

**Praxisphasen Bericht**

# **Automatisierte Erkennung von Klarschrift auf unterschiedlichen Containertypen**

Jan Wille 1535115

2. Mai 2022

Professor(in)/Lehrbeauftragte(r): Prof. Dr.-Ing. Hanno Homann

---

## **Selbstständigkeitserklärung**

Hiermit bestätige ich, dass die folgende Arbeit eigenständig von mir allein erstellt und unter Berücksichtigung der zur Verfügung gestellten Aufgabenstellung sowie dem Arbeitsmaterial unter Angabe aller verwendeten Quellen erarbeitet wurde. Die Regelungen und Konsequenzen eines Plagiats, inklusive disziplinarischer Maßnahmen, sind mir bewusst. Insbesondere wurden alle Zitate und gedanklichen Übernahmen als solche kenntlich gemacht.

---

Jan Wille

# Inhaltsverzeichnis

<b>1</b>	<b>Problemstellung</b>	<b>1</b>
1.1	Container Typen . . . . .	1
1.2	Hardware Voraussetzungen . . . . .	1
1.3	Anforderungen an das System . . . . .	2
1.4	Testumgebung . . . . .	2
<b>2</b>	<b>Umsetzung in Python</b>	<b>3</b>
2.1	Projektstruktur . . . . .	3
2.2	Verwendete Pakete . . . . .	4
2.3	Schnittstelle . . . . .	5
2.4	Vorgehen bei der Erkennung . . . . .	5
2.4.1	Verbesserung der Bildqualität . . . . .	5
2.4.2	Binarisierung . . . . .	6
2.4.3	Morphologische Operationen . . . . .	7
2.4.4	Finden und filtern von Konturen . . . . .	7
2.4.5	Rotieren der gefundenen Sektionen . . . . .	8
2.4.6	Übergabe an Tesseract . . . . .	9
2.4.7	Textanalyse des Ergebnisses . . . . .	9
2.5	Automatisiertes Testen . . . . .	9
2.6	Packaging . . . . .	10
<b>3</b>	<b>Zusammenfassung</b>	<b>11</b>
3.1	Funktionalität . . . . .	11
3.2	Probleme . . . . .	11
3.3	Verbesserungsmöglichkeiten . . . . .	11
	<b>Literatur</b>	<b>12</b>
	<b>Abbildungsverzeichnis</b>	<b>13</b>
	<b>Tabellenverzeichnis</b>	<b>13</b>

# 1 Problemstellung

Eine Reihe von Container (im Folgenden auch als *Gebinde* bezeichnet) soll von einer Umladehalle an ihren Lagerort gebracht werden. Dabei soll vor dem Aufnehmen auf ein Stapelfahrzeug sichergestellt werden, dass es sich tatsächlich um das richtige Gebinde handelt.

Dazu sind alle Gebinde mit einem oder mehreren Codes in Klarschrift beschriftet, welche erkannt und verifiziert werden müssen. Wie genau diese Erkennung stattfinden, ist im Folgenden Dokumentiert.

## 1.1 Container Typen

In der Lagerhalle werden zwei unterschiedliche Arten von Containern eingelagert. Rechteckige Container werden direkt vom Stapelfahrzeug aufgenommen und haben nur einen einzigen zu lesenden Code. Die dem Erkennungssystem zugewandte Seite hat dabei eine Höhe von 1,7 m und eine maximale Breite von 3,2 m, es gibt aber auch schmalere Container. Die Skizze in Abbildung 1.1(a) zeigt dies ebenfalls.

Des Weiteren gibt es tonnenförmige Rundgebände, welche unterschiedliche Größen haben können. Diese werden auf Transportrahmen befestigt und dann als Gebinde bezeichnet. Ein Gebinde kann aus einem oder zwei Rundcontainern bestehen, welche mit dem Deckel zum Erkennungssystem liegend verladen sind. Jeder Rundcontainer ist einzeln auf dem Deckel mit einem Code versehen, diese können beliebig rotiert sein. Zusätzlich befindet sich noch ein weiterer Code auf dem Transportrahmen. Siehe auch Abbildung 1.1(b).



Abb. 1.1: Skizzen der unterschiedlichen Containertypen mit den möglichen Textpositionen (nach [1])

## 1.2 Hardware Voraussetzungen

Im Pflichtenheft [1] ist die bestellte Hardware aufgelistet. Für die hier behandelte Aufgabe sind aber nur einige wenige Komponenten relevant. Die genauen Bezeichnungen sind in Tabelle 1.1 aufgelistet, im Folgenden wird aber nur noch von *Kamera* und *Panel-PC* gesprochen.

Tab. 1.1: Für die Aufgabe relevante Hardware laut Pflichtenheft [1]

01	SIMATIC MV440 UR optischer Leser; Auflösung: 1600×1200 Pixel; für 1D/2D-Codelesen, Texterkennung und Objekterkennung; Bildfeld und Abstand: variabel; PoE; IP67 (mit 6GF3440-8AC1X); Lieferung: Lesegerät, CD-ROM und Kunststoff-Schutz-Tubus; (ohne: Kabel, Leuchte, Objektiv, Lizenzen)
02	MV440 Montageplatte Lesegerät; Material: Edelstahl (4 mm), vielfältig anwendbar, Befestigung: metrische Gewinde, Fotogewinde (1/4-Zoll, 2x mittig) BxHxT (mm) 80×80×60
03	Mini-Objektiv 6 mm, 1: 1,4 PENTAX C60636KP mit fester Brennweite, Blende und Fokus einstellbar D = 32 mm, L = 37,5 mm
04	D65-Objektiv-Schutz Metall-Glas, Schutzart IP67 für MV440-Geräte; Frontscheibe: Glas, Gehäuse: Metall enthält: Tubus, O-Ring und Schutzkappen (M12, M12, M16), Innen-Durchmesser: 57 mm, max. Objektivlänge: 57 mm geeignete Objektive (MLFB): z.B. 6GF9001-1BL01, ...-1BF01, ...-1BG01, ...-1BH01, ...-1BJ01 geeignete Leuchten (MLFB): 6GF3440-8DA1, ...-8DA2, ...-8DA11 BxHxT (mm) 65×65×60
10	SIMATIC IPC277E (Nanopanel PC); 7" Touch TFT; 2x 10/100/1000 MBit/s Ethernet RJ45; 1x Display-Port Grafik; 1x USB 3.0; 2x USB 2.0; 1x seriell (COM 1); CFAST-Slot; DC 24V Stromversorgung Celeron N2807 (2C/2T) 4 GB RAM WIN Embedded Standard 7 P SP1, englisch; 64 Bit 80 GB SSD ohne SIMATIC Software

### 1.3 Anforderungen an das System

Das zu erstellende System soll auf Kommando des Stapelfahrzeug-Fahrers das vor dem Fahrzeug befindliche Gebinde scannen und die gefundenen Codes mit einer Liste von für diese Schicht gültigen Codes vergleichen. Dabei soll es keinen Unterschied machen, welcher Containertyp gerade vor dem Fahrzeug steht. Insbesondere die beliebige Rotation der Rundbinde muss berücksichtigt werden.

Zusätzlich zu der hier behandelten Komponenten kann eine weitere Software vorausgesetzt werden. Diese läuft auf dem Panel-PC und präsentiert dem Fahrer eine grafische Oberfläche (GUI). Am Anfang der Schicht erhält diese eine Liste mit Arbeitsaufträgen für die Schicht, inklusive der Codes der zu transportierenden Container. Diese Software stößt den Scanprozess an und erhält das Ergebnis, um es dem Fahrer grafisch darzustellen. Sollte der Lesevorgang nur teilweise oder gar nicht funktionieren, soll der Fahrer so viele Teilinformationen wie möglich erhalten und wird von der Zusatzsoftware zum manuellen Eingreifen aufgefordert.

Außerdem soll die Möglichkeit bestehen ein Foto des Abstellortes aufzunehmen und zur Archivierung an den erfolgreichen Auftrag anzuhängen.

### 1.4 Testumgebung

Da keine realen Fotos existieren und die Anlage erst in einigen Jahren fertiggestellt wird, stammen die für diesen Bericht verwendeten Aufnahmen von einem Testaufbau. Dieser wurde in einem Konferenzraum aufgebaut, welcher sich vollständig abdunkeln lässt. Dadurch lassen sich die Lichtverhältnisse kontrollieren und die in der Realität zu erwartenden Lichtverhältnisse durch manuelle Beleuchtung nachstellen.

## 2 Umsetzung in Python

Das Softwaremodul zur Schrifterkennung wird in Python umgesetzt. Das folgende Kapitel dient zur Dokumentation des Codes.

### 2.1 Projektstruktur

Grundsätzlich steht das Projekt unter Versionskontrolle durch die Software *Git*. Diese ermöglicht es Änderungen zwischen Softwareiterationen einfach nachzuverfolgen und macht eine Synchronisation mit einem oder mehreren Servern möglich. Dadurch ist sichergestellt, dass jederzeit auf eine funktionierende Version der Software zurückgerollt werden kann.

Dazu werden unter anderem *Tags* verwendet, die einen bestimmten Softwarezustand betiteln. In diesem Projekt werden die funktionierenden Programmversionen mit einem Tag versehen, welches die Versionsnummer enthält.

Alle Daten dieses Projektes sind auf einem Siemens internen Server unter der Adresse <https://code.siemens.com/jan.wille/klarschrifterkennung> abgelegt. Aus Sicherheitsgründen müssen Zugriffsrechte für dieses Projekt aber explizit erteilt werden.

Um das Projekt übersichtlich und organisiert zu halten, wurden mehrere Unterordner angelegt. Die Bedeutung dieser und der verbleibenden Dateien im Stammverzeichnis sind in der folgenden Tabelle erläutert:

Tab. 2.1: Erklärungen zur Projektstruktur

Ordner/Dateiname	Erklärung
<code>.venv/</code>	Dieser Ordner wird beim Erzeugend des <i>Virtual Environments</i> (siehe 2.2) erstellt. In ihn werden alle Bibliotheken für das venv installiert
<code>src/</code>	Hier befinden sich die eigentlichen Quellcodedateien
<code>tesseract/</code>	Die binären Dateien für das Programm Tesseract befinden sich hier
<code>tests/</code>	Die Testscripte für <code>pytest</code> befinden sich hier
<code>tests/img/</code>	Ein Ablageort für Testbilder
<code>.flake8</code>	Konfigurationsdatei für den Python-Syntaxchecker <code>flake8</code>
<code>.gitignore</code>	Diese Datei enthält Informationen für Git, welche Dateien/Ordner ignoriert werden sollen
<code>Makefile</code>	Das <code>Makefile</code> enthält Befehle oder Befehlsketten die für das Arbeiten mit dem Projekt relevant sein können
<code>README.md</code>	Eine Textdatei im Markdownformat die eine Projektbeschreibung enthält
<code>requirements.txt</code>	Diese Datei enthält eine Liste aller benötigten Bibliotheken (siehe 2.2)

## 2.2 Verwendete Pakete

Dieses Projekt verwendet einige Pakete. Diese sind in der Datei `requirements.txt` aufgelistet. Außerdem ist eine kurze Zusammenfassung ihrer Funktion in Tabelle 2.2 zu finden.

Tab. 2.2: Verwendete Pakete und ihre Funktion

*Runtime Pakete:*

<code>opencv-python</code> [2]	Ein Wrapper für das OpenCV Framework [3]. Stellt eine Reihe an Funktionen für die Bildverarbeitung zur Verfügung. Ein Großteil aller verwendeten Algorithmen sind hier implementiert.
<code>numpy</code> [4]	Eine Bibliothek zu schnellen Verarbeitung großer Datenstrukturen (insbesondere Matrizen). Sie liegt den Bild-Datentypen in <code>opencv-python</code> zugrunde.
<code>pytesseract</code> [5]	Wrapper für das OCR-Programm <i>Tesseract</i> [6]. Dieses für die eigentliche Texterkennung durch und kann durch diese Bibliothek bedient werden.

*Pakete für die Entwicklung:*

<code>pyinstaller</code> [7]	Eine Bibliothek zum Verpacken eines Pythonprogrammes in ein alleinstehendes Programm.
<code>black</code>	Python Formatierungsprogramm.
<code>flake8</code>	Python Syntaxchecker.
<code>pytest</code>	Python Unittest Framework. Ermöglicht automatisiertes Testen.

Um das Projekt zum ersten Mal auf einen neuen Computer zu benutzen, müssen diese Pakete installiert werden. Hierzu wird das Anlegen eines sogenannten *Virtual Enviroments* empfohlen. Dadurch werden die Pakete nicht für den gesamten Computer installiert, sondern in einem Unterordner dieses Projektes abgelegt und nur diesem Projekt zur Verfügung gestellt. Dazu werden die folgenden Befehle oder das Makefilerezept `setup` verwendet.

```
# virtual env erstellen:
python -m venv .venv
# venv für diese Terminal-Session aktivieren (windows):
call .venv\Scripts\activate
# pakete installieren:
pip install -r requirements.txt
```

Der Quellcode des Projektes befindet sich im Unterordner `./src`. Dort befindet sich die Datei `__main__.py`, als Haupteinstiegspunkt des Programms, sowie alle weiteren selbsterstellten Pakete. Das Programm wird also mittels `python src/__main__.py` gestartet (ein Beispiel ist im Makefilerezept `run` gezeigt).

## 2.3 Schnittstelle

Da das Softwaremodule von einem übergeordneten Programm aufgerufen wird, erhält es beim Start von diesem die nötigen Informationen. Diese werden einfach als Kommandozeilenparameter übergeben. Das erste Argument muss ein Dateipfad zum zu analysierenden Bild sein, danach folgen eine beliebig lange Reihe an gültigen Codes. Über die Flag `-h` kann außerdem eine Hilfe aufgerufen werden, die die Benutzung erklärt und nachfolgend abgedruckt ist.

```
usage: __main__.py [-h] imagepath code [code ...]

positional arguments:
imagepath  Path to the image that should be processed
code       valid codes to be compared against

options:
-h, --help  show this help message and exit
```

Die Kommandozeilenparameter werden direkt an die Funktion `main()` übergeben, die alternativ zur Nutzung über die Kommandozeile von anderen Python-Programmen importiert und aufgerufen werden kann.

Sowohl die Funktion `main()` als auch der Aufruf über die Kommandozeile geben eine Liste mit gefundenen Codes, die mit der gegebenen Liste übereinstimmen, zurück.

## 2.4 Vorgehen bei der Erkennung

Das Programm durchläuft mehrer Schritte, um den Text auf einem Bild zu identifizieren. Diese werden im Folgenden einzeln erläutert.

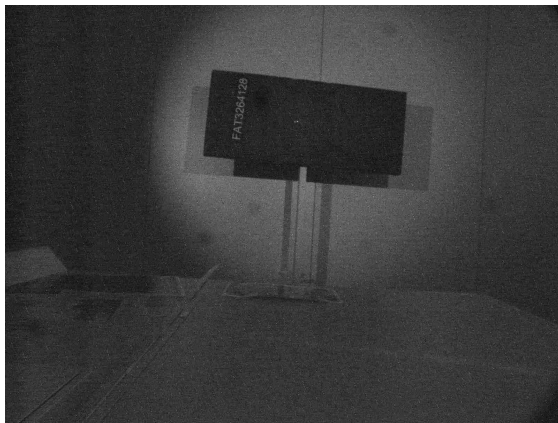
### 2.4.1 Verbesserung der Bildqualität

Da die Bilder direkt von der Kamera viele Störungen aufweisen, wird das Bild zuerst gefiltert. Diese Funktionalität ist in der Funktion `preprocessing.smooth_image()` implementiert, welche ein Bild als Parameter annimmt und die gefilterte Version zurückgibt.

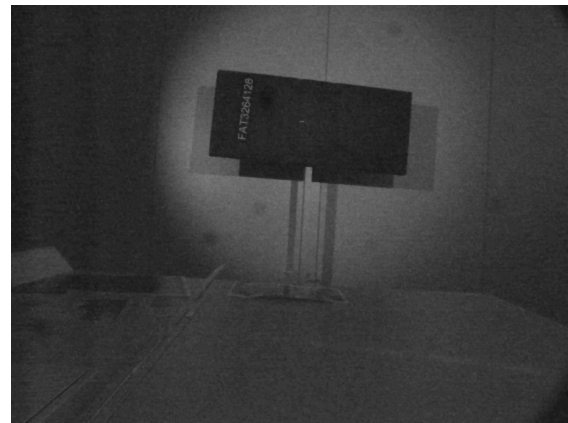
Der erste Filterschritt entfernt die sogenannten *Salt&Pepper* Störungen. Damit sind einzelne weiße oder schwarze Pixel gemeint, die durch Fehler in einzelnen Transistoren des Kamerasensors oder im Kamerachip entstehen. Diese werden durch einfache Mittelwertbildung mit allen angrenzenden Pixeln eliminiert. [8]

Danach wird ein bilateraler Filter mit einem  $7 \times 7$  Kernel auf das gesamte Bild angewandt. Dadurch werden Bildsektionen mit annähernd identischen Farbwerten vereinheitlicht, gleichzeitig bleiben Kanten (also stark unterschiedliche Farbwerte bei Nachbarn) aber erhalten. [8]

Abbildung 2.1 zeigt einen direkten Vergleich vor und nach der Filterung. Wie man sieht, sind die kleinen Störungen alle herausgefiltert, das Bild ist aber etwas unschärfer geworden. Das ist aber notwendig um die Ergebnisse der nachfolgenden Schritte zu verbessern.



(a) Originalbild direkt von der Kamera



(b) Gefiltertes Bild

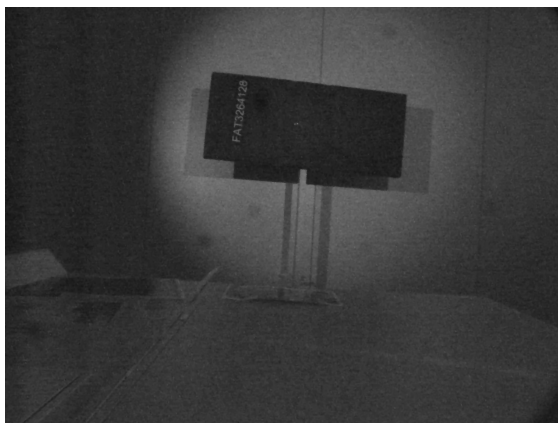
Abb. 2.1: Vergleich eines Bildes vor und nach der Filterung

### 2.4.2 Binarisierung

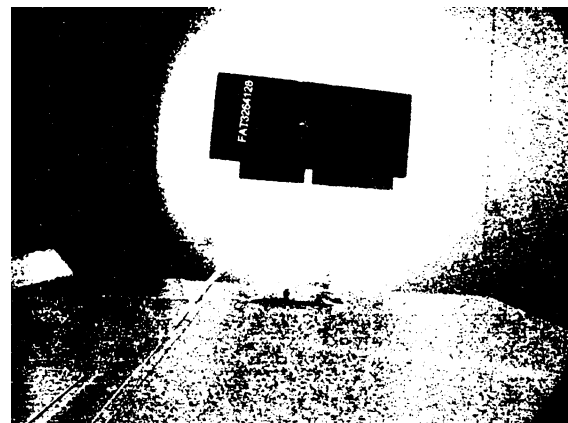
Als Nächstes wird das Bild binarisiert. Das Bild wird also auf nur zwei Farben heruntergebrochen, in dem nur noch die prominenten Features enthalten sind. Implementiert ist dies in der Funktion `preprocessing.make_binary_image()`.

Dazu wird ein in OpenCV fertig implementiert Algorithmus namens *Otsu* verwendet. Dieser ermittelt automatisch einen geeigneten Grenzwert und färbt dunklere Bereiche schwarz und hellere Bereich weiß. [9]

Durch Tests hat sich ergeben, dass die Beleuchtung und andere Faktoren so unterschiedlich sein können, dass mehrere Binarisierungsschritte notwendig sind. Dazu wird anhand des ersten Grenzwertes ein Teil des Bildes weiß eingefärbt und dann ein zweiter Grenzwert bestimmt. Die Abbildung 2.2 zeigt das fertig binarisierte Bild nach dem zweiten Schritt.



(a) Gefiltertes Bild



(b) Binarisiertes Bild

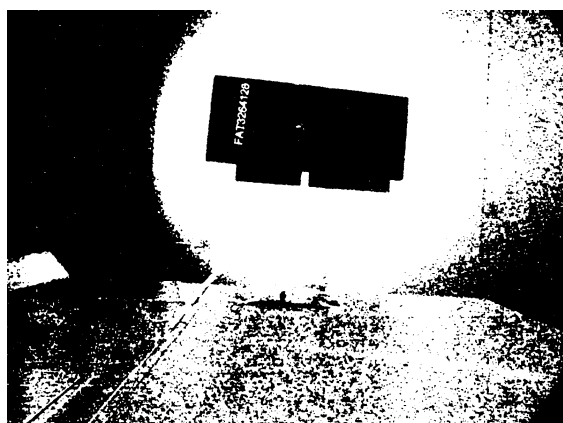
Abb. 2.2: Vergleich eines Bildes vor und nach der Binarisierung

### 2.4.3 Morphologische Operationen

Um einzelne, zufällig Störungen oder andere Objekte im Bild von tatsächlicher Schrift zu unterscheiden, werden einige morphologische Operationen angewandt. Diese sind in der Funktion `preprocessing.morphologic_close()` implementiert.

Zuerst wird eine *Closing* Operation mit einem Rechteck-Kernel angewandt. Die Kernelgröße entspricht dabei ungefähr der erwarteten Größe eines Buchstabens. Dadurch werden Buchstaben zu Rechtecken und kleine Lücken zwischen den Buchstaben werden geschlossen. Schriftzüge werden dadurch zu langgezogenen Boxen, wie in Abbildung 2.3 gut zu erkennen ist. [10]

Um verbleibende kleine Störungen zu eliminieren, wird das Bild außerdem zuerst *erodiert* und danach *dilatiert*. Durch die Erosion werden alle Umrisse verkleinert, wobei kleine Objekte ganz verschwinden. Danach wird durch die Dilatation der Ursprungszustand wieder hergestellt. [10]



(a) Binarisiertes Bild



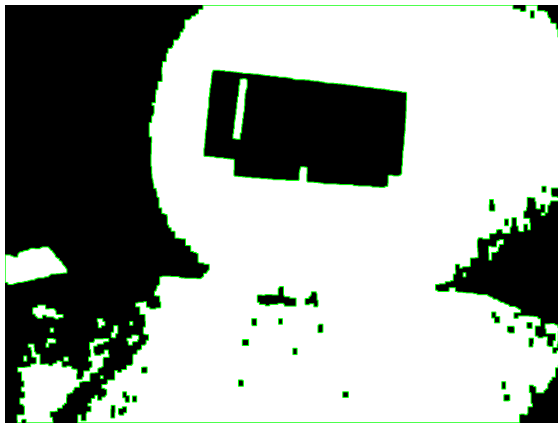
(b) Gemorphtes Bild

Abb. 2.3: Vergleich eines Bildes vor und nach den morphologischen Operationen

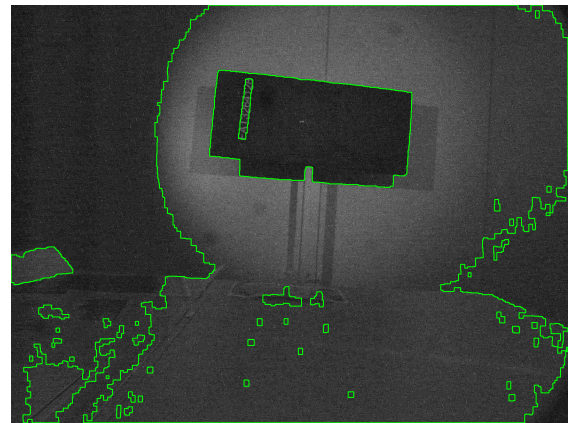
### 2.4.4 Finden und filtern von Konturen

In dem vorbereiteten Bild können nun über einen von OpenCV zur Verfügung gestellten Algorithmus Konturen gesucht werden. Die gefundenen Konturen lassen sich dann auf das originale Bild übertragen wie es in Abbildung 2.4 gezeigt ist.

Wie man sieht, werden viel mehr Konturen gefunden als eigentlich relevant sind. Daher werden die Konturen nach Fläche und Seitenverhältnis gefiltert. Um eine gültige Kontur wird dann eine rechteckige Box gelegt und etwas vergrößert, sodass ein kleiner Rand um den Text verbleibt.



(a) Konturen im binarisierten Bild



(b) Konturen übertragen auf das originale Bild

Abb. 2.4: Im Bild gefundene Konturen

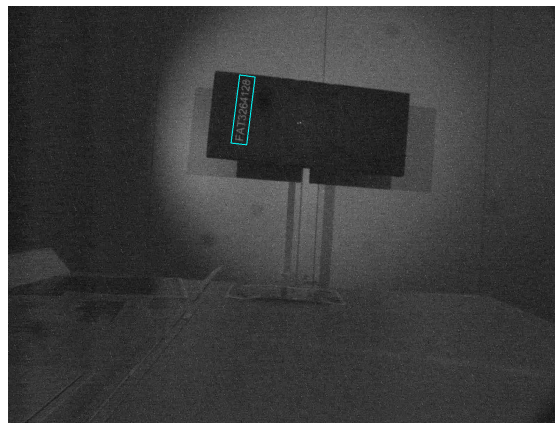


Abb. 2.5: Im Bild gefundene Textbox

### 2.4.5 Rotieren der gefundenen Sektionen

Die gefundenen Boxen liefern direkt eine Winkelinformation. Mit dieser und der Bildgröße lässt sich eine Rotationsmatrix erzeugen mit deren Hilfe das Originalbild rotiert wird. Gleichzeitig wird auch die Box rotiert und aus dem Bild ausgeschnitten.

Da der Text auch vertikal ausgerichtet sein kann, wird dies anhand des Seitenverhältnisses überprüft. Ist der Bildausschnitt höher als Breit, wird er um  $90^\circ$  rotiert.

Abbildung 2.6 zeigt, wie das Ergebnis bei diesem Beispiel aussieht.

A small rectangular image showing the text 'FAT3264128' rotated 90 degrees counter-clockwise. The text is white on a dark background.

Abb. 2.6: Ausgeschnittene und rotierte Textbox

### 2.4.6 Übergabe an Tesseract

Für die eigentliche Texterkennung wird eine weite Software namens *Tesseract* verwendet. Für diese gibt es eine Python-Wrapper-Bibliothek namens *pytesseract*, welche die Verwendung stark vereinfacht. [6, 5]

Die Kommunikation mit der Bibliothek ist in die Datei `src/tesseract.py` ausgelagert. Dort steht die Funktion `tesseract.read_text_in_image()` zur Verfügung, die ein Bild akzeptiert und den gefundenen Text zurückgibt. Diese wird für jeden Bildausschnitt aufgerufen.

Für das Beispiel aus Abbildung 2.6 wird das erwartete Ergebnis `FAT3264128` auch geliefert.

### 2.4.7 Textanalyse des Ergebnisses

Der von Tesseract gefundene Text wird zum Abschluss analysiert und mit der Liste an gültigen Codes verglichen. Dadurch werden weitere Beschriftungen oder Hersteller-Logos herausgefiltert. Alle Funktionalitäten hierfür sind in die Datei `src/findcodes.py` ausgelagert. Hier ist ein regulärer Ausdruck definiert, welcher die Voraussetzung, das gültige Codes mit 2-3 Großbuchstaben beginnen und darauf 5-9 Ziffern folgen, abbildet. Dies wird mit dem übergebenen Text verglichen und Code-Kandidaten ermittelt. Alle Kandidaten werden mit der Liste von bekanntem Code verglichen und die gültigen Codes zurückgegeben.

## 2.5 Automatisiertes Testen

Im Unterordner `./tests` befindet sich Testcode für das Python-Modul `pytest`. Dieser macht es sehr einfach, eine Reihe an Testfällen zu definieren.

Dazu ist in der Datei `tests/test_imagerecon.py` die Funktion `test_knownimages()` implementiert. Für diese ist zuerst eine Liste gültiger Codes global definiert. Dann werden über den Funktions-Dekorator `@pytest.mark.parametrize` einzelne Testfälle dekoriert. Dieser definiert eine Liste mit Bilddateien und dem darauf befindlichen Codes und testet dann, ob die `main()` mit diesen Parametern das erwartete Ergebnis liefert. Die Funktion kann also leicht an aktuelle Bilder angepasst werden. Hier der entsprechende Code:

```
valid_codes = ["SIE20220101", "FAT3264128"]
@pytest.mark.parametrize(
    ("path", "contained_codes"),
    [
        ("tests/img/image1.png", ["FAT3264128"]),
        ("tests/img/image2.png", ["SIE20220101"]),
    ],
)
def test_knownimages(path, contained_codes):
    assert contained_codes == main(path, valid_codes)
```

Um die Tests durchzuführen, kann direkt der Befehl `pytest` oder das Makefilerezept `test` verwendet werden.

## 2.6 Packaging

Da das Programm auf einem Panel-PC mit Windows laufen soll und das Unterhalten einer Python-Installation auf diesem zusätzlichen Aufwand bedeuten würde, wird das Programm verpackt (*ge-packaged*). Dazu dient das Python-Modul `pyinstaller`, welches das Programm in eine ausführbare Datei verpackt und die vollständige Python-Runtime dazu *packaged*. Zusätzlich werden die binären Dateien für Tesseract mit verpackt.

Um den Packaging-Prozess anzustoßen, kann das Makefilerezept *pack* verwendet werden. Dadurch wird `pyinstaller` mit den entsprechenden Optionen angestoßen und erzeugt den Unterordner `dist/klarschrifterkennung`. Hier liegt dann das Programm `Klarschrifterkennung.exe`, dass auf jedem Windows Rechner einfach gestartet werden kann.

Da es sich bei dem auf dem Panel-PC installieren Windows um die Version 7 handelt, ist es notwendig, eine zusätzliche Datei zur Verfügung zu stellen. Diese wird ebenfalls vom Makefile heruntergeladen und im Ordner abgelegt.

Ist das Programm *gepackaged*, kann der Ordner einfach gezippt und auf den Panel-PC kopiert werden.

## 3 Zusammenfassung

### 3.1 Funktionalität

Die Herangehensweise funktioniert auf den ausgewählten Testbildern mit guten Verhältnissen sehr gut. Zwar werden bei bestimmten Voraussetzungen einige zufällige, falsche Bildausschnitte detektiert, diese liefern aber einfach keinen Text zurück und werden verworfen. Es können beliebige Rotationen erkannt werden und auch eine Spiegelung des Textes ist theoretisch kein Problem. Außerdem können beliebig viele Codes im selben Bild detektiert werden.

### 3.2 Probleme

Unterschiedliche, suboptimale Bildverhältnisse können immer noch zu Fehlern führen. Zuerst einmal kann Schrift, die sich zu nah am Rand befindet bei den morphologischen Operationen mit diesem Verschmelzen. Dadurch wird sich nicht mehr erkannt. Außerdem sind sehr schwache Kontraste, z.B. durch ungünstige Beleuchtung, ein Problem. Diese werden evtl. nicht korrekt binarisiert.

Das größte Problem ist aber die Bildqualität. Sowohl Bildstörungen durch Staub, Beschädigungen und Ähnliches können den Text unlesbar machen. Außerdem muss grundsätzlich genug Beleuchtung vorhanden sein, das ansonsten die Tiefenschärfe des Bildes nicht ausreicht. In sehr seltenen Fällen ist ein Bild außerdem einfach zu verschwommen, um den Text zu erkennen.

### 3.3 Verbesserungsmöglichkeiten

Grundsätzlich ist es nötig, die genauen Bedingungen vor Ort zu kennen. Damit lassen sich die Parameter des Programms anpassen und das Ergebnis weiter verbessern.

Will man noch weiter gehen, wäre auch ein Ansatz mit einem Neuralen Netz denkbar. Dieser erfordert aber viel Rechenleistung, und da der Panel-PC über keine Grafikkarte verfügt, ist dies nur bedingt umsetzbar.

## Literatur

- [1] Actemium Cegelec GmbH / A. Buchholz. *Pflichtenheft ZLT - Hardwarebeschreibung - Gebindeerkennung und -verfolgung*. GER. 16. Juli 2018.
- [2] *Python Package Index - opencv-python*. ENG. 9. März 2022. URL: <https://pypi.org/project/opencv-python/> (besucht am 02.05.2022).
- [3] *OpenCV*. ENG. 2022. URL: <https://opencv.org/> (besucht am 02.05.2022).
- [4] *Python Package Index - numpy*. ENG. URL: <https://pypi.org/project/numpy/> (besucht am 07.03.2022).
- [5] *Python Package Index - pytesseract*. ENG. 19. Feb. 2022. URL: <https://pypi.org/project/pytesseract/> (besucht am 02.05.2022).
- [6] *Tesseract User Manual*. ENG. 24. Apr. 2022. URL: <https://tesseract-ocr.github.io/tessdoc/> (besucht am 02.05.2022).
- [7] *Python Package Index - pyinstaller*. ENG. 25. Apr. 2022. URL: <https://pypi.org/project/pyinstaller/> (besucht am 02.05.2022).
- [8] *OpenCV - Smoothing Images*. ENG. 2. Mai 2022. URL: [https://docs.opencv.org/4.x/d7/d4d/tutorial\\_py\\_thresholding.html](https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html) (besucht am 02.05.2022).
- [9] *OpenCV - Image Thresholding*. ENG. 2. Mai 2022. URL: [https://docs.opencv.org/4.x/d7/d4d/tutorial\\_py\\_thresholding.html](https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html) (besucht am 02.05.2022).
- [10] *OpenCV - Morphological Transformations*. ENG. 2. Mai 2022. URL: [https://docs.opencv.org/4.x/d9/d61/tutorial\\_py\\_morphological\\_ops.html](https://docs.opencv.org/4.x/d9/d61/tutorial_py_morphological_ops.html) (besucht am 02.05.2022).

## Abbildungsverzeichnis

1.1	Skizzen der unterschiedlichen Containertypen mit den möglichen Textpositionen (nach [1]) . . . . .	1
2.1	Vergleich eines Bildes vor und nach der Filterung . . . . .	6
2.2	Vergleich eines Bildes vor und nach der Binarisierung . . . . .	6
2.3	Vergleich eines Bildes vor und nach den morphologischen Operationen . . . . .	7
2.4	Im Bild gefundene Konturen . . . . .	8
2.5	Im Bild gefundene Textbox . . . . .	8
2.6	Ausgeschnittene und rotierte Textbox . . . . .	8

## Tabellenverzeichnis

1.1	Für die Aufgabe relevante Hardware laut Pflichtenheft [1] . . . . .	2
2.1	Erklärungen zur Projektstruktur . . . . .	3
2.2	Verwendete Pakete und ihre Funktion . . . . .	4