Barcode Detector And Decoding

Submitted to:

MUHAMMAD SIDDIQUE

Submitted by

Hammad Shahid
01-134221-026

Maria Ashraf
01-134221-037

Department of Computer Science, Bahria University, Islamabad.

Date: 30/05/2025

Contents

Abstract

This project presents the development and implementation of an advanced system for localizing and decoding various types of barcodes from digital images and live camera feeds. The system leverages a suite of Digital Image Processing (DIP) techniques through the OpenCV library in Python. It comprises three main modules: (1) A robust barcode localization module that employs gradient analysis, adaptive thresholding, and extensive morphological operations to identify potential barcode regions, particularly horizontal 1D barcodes. (2) A versatile barcode decoding module that utilizes the Pyzbar library, enhanced by a pipeline of preprocessing steps including CLAHE, Gaussian blurring, multiple thresholding methods (adaptive and global, normal and inverted), and morphological operations to improve decoding accuracy across diverse image conditions. (3) A camera captures utility for acquiring images for real-time or batch processing. The project demonstrates the practical application of DIP concepts such as image enhancement, filtering, segmentation, and feature analysis for real-world computer vision task. The system provides configurable parameters for fine-tuning performance and includes visualization of intermediate and results.

1 Introduction

1.1 Background

Barcodes are ubiquitous in modern society, serving as a compact and efficient means of encoding information for automated identification and data capture across various sectors, including retail, logistics, healthcare, and manufacturing. They represent data by varying the widths and spacings of parallel lines (1D barcodes like EAN, UPC, Code128) or patterns of squares, dots, hexagons, and other geometric patterns within a 2D matrix (2D barcodes like QR Codes, DataMatrix). The ability to accurately and rapidly detect and decode these symbols from images is crucial for many automated systems.

1.2 The Role of Digital Image Processing (DIP)

Digital Image Processing plays a fundamental role in enabling machines to "see" and interpret barcodes. Raw images captured by cameras often suffer from imperfections such as poor lighting, noise, blur, perspective distortion, and varying orientations. DIP techniques are essential for:

- **Preprocessing:** Enhancing the image quality, reducing noise, and improving contrast to make barcodes more discernible.
- **Localization:** Identifying the region(s) of interest (ROI) within an image that potentially contain a barcode.
- **Segmentation:** Separating the barcode pattern from the background.
- **Decoding:** Extracting the encoded information from the segmented barcode pattern.

1.3 Project Aim and Objectives

The primary aim of this project is to design, implement, and evaluate a comprehensive system for localizing and decoding 1D and 2D barcodes using Python and the OpenCV library.

The specific objectives are:

1. To implement a robust algorithm for localizing potential barcode regions in an image, focusing on morphological operations and gradient analysis.
2. To develop a flexible decoding module that employs various preprocessing strategies to enhance the success rate of a standard decoding library (Pyzbar).
3. To create a utility for capturing images from a live camera feed for testing and potential real-time application.
4. To demonstrate the practical application of core DIP concepts learned in the CEN 444 course.
5. To provide a well-documented system with configurable parameters for experimentation and optimization.

1.4 Motivation

The motivation stems from the practical need for reliable barcode reading systems and the academic desire to apply DIP principles to a tangible problem. Challenges such as variable image quality, barcode damage, or complex backgrounds make barcode detection and decoding non-trivial. This project seeks to address some of these challenges by combining multiple DIP techniques.

1.5 Scope of the Project

This project focuses on:

- Detection and localization of primarily horizontal 1D barcodes using morphological image processing.
- Decoding of common 1D (EAN, UPC, Code39, Code128, ITF) and 2D (QR Code) barcodes.
- Image-based processing and a utility for image acquisition from a webcam.
- Using Python with OpenCV for image processing and Pyzbar for decoding.

The project does not delve into advanced machine learning or deep learning models for barcode detection, focusing instead on classical DIP techniques.

2 Literature Review

Barcode detection and decoding have been active research areas for decades. Early methods often relied on specific hardware scanners. With the advent of digital cameras and powerful image processing, software-based solutions became prevalent.

2.1 Barcode Localization Techniques (`detect_barcode_opencv.py`)

Localization is the first critical step. Various approaches exist:

- **Edge-Based Methods:** Barcodes inherently possess strong edge information. Techniques like Sobel, Canny, or Prewitt edge detectors can highlight barcode bars. Subsequent processing, such as Hough Transform for line detection or connected component analysis, can group these edges (Kaur & Singh, 2017).
- **Gradient-Based Methods:** Similar to edge-based, these methods analyze image gradients. The provided `barcode_localization.py` script uses Sobel gradients, particularly the difference between horizontal and vertical gradients, to emphasize structures with strong vertical intensity changes (typical of horizontal barcodes).
- **Morphological Operations:** These are powerful for shape analysis. Operations like erosion, dilation, opening, and closing, often applied to binarized gradient images, can help connect disparate barcode bars into a single region and remove noise (Gonzalez & Woods, 2018). The `barcode_localization.py` script heavily relies on these for refining candidate regions. Rectangular structuring elements are particularly useful for 1D barcodes.
- **Texture Analysis:** Barcodes exhibit a distinct texture. Methods like Gabor filters or Local Binary Patterns (LBP) can be used to identify barcode-like textures.
- **Machine Learning/Deep Learning:** More recent approaches use classifiers (e.g., SVMs trained on HOG features) or deep convolutional neural networks (CNNs) like YOLO or SSD to directly detect barcode regions. These often offer higher accuracy but require significant training data and computational resources.

2.2 Barcode Decoding

Once localized, the barcode region is processed for decoding.

- **Preprocessing for Decoding:** Before feeding to a decoder, images often undergo binarization (e.g., Otsu's method, adaptive thresholding) to clearly separate bars from spaces. The `barcode_decode.py` script demonstrates a multi-strategy preprocessing pipeline, trying various techniques like CLAHE, blurring, and different thresholding methods, as no single method works universally.
- **Scanning and Pattern Recognition:** For 1D barcodes, scanlines are typically drawn across the barcode region. The sequence of black (bars) and white (spaces) pixel run-lengths is measured and compared against standard barcode symbologies. For 2D barcodes, finder patterns, alignment patterns, and timing patterns are identified to establish a grid, and then individual modules are read.
- **Decoding Libraries:** Several open-source libraries encapsulate these complex decoding algorithms:
  - **ZBar:** A popular, mature library for reading various barcode symbologies from images and video. Pyzbar is a Python wrapper for ZBar, used in this project's `barcode_decode.py`.
  - **ZXing ("Zebra Crossing"):** Another widely used, multi-format 1D/2D barcode image processing library, originally in Java, with ports to other languages.
  - **Proprietary SDKs:** Many commercial SDKs offer advanced decoding capabilities.

2.3 Challenges in Barcode Processing

Common challenges include:

- **Image Quality:** Low resolution, poor focus (blur), motion blur.
- **Lighting Conditions:** Uneven illumination, shadows, glare, low contrast.
- **Noise:** Sensor noise, compression artifacts.
- **Occlusion and Damage:** Partially hidden or physically damaged barcodes.
- **Geometric Distortions:** Perspective skew, rotation.
- **Complex Backgrounds:** Textures or patterns similar to barcodes.

This project addresses some of these through robust localization and adaptive preprocessing before decoding.

3 Methodology

The system is architected as a set of modular Python scripts, each addressing a specific part of the barcode processing pipeline. The core DIP techniques are implemented using OpenCV.

3.1 Module 1: Advanced Barcode Localization (`detect_barcode_opencv`)

This module aims to identify regions in an image that are likely to contain a horizontal 1D barcode. It employs a series of DIP steps:

1. **Image Acquisition and Resizing:**
   - The input image is loaded using `cv2.imread()`.
   - It's resized using `cv2.resize()` with configurable factors (`resize_fx`, `resize_fy`). Resizing can speed up processing and sometimes help normalize feature scales. `cv2.INTER_AREA` is used for downscaling, which is generally good for preserving image quality.
2. **Preprocessing for Gradient Calculation:**
   - **Grayscale Conversion:** The image is converted to grayscale using `cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)` because barcode information is primarily luminance-based, and this reduces computational complexity.
   - **Contrast Limited Adaptive Histogram Equalization (CLAHE):** `cv2.createCLAHE()` followed by `clahe.apply()` enhances local contrast. This is particularly useful for images with varying illumination, making faint barcode bars more prominent. Parameters like `clahe_clip_limit` and `clahe_tile_grid_size` control its behavior.
3. **Gradient Computation:**
   - **Sobel Operator:** `cv2.Sobel()` is used to compute the image gradient in the X (`dx=1, dy=0`) and Y (`dx=0, dy=1`) directions. The `ksize` parameter determines the kernel size (e.g., -1 for Scharr filter, which can be more accurate for small kernels). Gradients highlight regions of rapid intensity change, characteristic of barcode edges.
   - **Gradient Subtraction:** The vertical gradient (`gradY`) is subtracted from the horizontal gradient (`gradX`) using `cv2.subtract(gradX, gradY)`. For horizontal barcodes, `gradX` (vertical edges) will be strong, while `gradY` (horizontal edges) will be weak. This subtraction enhances the vertical bar structures.
   - **Conversion to 8-bit:** `cv2.convertScaleAbs()` converts the resulting gradient image to an 8-bit unsigned integer format, suitable for subsequent operations.
4. **Binarization of Gradient Image:**
   - **Gaussian Blurring:** The gradient image is blurred using `cv2.GaussianBlur()` with `gradient_blur_ksize`. This smooths the image, reducing noise and helping to merge closely spaced gradient responses before thresholding.
   - **Adaptive Thresholding:** `cv2.adaptiveThreshold()` is applied. Unlike global thresholding, it calculates different thresholds for different regions of the image based on local pixel neighborhoods (`adaptive_thresh_block_size`). `cv2.ADAPTIVE_THRESH_GAUSSIAN_C` uses a weighted sum of neighborhood pixel values. The C parameter (`adaptive_thresh_C`) is a constant subtracted from the mean or weighted sum. This method is robust to varying illumination. `THRESH_BINARY` is used.
5. **Morphological Operations for Region Refinement:**
   - **Opening:** `cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel, iterations)` with a small rectangular kernel (`morph_open_ksize`) is performed. Opening (erosion followed by dilation) removes small noise elements (salt noise) and tiny protrusions.

- **Closing:** `cv2.morphologyEx(opened, cv2.MORPH_CLOSE, kernel)` with a wider rectangular kernel (`morph_close_kernel_w`, `morph_close_kernel_h`) is applied. Closing (dilation followed by erosion) fills small holes and connects nearby components, aiming to merge the individual bars of a barcode into a single solid region. The kernel is designed to be wide and short to connect horizontal barcode elements.
- **Optional Erosion and Dilation:** `cv2.erode()` followed by `cv2.dilate()` with `erode_dilate_iterat` can further refine regions by removing thin connections or enlarging core components. This step is made optional as it can be aggressive.

6. **Contour Detection and Filtering:**
   - **Contour Finding:** `cv2.findContours(closed.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_S` extracts the boundaries of the white regions (potential barcodes) from the morphologically processed binary image. `RETR_EXTERNAL` retrieves only the outermost contours.
   - **Filtering Criteria:** Each contour is evaluated against a set of geometric properties to determine if it represents a barcode:
     - **Area:** `cv2.contourArea()`. Contours smaller than `min_barcode_area_ratio` (relative to image size) are discarded.
     - **Aspect Ratio:** Calculated from the `cv2.minAreaRect()` (which finds the minimum-area rotated rectangle enclosing the contour). The ratio of the longer side to the shorter side must fall within `min_aspect_ratio` and `max_aspect_ratio`. Barcodes are typically elongated rectangles.
     - **Number of Vertices:** `cv2.approxPolyDP()` approximates the contour shape. Barcodes, being roughly rectangular, should have a small number of vertices (e.g., 4 to 8, `min_vertices_for_rect`, `max_vertices_for_rect`).
     - **Solidity:** `cv2.contourArea(c) / cv2.contourArea(cv2.convexHull(c))`. The ratio of contour area to its convex hull area. Barcode regions are generally convex, so solidity should be high (above `min_solidity`).
     - **Orientation/Angle:** The angle of the `minAreaRect`. For horizontal barcodes, this angle should be close to 0° or -90° (depending on how `minAreaRect` defines width/height). A `max_angle_deviation` parameter controls this.

7. **Visualization:**
   - Detected barcode regions passing all filters are highlighted by drawing their bounding boxes (obtained from `cv2.boxPoints()`) onto a copy of the original resized image using `cv2.drawContours()`.

3.2 Module 2: Barcode Decoding with Preprocessing (`barcode_decode.py`)

This module focuses on decoding barcodes using the Pyzbar library, augmented by a flexible preprocessing pipeline to improve robustness.

1. **Image Acquisition and Resizing:**
   - Similar to Module 1, the image is loaded and optionally resized using configurable factors.

2. **Multi-Strategy Preprocessing Pipeline:**
   - The core idea is to try decoding on several variations of the input image, as different preprocessing steps might be optimal for different image conditions or barcode types.
   - **Grayscale Conversion:** The base for most subsequent operations.
   - **CLAHE (Optional):** `cv2.createCLAHE()` if `use_clahe` is enabled. Improves local contrast.

- **Gaussian Blur (Optional):** `cv2.GaussianBlur()` if `blur_ksize > 0`. Reduces noise before thresholding.
- **Adaptive Thresholding (Optional):** `cv2.adaptiveThreshold()` if `use_adaptive_thresh` is enabled.
  - `THRESH_BINARY`: Standard adaptive threshold.
  - `THRESH_BINARY_INV`: Inverted adaptive threshold (using `cv2.bitwise_not()`). Useful if barcodes are light on a dark background or if the standard thresholding inverts the barcode.
- **Global Thresholding (Optional):** `cv2.threshold()` with `global_thresh_val` if `use_global_thresh` is enabled.
  - `THRESH_BINARY`: Standard global threshold.
  - `THRESH_BINARY_INV`: Inverted global threshold.
- **Morphological Operations (Optional):** Applied if `morph_kernel_size > 0`.
  - `cv2.MORPH_CLOSE`: To close gaps in barcode bars.
  - `cv2.MORPH_OPEN`: To remove noise.
  - These are typically applied to one of the thresholded images.

3. **Barcode Decoding:**
   - **Pyzbar Integration:** The `decode(image)` function from `pyzbar.pyzbar` is called for each preprocessed image variant.
   - **Unique Detections:** A set (`found_data_hashes`) is used to store a hash of (`data, type, rect`) for each detected barcode to ensure that the same barcode detected through multiple preprocessing variants is only reported once.
4. **Output Visualization:**
   - For each unique decoded barcode:
     - The barcode type (e.g., 'QRCODE', 'EAN13') and data (decoded string) are extracted.
     - A bounding box (`obj.rect`) is drawn on the *original resized color image* using `cv2.rectangle()`. The color of the box is chosen based on the barcode type (defined in `BARCODE_COLORS`).
     - The type and data are displayed as text near the bounding box using `cv2.putText()`. Font scale is adaptively adjusted, and a background rectangle is drawn behind the text for better visibility.

3.3 Module 3: Image Acquisition Utility (`barcode_camera.py`)

This module provides a command-line utility to capture images from a connected webcam.

1. **Camera Initialization:**
   - `cv2.VideoCapture(camera_index, cv2.CAP_DSHOW)` opens the camera. `CAP_DSHOW` is a backend preference for Windows.
   - Desired frame width and height can be set using `cap.set()`.
2. **Real-time Display and Information Overlay:**
   - The camera feed is displayed in a window.
   - Information like current resolution, FPS, and number of images captured in the session is overlaid on the frame using `cv2.putText()` with a helper function `draw_text_on_frame` for backgrounded text.
3. **Capture Mechanisms:**

- **Immediate Capture ('c' key):** Saves the current frame instantly.
- **Countdown Capture (Spacebar):** Initiates a 3-second countdown displayed on screen. The frame is captured when the countdown reaches zero. This allows users to position themselves or the object.

4. **Image Saving:**
   - Captured frames are saved to the `output_folder` with a `filename_prefix` and a timestamp (including milliseconds) to ensure unique names.
   - Images can be saved in `png` (lossless) or `jpg` (lossy, with configurable `jpg_quality`) formats.
   - Messages indicating successful save or errors are displayed.

3.4 Overall System Workflow (Conceptual)

While the modules are separate scripts, a typical workflow would be:

1. Acquire an image (either from a file or using `barcode_camera.py`).
2. (Optional but recommended) Use `barcode_localization.py` to find potential barcode ROIs.
3. For each ROI (or the whole image if localization is skipped/fails), use `barcode_decode.py` to attempt decoding.
4. Display/log the results.

4 Implementation

4.1 Software and Libraries

The project is implemented in Python 3.x and relies on the following key libraries:

- **OpenCV (cv2):** Version 4.x. Used for all core image processing tasks: loading, resizing, color conversion, filtering, gradient calculation, thresholding, morphological operations, contour analysis, drawing, and camera interfacing.
- **NumPy:** Version 1.x. Used for efficient numerical operations, especially array manipulations which are fundamental to OpenCV.
- **Pyzbar (pyzbar.pyzbar):** Used for decoding various barcode symbologies. It's a Python wrapper for the ZBar library.
- **argparse:** Standard Python library for parsing command-line arguments, making the scripts highly configurable.
- **os, time, datetime:** Standard Python libraries for file system operations, timing, and generating timestamps.

4.2 Challenges Faced and Solutions

- **Parameter Tuning:** The performance of both localization and decoding is highly sensitive to the numerous parameters (e.g., kernel sizes, threshold values, filter ranges).
  - **Solution:** Extensive use of argparse makes all critical parameters configurable from the command line. The `debug_show` and `show_intermediate` options were crucial for visualizing the effect of parameter changes during development and tuning. Default values were chosen based on empirical testing on a small set of sample images.
- **Variability in Barcode Appearance:** Barcodes differ in size, orientation (though localization primarily targets horizontal), print quality, and illumination.
  - **Solution:**
    - For localization: CLAHE helps with illumination; adaptive thresholding adapts to local changes; filtering criteria are ranges rather than exact values.
    - For decoding (`barcode_decode.py`): The multi-strategy preprocessing approach (trying CLAHE, blur, various thresholds, morphological ops) significantly increases the chance of successful decoding by Pyzbar under diverse conditions.
- **Distinguishing Barcodes from Similar Structures:** Text lines or other regular patterns can sometimes be falsely detected by the localization module.
  - **Solution:** The combination of multiple strict geometric filters (aspect ratio, solidity, vertices, orientation) in `barcode_localization.py` helps to reduce false positives. However, it's not foolproof without more advanced classification.
- **Pyzbar Limitations:** While good, Pyzbar may fail on very noisy, blurry, or distorted barcodes.
  - **Solution:** The extensive preprocessing in `barcode_decode.py` aims to "clean up" the image as much as possible before handing it to Pyzbar, thereby improving its success rate. The script tries decoding on raw grayscale, CLAHE-enhanced, blurred, and various binarized versions.

5 Results

The implemented system was tested on a variety of images containing different types of barcodes under various conditions. The `barcode_camera.py` utility was used to acquire test images from a live webcam feed.

5.1 Barcode Localization

The localization script was tested with default and tuned parameters.

- **Observations on Localization:**
  - The script performed well on images with relatively clear, horizontally-oriented 1D barcodes.
  - CLAHE and adaptive thresholding were crucial for handling variations in lighting.
  - The morphological closing operation with an appropriately sized wide kernel was effective in merging barcode bars.
  - Filtering based on aspect ratio and solidity significantly reduced false positives from text or other non-barcode structures.
  - Sensitivity to parameters was observed; for challenging images, tuning `adaptive_thresh_block_size`, `adaptive_thresh_C`, `morph_close_kernel_w`, and `min_aspect_ratio` was sometimes necessary.
  - The system sometimes struggled with highly skewed or very small barcodes if they fell outside the filter parameter ranges.

5.2 Barcode Decoding

The decoding script was tested on raw images and images pre-localized by the first script.

- **Observations on Decoding:**
  - The multi-strategy preprocessing significantly improved Pyzbar's success rate compared to using only raw grayscale.
  - For QR codes, CLAHE enhancement or direct grayscale often worked well.
  - For 1D barcodes, especially those with challenging contrast, adaptive thresholding (both normal and inverted) proved beneficial.
  - Morphological operations sometimes helped clean up noisy 1D barcodes before decoding.
  - The ability to toggle preprocessing steps via command-line arguments (`--use_clahe`, `--use_adaptive_t` etc.) was useful for experimentation.
  - The script successfully decoded various types, including EAN13, UPC-A, Code128, Code39, ITF, and QRCODE.

5.3 Camera Capture Utility

- **Observations on Camera Capture:**
  - The utility provided a stable interface for image acquisition.
  - FPS varied depending on the camera and resolution (e.g., 30 FPS at 1280x720).
  - The countdown feature was helpful for positioning items before capture.
  - Saved images were correctly formatted and timestamped.

5.4 Performance Summary (Qualitative)

Overall, the system demonstrated a good capability to locate and decode barcodes under reasonably good conditions. The localization module is well-suited for horizontal 1D barcodes. The decoding module's strength lies in its adaptive preprocessing, enhancing the underlying Pyzbar decoder. Parameter tuning remains key for optimal performance in challenging scenarios.

6 Discussion

6.1 Interpretation of Results

The results demonstrate that a combination of classical Digital Image Processing techniques can effectively address the problem of barcode localization and decoding.

- The **localization module** shows the power of gradient analysis combined with morphological operations for segmenting regions of interest. The subtraction of Y-gradient from X-gradient specifically targets features with strong vertical edges, making it suitable for horizontal 1D barcodes. The sequential filtering based on geometric properties (area, aspect ratio, solidity, vertices, orientation) is crucial for distinguishing barcodes from other image elements.
- The **decoding module** highlights the importance of preprocessing. No single preprocessing technique is universally optimal. By trying multiple strategies (CLAHE, blur, various thresholding methods, morphological touch-ups), the system increases the likelihood that at least one variant will be suitable for the Pyzbar decoder, especially for images with varying quality or barcodes with inverted polarity.
- The **camera utility (`barcode_camera.py`)** serves as a practical tool for image acquisition, completing the pipeline from capture to potential processing.

6.2 Strengths of the System

- **Modularity:** Separate scripts for localization, decoding, and capture allow for independent testing and use.
- **Configurability:** Extensive command-line arguments provide fine-grained control over most DIP parameters, facilitating experimentation and tuning for specific conditions.
- **Robustness through Multi-Strategy Preprocessing (Decoding):** Trying multiple preprocessing methods before decoding significantly enhances the chances of success with Pyzbar.
- **Detailed Localization:** The localization script uses a comprehensive set of morphological operations and filters tailored for barcode-like structures.
- **Clear Visualization:** Options to show intermediate steps and clear final output with bounding boxes and decoded data are invaluable for debugging and demonstration.
- **Practical Application of DIP Concepts:** The project effectively demonstrates various DIP concepts like enhancement (CLAHE), filtering (Gaussian Blur), segmentation (thresholding, contouring), morphological processing, and feature analysis.

6.3 Limitations of the System

- **Sensitivity to Parameters (Localization):** The localization module's performance can be sensitive to its many parameters. Finding a universal set of parameters that works well across all image types is challenging.
- **Limited Orientation Handling (Localization):** The `barcode_localization.py` script is primarily designed for horizontal or near-horizontal 1D barcodes due to the gradient subtraction method and the typical shape of morphological kernels used. It may not perform well on highly rotated 1D barcodes or some 2D barcodes without parameter adjustment.
- **No Advanced Classification:** The localization relies on heuristic geometric filtering. It lacks a machine learning classifier, which could make it more robust in distinguishing barcodes from complex backgrounds or other rectangular objects.

- **Dependence on Pyzbar:** The decoding quality is ultimately limited by the capabilities of the underlying Pyzbar library. Very severely damaged or obscured barcodes might still be undecodable.
- **Computational Cost:** The multi-strategy preprocessing in the decoding module, while effective, increases computation time as decoding is attempted on several image variants. For real-time applications on resource-constrained devices, this might be a concern.
- **Integration:** The three modules are currently separate scripts. A fully integrated application would require a master script or GUI to orchestrate the workflow (e.g., capture -¿ localize -¿ decode).

6.4 Future Improvements

- **Machine Learning Integration:**
  - For localization: Train a lightweight object detection model (e.g., using MobileNet-SSD or a custom CNN) to detect barcode regions more robustly and with better orientation invariance.
  - For preprocessing selection: A simple classifier could potentially predict the best preprocessing chain for a given image/ROI.
- **Enhanced Orientation Handling:** Modify the localization script to detect barcodes at various orientations, perhaps by analyzing gradients in multiple directions or using rotation-invariant features.
- **Adaptive Parameter Tuning:** Explore methods for automatically adjusting key parameters based on image characteristics.
- **Improved Integration and GUI:** Develop a single application with a graphical user interface (GUI) using libraries like Tkinter, PyQt, or Kivy for a more user-friendly experience, integrating camera capture, localization, and decoding.
- **Real-time Optimization:** If targeting real-time video processing, optimize the pipeline, possibly by processing at a lower resolution, skipping some preprocessing steps, or running detection/decoding less frequently (e.g., every Nth frame).
- **Support for More Barcode Types / Advanced Decoding:** Investigate or contribute to libraries for decoding more obscure or damaged barcode types.
- **Error Correction Exploration:** While Pyzbar handles some error correction (especially for QR codes), advanced techniques could be explored for severely damaged 1D barcodes.

7 Conclusion

This project successfully demonstrated the design and implementation of a system for barcode localization and decoding using Digital Image Processing techniques with Python, OpenCV, and Pyzbar. The system effectively localizes primarily horizontal 1D barcodes through a pipeline of gradient analysis, adaptive thresholding, morphological operations, and geometric filtering. Furthermore, it decodes a wide range of 1D and 2D barcodes by employing a multi-strategy preprocessing approach that enhances the robustness of the Pyzbar decoding library. A camera capture utility was also developed to facilitate image acquisition.

The project achieved its objectives by applying core DIP concepts to a practical computer vision problem. The modular design and extensive configurability allow for flexibility and further experimentation. While the system exhibits good performance under many conditions, its limitations, particularly regarding parameter sensitivity and orientation handling in localization, open avenues for future enhancements, notably through the integration of machine learning techniques. This work serves as a solid foundation and a practical demonstration of DIP principles in automated data capture.

8 References

- Gonzalez, R. C., & Woods, R. E. (2018). *Digital Image Processing* (4th ed.). Pearson.
- Kaur, M., & Singh, K. (2017). A Survey of Barcode Recognition System. *International Journal of Computer Applications, 168*(11), 28-32. (You can find similar survey papers for more specific citations).
- OpenCV Documentation. (n.d.). Retrieved from https://docs.opencv.org/
- Pyzbar Documentation (typically found on its GitHub repository or PyPI page). Example: https://pypi.org/project/pyzbar/
- Bradski, G., & Kaehler, A. (2008). *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media.

A Code Listings

A.1 detect_barcode_opencv.py

```
import argparse
import cv2

def debug_show(window_name, image, show_flag, scale_factor=0.5):
    """Helper function to display intermediate images if show_flag is True."""
    if show_flag:
        if image is None:
            print(f"Warning: Image for window '{window_name}' is None.")
            return

        img_to_show = image.copy()
        if img_to_show.dtype != np.uint8:
            if img_to_show.max() <= 1.0 and img_to_show.min() >= 0.0 and
                img_to_show.dtype in [np.float32, np.float64]:
                 img_to_show = (img_to_show * 255).astype(np.uint8)
            else:
                 if img_to_show.min() < 0 or img_to_show.max() > 255 or
                     img_to_show.max() > 1.0: # Heuristic for normalization
                      img_to_show = cv2.normalize(img_to_show, None, 0, 255,
                          cv2.NORM_MINMAX, cv2.CV_8U)
                 elif img_to_show.dtype != np.uint8 : # Handles cases like CV_32F
                     gradient images not scaled
                      img_to_show = cv2.convertScaleAbs(img_to_show)

        if len(img_to_show.shape) == 3 and img_to_show.shape[2] == 1:
             img_to_show = img_to_show.squeeze(axis=2)
        elif len(img_to_show.shape) == 2: # Ensure grayscale is displayable as
            grayscale
             pass # Already fine
        elif len(img_to_show.shape) == 3 and img_to_show.shape[2] == 3: # BGR
             pass # Already fine
        else:
             print(f"Warning: Image '{window_name}' has unexpected shape
                 {img_to_show.shape} or dtype {img_to_show.dtype}. Attempting to
                 display.")

        try:
             # Adjust scale factor if image is already very small
             effective_scale_factor = scale_factor
             if img_to_show.shape[0] < 300 or img_to_show.shape[1] < 300:
                  effective_scale_factor = 1.0 # Don't shrink small images further

             cv2.imshow(window_name, cv2.resize(img_to_show, None,
                 fx=effective_scale_factor, fy=effective_scale_factor,
                 interpolation=cv2.INTER_AREA))
        except cv2.error as e:
             print(f"OpenCV error displaying '{window_name}': {e}")
             print(f"Image shape: {img_to_show.shape}, dtype: {img_to_show.dtype},
                 min: {img_to_show.min()}, max: {img_to_show.max()}")


def detect_barcodes_advanced(image_path, params, show_intermediate_steps=False):
    """
    Detects potential barcode regions in an image using advanced morphological
        operations.
    Focuses on localizing horizontal 1D barcodes.
    """
    print("\n--- Starting Barcode Detection ---")
    print("Parameters in use:")
    for key, value in params.items():
         print(f"  {key}: {value}")
    print("-----------------------------------")

    # --- 1. Load and Preprocess Image ---
    image_orig = cv2.imread(image_path)
    if image_orig is None:
         print(f"Error: Image not found at path {image_path}")
         return None, 0

    print(f"Original image dimensions: {image_orig.shape[1]}x{image_orig.shape[0]}")
    image = cv2.resize(image_orig, None, fx=params['resize_fx'],
```

A.2 barcode_camera.py

```
import os
import argparse
import time
from datetime import datetime

def draw_text_on_frame(frame, text, position, font_scale=0.7, color=(255, 255, 255),
    thickness=1, bg_color=(0,0,0), padding=5):
    """Helper function to draw text with a background on the frame."""
    font = cv2.FONT_HERSHEY_SIMPLEX
    text_size, _ = cv2.getTextSize(text, font, font_scale, thickness)
    text_w, text_h = text_size

    # Add padding to background
    bg_x1 = position[0] - padding
    bg_y1 = position[1] - text_h - padding
    bg_x2 = position[0] + text_w + padding
    bg_y2 = position[1] + padding

    # Ensure background is within frame boundaries (optional, but good practice)
    bg_x1 = max(bg_x1, 0)
    bg_y1 = max(bg_y1, 0)
    bg_x2 = min(bg_x2, frame.shape[1])
    bg_y2 = min(bg_y2, frame.shape[0])

    # Draw background rectangle
    if bg_color is not None:
        cv2.rectangle(frame, (bg_x1, bg_y1), (bg_x2, bg_y2), bg_color, -1)

    # Draw text
    cv2.putText(frame, text, position, font, font_scale, color, thickness,
        cv2.LINE_AA)


def main(args):
    # --- Camera Setup ---
    cap = cv2.VideoCapture(args.camera_index, cv2.CAP_DSHOW) # CAP_DSHOW for
        Windows, might improve performance/stability
    if not cap.isOpened():
        print(f"Error: Could not access camera at index {args.camera_index}.")
        return

    # Set camera resolution (if specified)
    if args.width and args.height:
        cap.set(cv2.CAP_PROP_FRAME_WIDTH, args.width)
        cap.set(cv2.CAP_PROP_FRAME_HEIGHT, args.height)
        print(f"Attempted to set resolution to: {args.width}x{args.height}")

    actual_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
    actual_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
    print(f"Actual camera resolution: {actual_width}x{actual_height}")

    # --- Output Folder ---
    if not os.path.exists(args.output_folder):
        try:
            os.makedirs(args.output_folder)
            print(f"Created output folder: {args.output_folder}")
        except OSError as e:
            print(f"Error creating output folder '{args.output_folder}': {e}")
            cap.release()
            return

    # --- Variables for Loop ---
    fps = 0
    frame_count = 0
    start_time = time.time()

    countdown = -1  # -1 means no countdown active, 3, 2, 1, 0 for countdown
    countdown_timer_start = 0

    capture_message = ""
    capture_message_display_time = 0
```

A.3 barcode_detect_and_decode.py

```
import cv2
from pyzbar.pyzbar import decode
import numpy as np
import os
from datetime import datetime

# --- Configuration for visual output ---
BARCODE_COLORS = {
    "QRCODE": (0, 0, 255),    # Red for QR Codes
    "EAN13": (0, 255, 0),     # Green for EAN13
    "UPCA": (0, 255, 0),      # Green for UPC-A
    "CODE128": (255, 0, 0),   # Blue for Code128
    "CODE39": (255, 0, 0),    # Blue for Code39
    "ITF": (255, 165, 0),     # Orange for ITF
    "DEFAULT": (255, 255, 0)  # Cyan for others
}
FONT = cv2.FONT_HERSHEY_SIMPLEX
FONT_SCALE_BASE = 0.5
FONT_THICKNESS = 2

def preprocess_image(image, args):
    """
    Applies a series of preprocessing steps to the image to enhance barcode
        detection.
    Returns a list of images to try decoding on.
    """
    processed_images = {} # Dictionary to store different processed versions

    # 0. Original Resized Grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    processed_images['gray'] = gray.copy()
    if args.show_intermediate:
        cv2.imshow("0. Grayscale", gray)

    # 1. CLAHE (Contrast Limited Adaptive Histogram Equalization)
    if args.use_clahe:
        clahe_cv = cv2.createCLAHE(clipLimit=args.clahe_clip_limit,
            tileGridSize=(args.clahe_tile_size, args.clahe_tile_size)) # Renamed to
            avoid conflict
        clahe_gray = clahe_cv.apply(gray)
        processed_images['clahe'] = clahe_gray.copy()
        if args.show_intermediate:
            cv2.imshow("1. CLAHE", clahe_gray)
        current_gray = clahe_gray # Use CLAHE output for subsequent steps if enabled
    else:
        current_gray = gray

    # 2. Gaussian Blur (Optional, can help reduce noise before thresholding)
    if args.blur_ksize > 0:
        blurred = cv2.GaussianBlur(current_gray, (args.blur_ksize, args.blur_ksize),
            0)
        processed_images['blurred'] = blurred.copy()
        if args.show_intermediate:
            cv2.imshow("2. Blurred", blurred)
        current_gray_for_thresh = blurred
    else:
        current_gray_for_thresh = current_gray


    # 3. Adaptive Thresholding
    if args.use_adaptive_thresh:
        adaptive_thresh = cv2.adaptiveThreshold(current_gray_for_thresh, 255,
                                                cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                                cv2.THRESH_BINARY,
                                                args.adaptive_thresh_block_size,
                                                args.adaptive_thresh_c)
        processed_images['adaptive_thresh'] = adaptive_thresh.copy()
        if args.show_intermediate:
            cv2.imshow("3a. Adaptive Threshold", adaptive_thresh)

        # Inverted adaptive threshold
```