache (generiert)
├── log/                                              # Build-Logs (generiert)
├── README.md                                         # Vollständige Dokumentation
├── QUICKSTART.md                                    # 5-Minuten-Start-Guide
├── OVERVIEW.md                                       # Diese Datei
├── build.sh                                          # Automatisches Build-Skript
├── test_camera.sh                                   # Kamera-Test-Skript
└── .gitignore                                        # Git-Ignore-Regeln

```

Komprimierte Bilder: `sensor_msgs/CompressedImage`

Problem: Unkomprimierte Bilder sind groß!

- Full HD (1920×1080 RGB): ~6 MB pro Frame
- Bei 30 fps: ~180 MB/s Netzwerklast!

Lösung: JPEG-Kompression

```
from sensor_msgs.msg import CompressedImage

class CompressedImageProcessor(Node):
    def __init__(self):
        super().__init__('compressed_processor')
        self.bridge = CvBridge()

        # Subscriber für komprimiertes Bild
        self.sub = self.create_subscription(
            CompressedImage,
            '/camera/image_raw/compressed',
            self.callback,
            10
        )

    def callback(self, msg):
        # CompressedImage → OpenCV
        cv_image = self.bridge.compressed_imgmsg_to_cv2(msg, 'bgr8')

        # ... Verarbeitung ...

        # OpenCV → CompressedImage
        output_msg = self.bridge.cv2_to_compressed_imgmsg(cv_image)
        # ...
```

Vergleich:

Format	Größe (Full HD)	Latenz	Qualität	Anwendung
Unkomprimiert	~6 MB	Niedrig	Perfekt	LAN, kurze Distanzen
JPEG (komprimiert)	~100-500 KB	Mittel	Gut	WLAN, Remote-Steuerung

Zusammenfassung und Ausblick

Was wir gelernt haben

1. Digitale Bilder

- Pixel-Repräsentation (RGB, Graustufen)
- Farbräume (RGB, HSV) und ihre Anwendungen
- Speicherbedarf und Auflösungen

2. Grundlegende Operationen

- Punktoperationen (Helligkeit, Kontrast)
- Histogramme und Equalization
- Thresholding (fest und adaptiv)
- Morphologische Operationen (Erosion, Dilatation)

3. Filterung

- Faltung und Kernel-basierte Filter
- Gaußfilter für Rauschunterdrückung
- Median-Filter für Salt-and-Pepper-Rauschen
- Bilateraler Filter für kantenerhaltende Glättung

4. Kantenerkennung

- Gradienten und Sobel-Operator
- Canny-Algorithmus als Goldstandard

5. ROS 2 Integration

- `sensor_msgs/Image` und `CompressedImage`
- `cv_bridge` für OpenCV-Integration
- Praktische Beispiele (Farbbasierte Objekterkennung)

Nächste Schritte: VL 7 – Kamerakalibrierung & Stereo

Was kommt als Nächstes?

- **Kamerakalibrierung:** Bestimmung intrinsischer und extrinsischer Parameter
- **Verzerrungskorrektur:** Praktische Anwendung
- **Stereo-Vision:** Zwei Kameras für 3D-Tiefenschätzung
- **Disparitätskarten:** Von 2D zu 3D
- **Triangulation:** Berechnung von 3D-Koordinaten

Vorbereitung:

- Installieren Sie OpenCV und ROS 2 Image-Pakete
- Testen Sie `cv_bridge` mit einer Webcam
- Experimentieren Sie mit den Beispielen aus dieser Vorlesung
- Entwickeln Sie einen Knoten, der eine Kantenerkennung durchführt