Institute of Visualization and Interactive Systems
Department of Intelligent Systems
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 2844

Robust 1D Barcode Recognition on Mobile Devices

Johann C. Rocholl

Course of Study:	Computer Science
Examiner:	Prof. Dr. G. Heidemann
Supervisor:	Dipl.-Inf. S. Klenk
Commenced:	October 28, 2008
Completed:	April 29, 2009
CR Classification:	C.5.3 Portable devices G.1.2 Least squares approximation I.4.3 Geometric correction I.4.7 Feature measurement

Abstract

This thesis describes a novel method for decoding linear barcodes from blurry camera images. It can be used on mobile devices to recognize product numbers from EAN or UPC barcodes and retrieve information from the Internet without input from the user.

Conventional decoders for linear barcodes are based on detecting the edges between bars and spaces. However, the locations of these edges may change or become undetectable if the input is very blurry. This is a problem for mobile devices with fixed-focus lenses. Their cameras are not designed to focus correctly in the macro range required for capturing barcodes. The Apple iPhone is a well-known example of such a device.

The proposed algorithm locates the barcode in the camera image and extracts a scan line of brightness values. It simulates the blurry barcode according to a mathematical model and chooses digits for which the simulation best approximates the camera input. The parameters of the simulation are adjusted automatically with an iterative approach.

A prototype was implemented to recognize UPC-A and EAN-13 barcodes on the Apple iPhone and on the MacBook, using their built-in cameras. The decoder was tested with several hundred images from different cameras. The proposed method correctly recognizes a high percentage of blurry barcode images. The performance of the prototype is also compared to four different existing decoders for linear barcodes, with good results.

Acknowledgments

I would like to thank my thesis supervisor Sebastian Klenk for his guidance and encouragement, and Prof. Dr. Gunther Heidemann for taking the time to answer my questions and ask new questions. Special thanks go to Marta Johnson for proofreading my English, and taking good care of me and our baby. I thank my friends Bloom, Georg, Lecci, Magnus, Martin and Matze for letting me stay at their house in Stuttgart while I was working on this thesis.

I am also grateful to Arndt Kritzner and Steven Thielemann of Logic Way GmbH in Schwerin for providing access to their test images, as well as test results from their decoder with my images, and for pointing out the problem of successful but incorrect recognition.

The OpenCV library was used for experimentation and implementation of the prototype. For my work on the source code and the thesis document, I used the open-source programs Emacs, GCC, LaTeX 2_ε, Python, Gnuplot, GNU Make and Bazaar. Thanks to the authors and the open-source community for producing such great software and helpful documentation.

Contents

List of Tables

List of Figures

1 Introduction

Barcodes are a ubiquitous technology for machine-readable data. Producing barcodes is as cheap and fast as monochrome printing, and they can be recognized quickly and optically with relatively simple hardware. This chapter provides some background information and explains the motivation and challenges for decoding blurry barcodes with mobile devices.

1.1 Linear barcodes

Linear barcodes have become widely used since they were invented in the 1970s. There are several well-known standards. Figures 1.1 and 1.2 present some examples. They were generated with the Barcode Writer in Pure PostScript by Burton (2009).

Almost all retail products worldwide are labeled with a barcode using either the "European Article Number" (EAN) or "Universal Product Code" (UPC) standard. They are collectively called GTIN (Global Trade Identification Numbers). Most EAN barcodes have 13 digits, but there are also compact versions with fewer digits for small products. UPC-A has 12 digits, being a subset of EAN-13 with the first digit implicitly set to zero. The structure of the barcode is exactly the same for UPC-A and EAN-13.

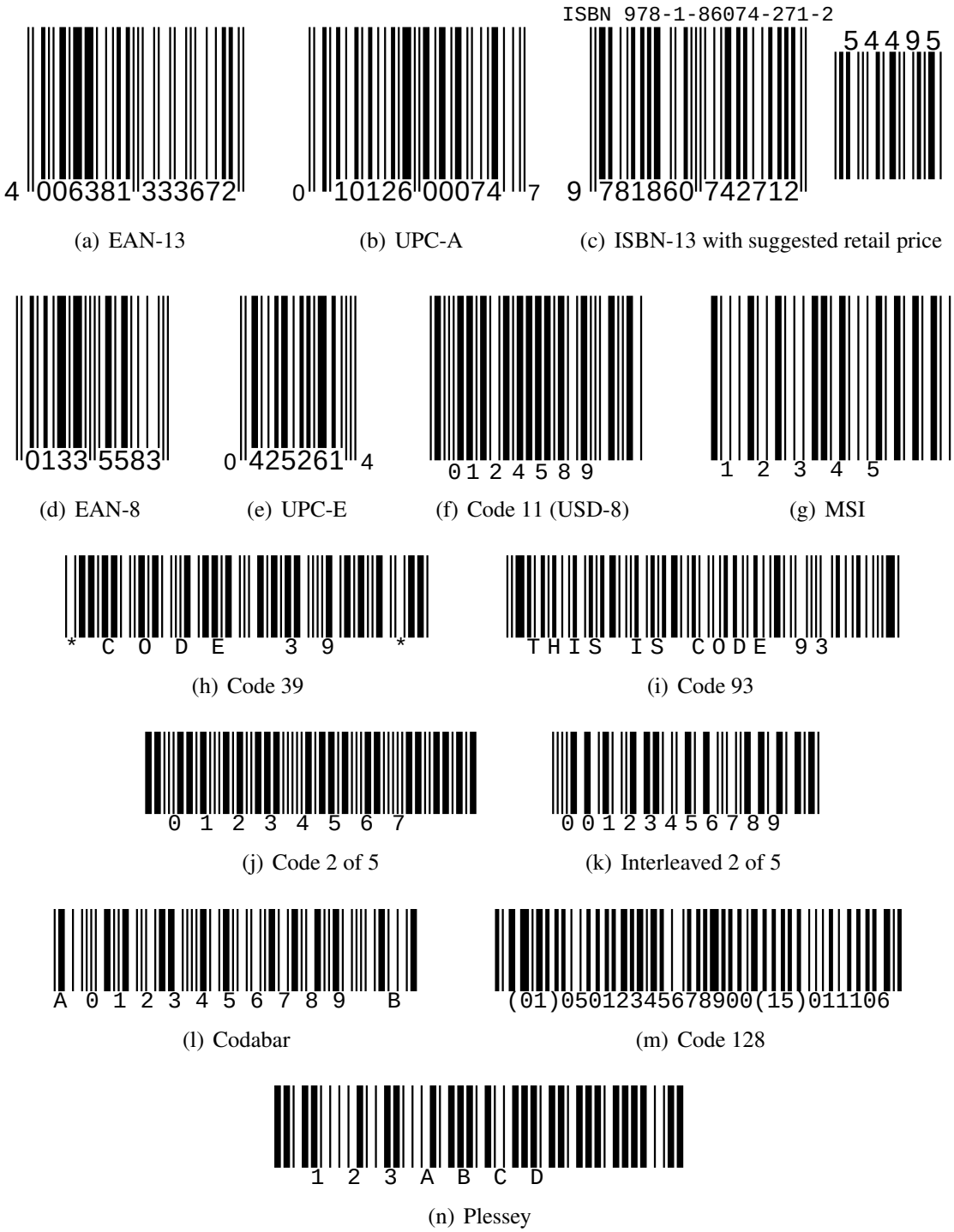
Most published books have an "International Standard Book Number" (ISBN). This standard was published in 1970 as ISO 2108 and used 10 digits. Since the beginning of 2007, ISBN are encoded as EAN-13 with the prefix 978. The ISBN barcode is often accompanied by an EAN-5 code on the right to indicate a suggested retail price.

1.2 Motivation

Modern mobile devices (smartphones) typically include a camera and a mobile Internet connection. This enables a new class of services related to online product lookup, if the product number can be recognized from a camera image of the barcode. The most obvious use is to check the price of a product in the store, to see if a similar product is available in a different store or online at a lower price. With location (using built-in GPS or wireless tracking) it is possible to restrict search results to stores in the same area.

Several online resources provide product reviews from customers or professional reviewers. If customers had easy access to these reviews in the store, it would help them better choose the right product for their needs. Persons with allergies or dietary restrictions could benefit from detailed lists of ingredients for processed food items.

Barcodes can also be found on electronic media like CDs and DVDs. Using a wireless broadband Internet connection, customers could download movie trailers or listen to music previews on their mobile devices to decide if they want to buy the disc.



(a) EAN-13 (b) UPC-A (c) ISBN-13 with suggested retail price

(d) EAN-8 (e) UPC-E (f) Code 11 (USD-8) (g) MSI

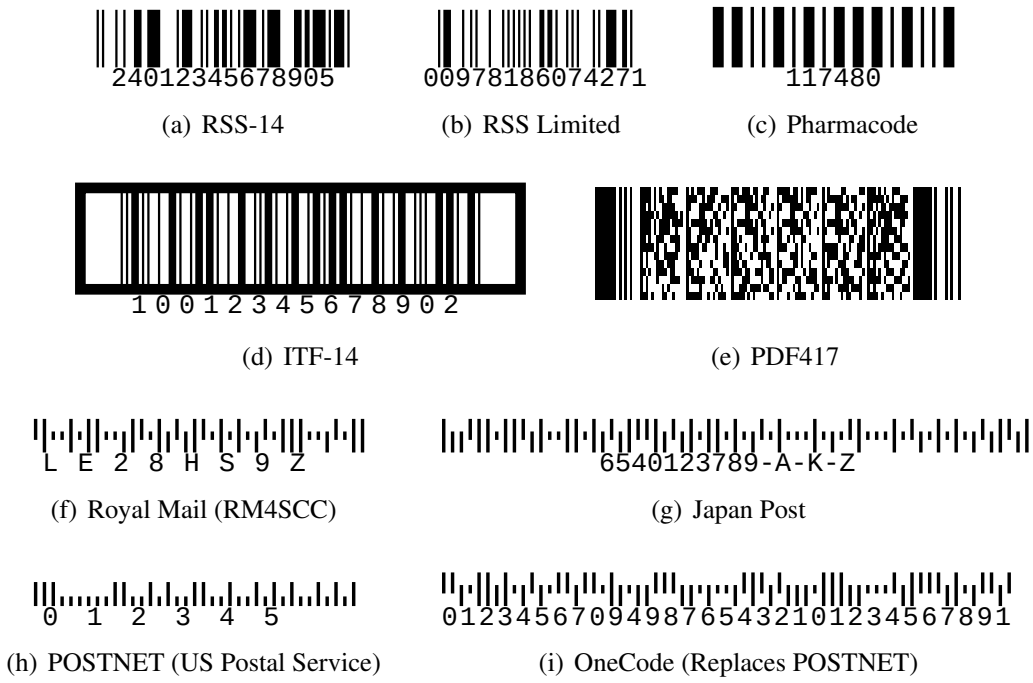
(h) Code 39 (i) Code 93

(j) Code 2 of 5 (k) Interleaved 2 of 5

(l) Codabar (m) Code 128

(n) Plessey

Figure 1.1: Linear barcode symbologies



(a) RSS-14 (b) RSS Limited (c) Pharmacode

(d) ITF-14 (e) PDF417

(f) Royal Mail (RM4SCC) (g) Japan Post

(h) POSTNET (US Postal Service) (i) OneCode (Replaces POSTNET)

Figure 1.2: Linear barcode symbologies, continued

The technology can also be used for shopping lists and wish lists. Users could scan an empty milk carton at home and see a reminder on the mobile device to buy milk when they get to the store. Or products could be added to a wish list by scanning their barcodes in the store. Friends and family could view this wish list online and purchase birthday presents accordingly.

1.3 Structure of the EAN-13 barcode

The EAN-13 barcode consists of thirty parallel vertical bars (dark stripes) separated by spaces (bright stripes). The width of the thinnest bars is called the module size. Each of the bars and spaces is exactly one, two, three or four modules wide. The total width of the barcode is 95 modules, and each module represents one bit. A dark bar denotes a logical one, and a bright space denotes a logical zero. The first and last three bits are always 101, and the five bits in the middle are always 01010. These are called “guard bars” and used for synchronizing the decoder.

13 digits are encoded by the EAN-13 barcode, but only 12 digits are represented explicitly as barcode characters. The first digit is printed outside the barcode area and is specified by the use of two alternative codes for the 6 digits between the left and middle guard bars. Every explicit digit occupies seven of the 95 bits in the barcode and is represented by exactly two bars and two spaces. The codes for each digit are shown in table 1.1. They are defined by the GS1 General Specifications (GS1, 2009, page 194).

(a) Three possible codes for each digit

Digit	Code A	Code B	Code C
0	0001101	0100111	1110010
1	0011001	0110011	1100110
2	0010011	0011011	1101100
3	0111101	0100001	1000010
4	0100011	0011101	1011100
5	0110001	0111001	1001110
6	0101111	0000101	1010000
7	0111011	0010001	1000100
8	0110111	0001001	1001000
9	0001011	0010111	1110100

(b) Codes for the extra digit

d_1	d_2 d_7	d_8 d_{13}
0	AAAAAA	CCCCCC
1	AABABB	CCCCCC
2	AABBAB	CCCCCC
3	AABBBA	CCCCCC
4	ABAABB	CCCCCC
5	ABBAAB	CCCCCC
6	ABBBAA	CCCCCC
7	ABABAB	CCCCCC
8	ABABBA	CCCCCC
9	ABBABA	CCCCCC

Table 1.1: Digit codes for the EAN-13 standard

The last digit is a checksum. It can be used for automatic error detection and correction, as described in section 5.2. The mandatory bright areas on the left and right side of the barcode are called quiet zones.

1.4 Challenges

Mobile devices have limited processing power and memory, because of size constraints and battery usage. Apple has not released official specifications for the iPhone's CPU and RAM. According to an online article by Hockenberry (2007), the iPhone has an ARM processor running with a clock rate of 400 or 600 MHz and 128MB of system RAM (which is separate from the 4, 8, or 16 GB of slower flash memory used for storage). This seems sufficient for simple computer vision tasks, in combination with the performance benefits of running native code written in Objective-C, rather than Java which uses a virtual machine.

Many cameras used in mobile devices, including the iPhone, have a fixed focus lens. They are not designed for macro photos and produce blurry pictures if the distance between camera and object is less than 20cm. EAN-13 barcodes are usually printed less than 5cm wide. If the camera is distant enough to avoid macro blur, the barcode will appear too small and the resolution will be too low for decoding. Existing applications for mobile barcode scanning (e.g. ShopSavvy or the PDA barcode scanner from Logic Way GmbH) require a camera with automatic focus or a separate macro lens.

Industrial barcode scanning solutions use controlled lighting, usually with lasers. This is not possible with mobile devices, as they may not provide a light or flash for the camera. If they do, the light is located close to the camera and produces reflections that complicate the barcode recognition. Therefore it is necessary to use ambient lighting and to deal with non-uniform illumination in the decoder software.

Figure 1.3: Image of ISBN-13 barcode, captured with the iPhone

Figure 1.3 shows an example of an ISBN-13 barcode that was captured with the iPhone's built-in camera. The camera blur is severe: bars and spaces are blurred together, forming ranges of varying brightness. The edges between bars and spaces cannot be detected. The image also contains some perspective distortion because the camera was not exactly in front of the center of the barcode. The brightness of the image increases slightly from left to right, and there is a bright reflection in the top right corner. This barcode image is part of the comparison test suite in appendix A. It could not be decoded by any of the existing decoders described in section 2.5. However, it was decoded correctly with the method proposed in this thesis, using the prototype decoder implementation.

2 Related work

This chapter provides a survey of general techniques and more specialized approaches for the challenges of locating and decoding blurry barcodes.

A concise introduction to barcode technology was written by Pavlidis et al. (1990). It gives a good overview of common barcode types, information density, scanning noise and distortions, and error correction.

The official specification for the EAN barcode symbologies and several others can be found in the GS1 General Specifications (GS1, 2009), available from local GS1 member organizations. The document also describes a reference decoder algorithm which is based on measuring the widths of the bars and spaces.

2.1 Locating the barcode

2.1.1 Blocks with consistent gradients

In a paper by Shams and Sadeghi (2007) the barcode is located by computing the gradient directions in the input and then finding blocks where the gradient direction is consistent. These blocks are first connected to find the region of interest (ROI), then the quiet zones next to the barcode are detected. This approach is successful even at very low resolutions. I implemented this algorithm for the thesis prototype, see section 4.1.

2.1.2 Hough transform

A patent by Hough (1962) describes a method for detecting patterns by transforming lines to slope-intercept representation. The method was refined by Duda and Hart (1972) to use angle-radius parameters instead. This avoids the problem of an unbounded transform space. I considered the Hough transform as a method for detecting the orientation of barcodes in the blurry camera image, but it is better suited for black and white bitmaps than for blurry gray images.

2.1.3 Canny edge detector

A famous edge detector by Canny (1986) was designed to be optimal under explicit requirements of good detection, localization, and minimal response. It works in four stages: input blurring for noise reduction, computation of gradient intensities and directions, non-maximum suppression, and edge tracing. The Canny detector fails to locate all bars if the input image is very blurry, but the first two stages were used for the implementation of the block-wise voting approach in section 2.1.1 above.

2.2 Distortion and lighting

2.2.1 Distorted geometry

Xu et al. (2007) propose a skew distortion correction method for 2D barcode images. The paper derives the transformation matrices based on two vanishing points that are detected from the the two sets of parallel lines in the barcode. The focus of their work is on 2-dimensional QR codes, but the modeling of perspective adjustment works for linear barcodes too.

2.2.2 Non-uniform illumination

A paper by Kim and Lee (2007) describes an iterative method for finding the background brightness across the image, while at the same time detecting the width and distance of the bars, even with some blurring. Their objective function is based on the proper parameterization of a barcode signal and illumination. The illumination is regularized using a smoothness penalty. The mathematical model for my thesis was partly inspired by this paper, especially the illumination estimation, and the use of a Gaussian kernel to simulate blurring.

2.3 Deblurring

2.3.1 Peaks

Joseph and Pavlidis (1992) present a decoder that operates on the locations of the peaks of the barcode waveform because peaks are more immune to convolution distortion (blurring) than edges. The peaks are used as estimates of the midpoints of the dark stripes. They also show that points of high curvature are still present when the blurring is has already removed separate peaks. I have considered similar methods based on the first or second derivative of the brightness across the barcode for this thesis, but preliminary testing showed that they are very susceptible to noise in the digital camera image.

2.3.2 Filters

Another paper by the same authors (Joseph and Pavlidis, 1993) presents a method for processing closely spaced edges and accurately restoring their locations. The algorithm considers sets of three interacting edges, forcing the effects of two edges to cancel each other, so that the position of the third edge can be detected more precisely. The paper presents different methods to reconstruct edges and estimate the point spread function, then compares different deblurring filters based on these techniques.

2.3.3 Blind deconvolution

Esedoglu (2004) presents a method for blind deconvolution of barcode signals based on minimizing the total variation. Blurring is modeled with a Gaussian kernel, and the iterative algorithm adjusts the blur kernel parameters. An implementation by Wittman (2004) processes each image in six minutes in MatLab on a Celeron with 2.4 GHz clock rate, so this method is not directly usable for an interactive mobile application. My thesis follows a similar approach but uses more a priori knowledge about the barcode, particularly the limited set of possible codes for each digit.

2.4 Decoding

2.4.1 Selective sampling

Shellhammer et al. (1999) analyze the derivative of the signal from a laser scanner to determine the strength of bar edges. Spurious edges are eliminated with Otsu's threshold. The paper also describes edge enhancement filters for improving the detection. These techniques are used in a commercial product called "Fuzzy Logic Scanner" by Symbol Technologies.

2.4.2 Hierarchical method

A hierarchical approach by Lu et al. (2006) combines several image filters and feature extractors to process different kinds of uncertain conditions. The algorithm starts with a barcode quality assessment based on omni-directional scan lines, then selects methods to improve the results, including contrast adjustment, morphological filters, bi-linear interpolation. It can recover barcode symbols from partly occluded images.

2.4.3 Statistical recognition

In a detailed paper by Wang et al. (2007), the barcode region is located with a wavelet transformation and then segmented into characters. The segmentation is based on the zero crossings of the second derivative of the brightness values in the input. A feature vector is extracted for each character. The recognition engine selects characters with the smallest recognition distance, using a modified Generalized Learning Vector Quantization (GLVQ) method. The classifier is trained with more than 1000 barcode images. The feature vector classification method was already published in an earlier conference paper by the same authors (Wang et al., 2005). My thesis follows a similar approach of character segmentation and selection, but explores alternative techniques for each of the steps.

2.4.4 Special hardware

Ohbuchi et al. (2004) present a method for locating and decoding EAN and QR Codes. Their algorithm uses spiral scanning to detect linear barcodes and performs an inverse perspective transformation. It can process camera images at more than five frames per second on a mobile phone, but it requires a special DSP chip.

2.5 Existing applications

This section lists some barcode scanning libraries and applications that are actively developed. Section 7.2 discusses comparison test results for all of the decoders used in the following solutions. The detailed results and some example images are shown in appendix A.

2.5.1 ZXing

ZXing (Owen et al., 2009, pronounced “zebra crossing”) is an open-source, multi-format 1D/2D barcode image processing library implemented in Java. It supports EAN-13, UPC-8 and several other 1D and 2D barcodes. There’s also a port to Objective C for the iPhone, but it supports only QR Code.

2.5.2 ShopSavvy

ShopSavvy (Barnes et al., 2009) is a shopping assistant developed for Google’s Android mobile phone platform. It runs on T-Mobile’s G1 phone produced by HTC and uses the ZXing library to decode barcodes. The recognition is fast and has a high success rate because the camera uses automatic focus during the barcode scan. The application supports online lookup for price comparison and product reviews.

2.5.3 PDAbar library

Logic Way GmbH in Schwerin (Kritzner, 2008) produces barcode reader software for PDA devices. It recognizes several different linear barcodes, including EAN-13 and Code 39. It has been commercially available since 2005. The supported platforms include PocketPC, Windows Mobile, Symbian, and recently Android (Java). The decoder requires a camera with automatic focus or a macro lens to reduce image blur.

2.5.4 Delicious Library

Delicious Library (Shipley, 2009) is a product management system for Mac OS X. It includes a barcode scanner that uses the Mac's built-in iSight camera to recognize UPC, EAN and ISBN barcodes. The product website explains that the algorithm uses the vector processor to perform image deconvolution because the camera images are out of focus. Among all decoders in the comparison, Delicious Library had the best results for blurry barcodes.

2.5.5 ItemShelf

ItemShelf (Murakami et al., 2009) is an open-source iPhone application that recognizes product barcodes with the built-in camera. It requires a macro lens and uses the ZBar barcode reader library by Brown (2009). The application uses Amazon Web Services to look up the recognized products, as does the Delicious Library above.

3 Blurry barcode model

In order to guess barcode digits, the blurry barcode must be simulated according to a mathematical model. The results of the simulation can then be compared to the camera input to find matching digits. This chapter describes the model and the parameters of the blurry barcode simulation.

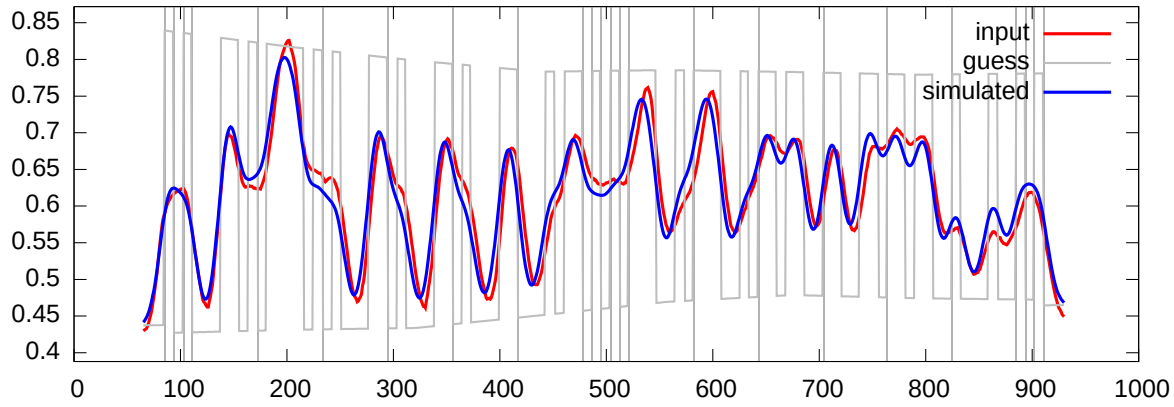


Figure 3.1: Input and output of the blurry barcode simulation

3.1 Continuous scan line

After the barcode has been located, processing can be restricted to monochrome brightness values on a scan line across the barcode area. The region of interest (ROI) is an interval from one quiet zone to the other. Its endpoints are denoted by l_{ROI} and r_{ROI} respectively. For example, $l_{\mathrm{ROI}} = 65$ and $r_{\mathrm{ROI}} = 930$ in figure 3.1.

$$X_{\mathrm{ROI}} = [l_{\mathrm{ROI}}, r_{\mathrm{ROI}}] = \{x \in \mathbb{R} \mid l_{\mathrm{ROI}} \leq x \leq r_{\mathrm{ROI}}\}$$

The waveform of brightness values on the scan line can be expressed as a continuous function over the region of interest. Several similar functions are introduced throughout this chapter. They all have the same domain and codomain, and are denoted by f with a subscript. Here the subscript "cam" stands for the camera input. It is shown as the red curve in figure 3.1.

$$f_{\mathrm{cam}} : X_{\mathrm{ROI}} \to [0,1]$$

Brightness is expressed in the range from zero to one, with 0.0 designating bright white (spaces) and 1.0 designating black (bars). This definition is consistent with common usage in related work, where bars appear as peaks.

3.2 Barcode position

The bit pattern for UPC and EAN barcodes consists of 95 bits: seven bits for each of the 12 digits, and 11 bits for the guard bars (three bits on the left, five bits in the middle, three bits on the right). The simulator algorithm attempts to find the left and right edges of the barcode inside the region of interest, denoted by l and r without subscript.

$$l_{\mathrm{ROI}} < l < r < r_{\mathrm{ROI}}$$

The function s denotes the estimated positions of the 95 bits between the edges of the barcode. The actual definition of this function will be presented at the end of the next section.

$$s : [0, 95] \to [l, r]$$

The function s is expected to be smooth and strictly monotonous, going from the left to the right edge of the barcode, so $s(0) = l$ and $s(95) = r$. The function values for integer arguments represent the boundaries between bits. For example, the boundary between the left guard bars and the first digit would be found at $s(3)$.

3.3 Perspective projection

If the camera axis is not orthogonal to the barcode surface, one side of the barcode may be closer to the camera and thus appear larger. This results in horizontal offset for the positions of the bars and digits, because the bars appear wider on the near side and narrower on the far side.

We model the perspective projection in a Cartesian coordinate system with the origin at the middle of the barcode, see figure 3.2. The projection plane is the x axis. For the purpose of this subsection, the barcode area is normalized so that $x = -1$ indicates the left side and $x = 1$ indicates the right side of the barcode.

Let d denote the distance between the camera and the center of the barcode, or more precisely the ratio between the camera distance and half the width of the barcode. The value of d is unchanged if both the camera distance and the size of the barcode are doubled. In figure 3.2 the camera is placed on the y axis at $y = -d$.

Next we define a parameter $u \in \mathbb{R}$ that is linear across the barcode and goes from $u = -1$ at the left side of the barcode to $u = 1$ at the right side. Using the angle α between the barcode and the projection plane, the coordinates of the interior points of the barcode can be expressed like this:

$$x_u = u \cos \alpha \qquad\qquad y_u = u \sin \alpha$$

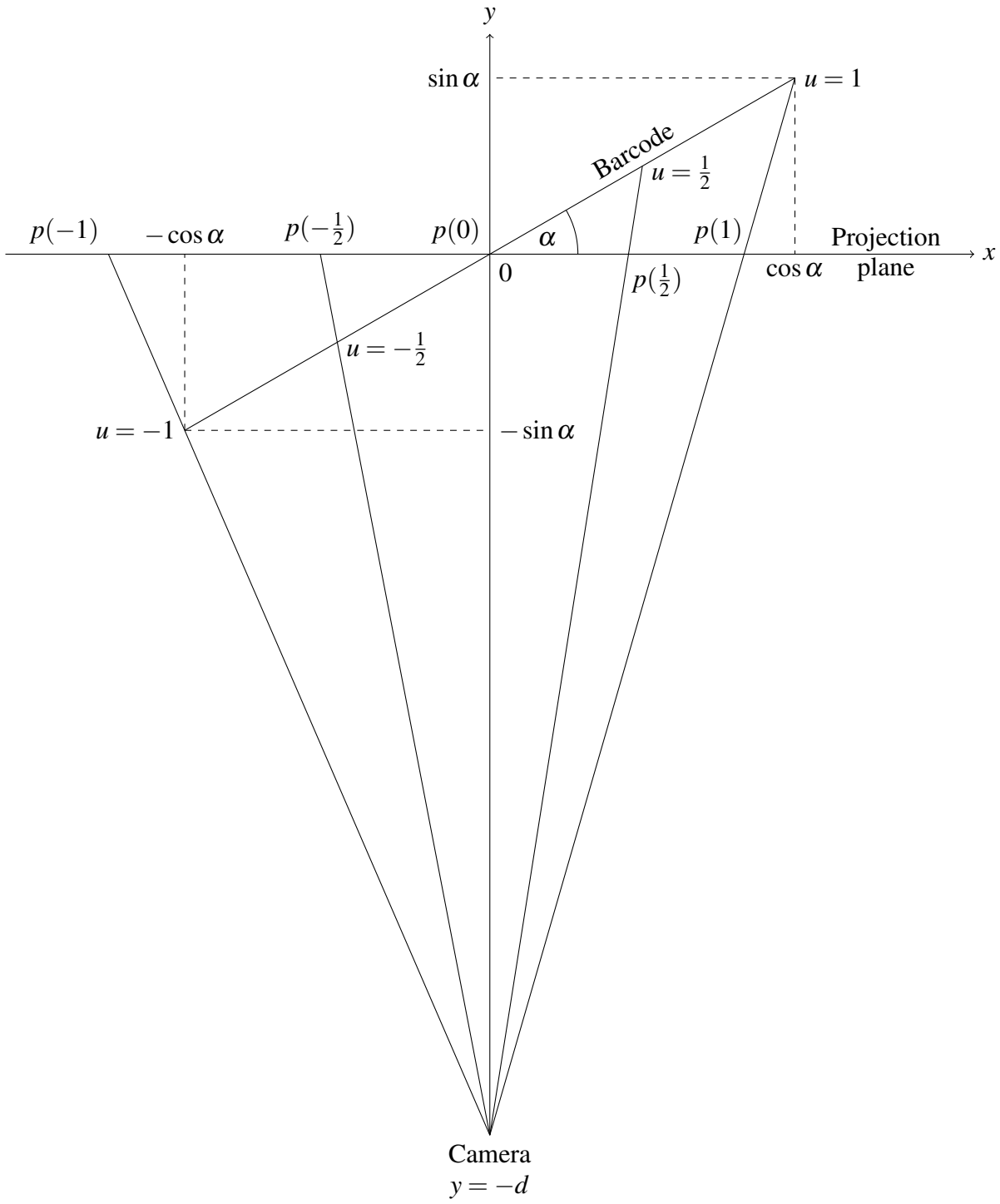


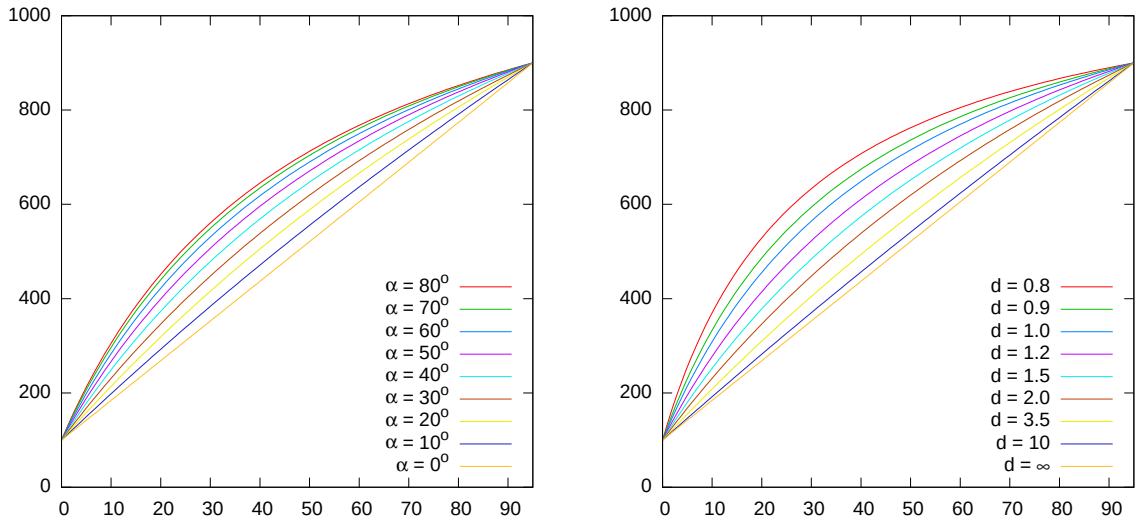
Figure 3.2: Perspective projection

Now we can formulate the camera ray as a straight line through the camera and a barcode point. The x-intercept of this line gives the projection point $p(u)$. It is the abscissa of the intersection of the camera ray and the projection plane.

$$y = \frac{d + u\sin\alpha}{u\cos\alpha}x - d \qquad\qquad p(u) = \frac{d \cdot u\cos\alpha}{d + u\sin\alpha}$$

To use this projection for estimating the locations of bits between the edges of the barcode l and r, we need a normalized projection function s that satisfies $s(0) = l$ and $s(95) = r$. The following definition achieves that by scaling the function argument to the interval $[-1,1]$ and the result to $[l,r]$:

$$s(i) = l + (r-l)\frac{p(-1+\frac{2}{95}i) - p(-1)}{p(1) - p(-1)}$$



(a) Constant distance $d = 2.0$

(b) Constant angle $\alpha = 30°$

Figure 3.3: Graphs of $s(i)$ for various camera angles and distances

3.4 Non-uniform illumination

The background brightness and contrast may change across the region of interest. This can be modeled by two continuous functions for the white and black level across the scan line, called f_{white} and f_{black} respectively. The barcode guess (before blurring) alternates between the values of these two functions, according to the bit pattern.

$$f_{\text{guess}}(x) = \begin{cases} f_{\text{black}}(x) & \text{for high bits} \\ f_{\text{white}}(x) & \text{for low bits and outside the barcode area} \end{cases}$$

A good initial estimate for the white level is a straight line that passes through the brightness values of the quiet zones on the left and right side of the barcode. For the black level, the current implementation uses a constant (horizontal) line at the 95th percentile of brightness values in the barcode area. These initial estimates are then refined iteratively, as described in section 5.3.4 and shown in figure 5.5.

3.5 Gaussian blur

Typical camera blur can be simulated by convolution with a Gaussian kernel. The standard deviation parameter σ controls the width of the blur kernel.

$$G : \mathbb{R} \to \mathbb{R}^+ \qquad\qquad G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}$$

The camera blur can be simulated by a convolution of the guessed bit pattern with the Gaussian kernel.

$$f_{\text{sim}} = G * f_{\text{guess}}$$

$$f_{\text{sim}}(x) = \int_{-\infty}^{\infty} G(\tau) \cdot f_{\text{guess}}(x-\tau)\, d\tau$$

The Gaussian function is always nonzero, even when $|x|$ is very large, so a perfect blurring kernel would have to be twice as wide as the input array. But in practice, the values of $G(x)$ are negligibly small for $|x| > 3\sigma$. For performance, we take samples from the Gaussian kernel and store them in a linear array of float values, represented here as a vector $\boldsymbol{g}$. The function f_{sim} can then be approximated using these discrete samples.

$$\boldsymbol{g}_i = G(i-k-1) \qquad 1 \le i \le 2k+1 \qquad k = \lceil 3\sigma \rceil$$

$$f_{\text{sim}}(x) \approx \sum_{i=-k}^{k} \boldsymbol{g}_{i+k+1} \cdot f_{\text{guess}}(x-i)$$

When the Gaussian function is sampled at a limited number of points, the sum of all values will be smaller than one. This would lead to a change of brightness during the convolution with image data. This problem can be avoided by normalization: each value in the kernel vector is divided by the sum of all values.

3.6 Residual

For each digit, the difference is computed between the result of the simulation and the brightness values from the camera input, using the sum of squares:

$$e_i = \int_{l_i}^{r_i} (f_{\text{sim}}(x) - f_{\text{cam}}(x))^2 dx$$

The values l_i and r_i are the estimated locations of the edges between digits, based on the function s that models perspective projection. The additive parts of the function arguments are the widths of the guard bars (3 modules for the left guard and 5 modules for the middle guard.)

$$l_i = \begin{cases} s(7i+3) & \text{for } 1 \leq i \leq 6 \\ s(7i+8) & \text{for } 7 \leq i \leq 12 \end{cases}$$

$$r_i = \begin{cases} s(7i+10) & \text{for } 1 \leq i \leq 6 \\ s(7i+15) & \text{for } 7 \leq i \leq 12 \end{cases}$$

4 Locating the barcode

The first step for locating the barcode is to find a straight line that intersects all bars. It does not have to be perpendicular to the bars, but it must go across the entire barcode.

One possible solution is to display this line through the middle of a live camera preview and require the user to move the camera and/or target until the line intersects all barcode stripes. This approach has two advantages. First and foremost, the user experience benefits from the visual clue and live feedback because it is immediately obvious how the barcode should be captured. In addition, decoding is faster because the processing of gradients can be omitted and we can start with tracing (see section 4.3). I used this approach for the iPhone prototype.

4.1 Position and orientation

If no assumptions can be made about the position and orientation of the barcode, the following algorithm works well, even on severely blurred input images. The block voting is based on an idea presented by Shams and Sadeghi (2007). My implementation uses the first two processing stages of the well-known Canny edge detector (Canny, 1986) to compute the gradient intensities and directions.

```
function locate_barcode(input_image):
    # Gaussian smoothing to reduce the effects of noise.
    smooth_image = convolve(input_image, gaussian_kernel)
    # Horizontal and vertical Sobel (Scharr) operator.
    sobel_image_h = convolve(smooth_image, sobel_kernel_h)
    sobel_image_v = convolve(smooth_image, sobel_kernel_v)
    # Intensity and direction of gradients.
    intensity_image = intensities(sobel_image_h, sobel_image_v)
    direction_image = directions(sobel_image_v, sobel_image_v)
    # Block voting for best gradient direction.
    blocks_image = vote_blocks(intensity_image, direction_image)
    best_quality = 0
    best_center = point(0, 0)
    best_direction = 0
    for direction from 0 to 7:
        quality, center = find_barcode_area(blocks_image, direction)
        if quality > best_quality:
            best_quality = quality
            best_center = center
            best_direction = direction
    return best_center, best_direction
```

The image is subdivided into blocks of 8×8 pixels. In each block, the dominant gradient direction is determined by counting pixels that fall in each of the first eight gradient directions in figure 4.0(b). The opposite directions are considered the same, using modulo-8 arithmetic. To reduce the effects of noise, a pixel with discrete direction 13 actually counts twice for direction 5, and once for each of directions 6 and 4. The gradient detection and block voting algorithms are implemented in `gradients.c`.

(a) 8 directions

3	2	1
4	•	0
5	6	7

(b) 16 directions

6	5	4	3	2
7				1
8		•		0
9				15
10	11	12	13	14

Table 4.1: Discrete directions

For each of the directions 0 to 7 in 4.0(b), the block image is searched for a large rectangular connected area that would represent the barcode. This is implemented by following several parallel lines perpendicular to the gradient direction, and finding the longest section where the dominant block direction varies only slightly. Overlapping sections in consecutive parallel scan lines are combined to form candidate barcode areas. The largest area with the highest gradient intensity is selected as the barcode location.

4.2 Adaptive threshold

An adaptive threshold is used at two distinct places in the algorithm. For bar tracing (section 4.3), it processes each horizontal row of raw byte samples in the monochrome camera image to produce a threshold image with the same dimensions. The other use is for detecting peaks and valleys for exact calibration of perspective projection (section 4.9). This step is performed after sub-pixel sampling, so the algorithm works on a linear array of floating point numbers in this case. An example graph of the threshold function is shown in figure 4.1.

The threshold is the result of a low-pass filter, computed directly from the input array as a sliding average. The implementation is very efficient because it maintains the sum of input values in the sliding window, adding the new value on the right and subtracting the oldest value on the left in each step. The sliding window shrinks at both ends of the input array, but that detail is not shown in the equation for simplicity.

$$f_{\text{avg}}(x) = \frac{1}{2r+1} \sum_{v=x-k}^{x+k} f_{\text{cam}}(v)$$

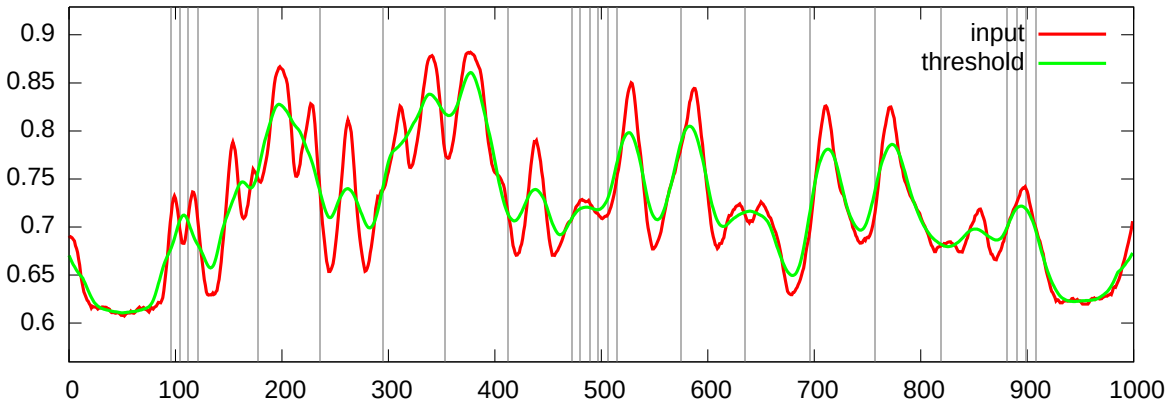


Figure 4.1: Adaptive threshold

The parameter k is the radius of the sliding window. A good value is $\frac{1}{60}$ of the width of the input array. Because EAN-13 barcodes have exactly 30 bars, this window spans roughly one edge with adjacent peak and valley, so the threshold curve crosses the input roughly in the middle between top and bottom.

4.3 Bar tracing

After a straight line has been found that intersects the entire barcode, we can trace the bars up and down from this line to find the top and bottom edges of the barcode. These edges are required to estimate perspective distortion, so that it can be compensated. The monochrome input image can be considered as a height map over a two-dimensional terrain. Let darker pixels denote ridges, and brighter pixels valleys.

The adaptive threshold from section 4.2 is used to find the brightest and darkest points along the scan line that crosses the barcode. These points are the starting points for tracing the valleys and ridges in both directions.

The following algorithm traces a valley from a starting point. It is simplified here, but works similarly for dark pixels (ridges) and in the other direction (y increasing). Example results of the tracing are shown on the left side of the images in figure 4.2. Red and blue lines represent valleys and ridges respectively.

If the main search direction is not parallel to the y axis, the eight discrete directions in table 4.0(a) can be used to compare pixels. The algorithm below is just a special case for direction 2. For example, if the line across the barcode roughly follows the 3-7 direction, the main search direction is 1, and the algorithm compares the pixels at position 0, 1 and 2 to find the darkest pixel for the next step.

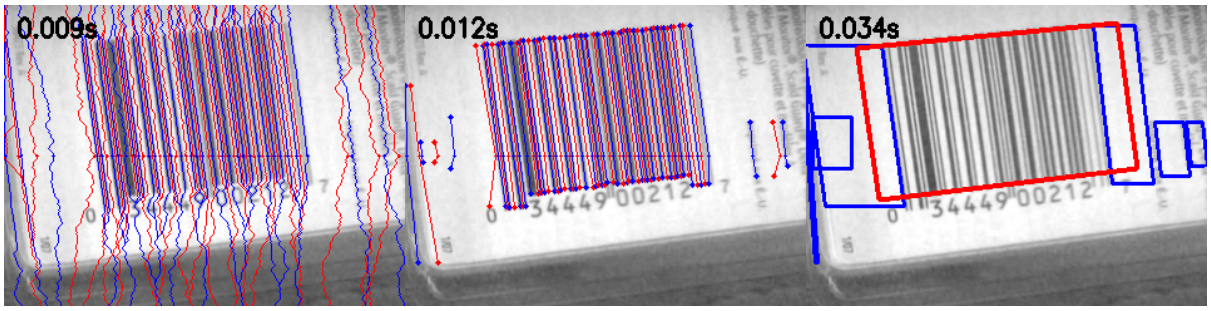
```
function trace_valley(input_image, x, y, trace_array):
    while y > 0:
        y = y - 1
        brightest_x = x
        if input_image[x - 1, y] > input_image[brightest_x, y]:
            brightest_x = x - 1
        if input_image[x + 1, y] > input_image[brightest_x, y]:
            brightest_x = x + 1
        x = brightest_x
        trace_array[y] = x
```

4.4 Endpoints of the bars

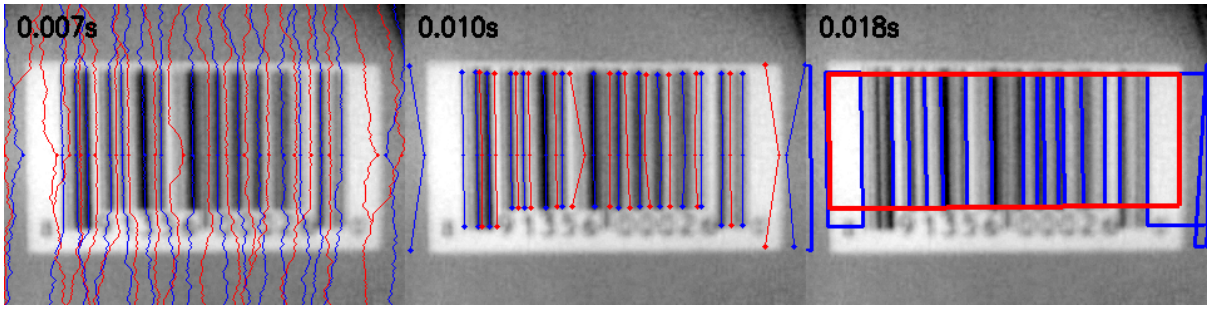
To find the endpoints of the bars and spaces, the pixel brightness of the traces are compared to both of their neighbor traces. If the middle trace is dark (valley), then the neighbors are bright (ridges) and vice versa. At the end of the bars, the contrast between the bright and dark traces disappears. However, this change may be gradual over several pixels because of camera blur. The algorithm presented below uses a circular buffer array of eight values to store the recent "history" of contrast values. The endpoint is detected if the current contrast is less than two thirds of the contrast that was detected seven steps earlier.

```
function find_endpoint(left_trace, middle_trace, right_trace, y):
    contrast[8] = [0, 0, 0, 0, 0, 0, 0, 0]
    while y > 0:
        contrast_left = abs(left_trace[y] - middle_trace[y])
        contrast_right = abs(right_trace[y] - middle_trace[y])
        contrast[y mod 8] = max(contrast_left, contrast_right)
        if 3 * contrast[y mod 8] < 2 * contrast[(y + 7) mod 8]:
            break
        y = y - 1
    return y
```

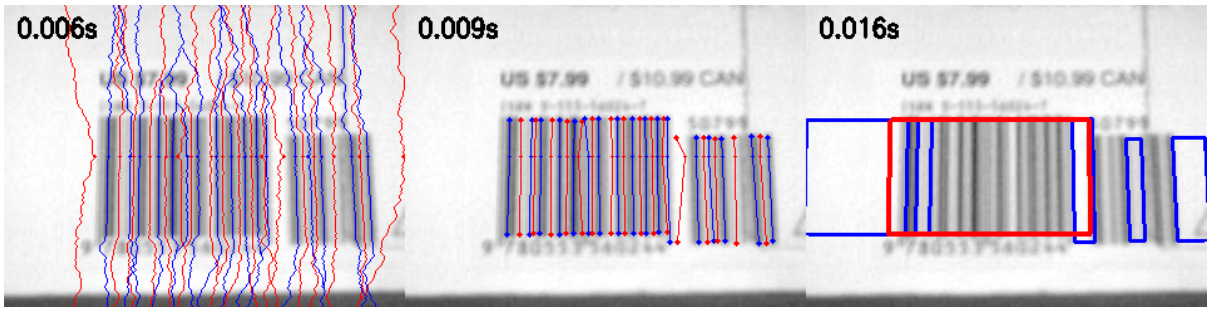
This function is called twice for each bar to find the endpoints, going up or down from the starting point in the middle. Bright bars are removed if the cross neighboring dark bars before the endpoint is reached. Finally, bars are trimmed if they exceed the length of both neighbors. The resulting bars with endpoints are shown in the middle images in figure 4.2. Red and blue lines represent light and dark bars respectively.



(a) Relatively clear barcode, captured on iPhone with macro lens



(b) Blurry barcode, captured on iPhone without macro lens



(c) Blurry ISBN-13 barcode with EAN-5 supplement

Figure 4.2: Processing steps for locating the barcode

4.5 Quiet zones

The mandatory bright areas on the left and right side of the barcode are called quiet zones. They are required for the decoder to find the beginning and end of the barcode area. The GS1 General Specifications (GS1, 2009, page 188) require 11 modules on the left and 7 modules on the right for an EAN-13 barcode, and 9 modules on each side for UPC-A.

After the dark bars have been traced, the algorithm attempts to find quiet zone candidates, looking on the left side of each dark bar for the left half of the image, and on the right side for the right half. The barcode area is searched among all combinations of quiet zone candidates. Every quiet zone on the left side is matched with each quiet zone on the right, and a quality value is computed for this pair. This search has quadratic complexity, but the number of quiet zone candidates is generally less than 10 on each side.

The results of the quiet zone detector are shown on the right side of figure 4.2. Quiet zone candidates are represented by blue frames. The red frame indicates the four corners of the barcode area. The method for finding these corners is described in the next section.

4.6 Robust corners

The four corners of the barcode are estimated from the endpoints of the traces with a robust method based on the medians of x and y positions. It is implemented in the source file `corners.c`. Each bar has a floating point value between zero and one which designates the horizontal ratio of its starting point compared to the first and last bar.

Because the bars are parallel, these ratios are preserved when going from the middle to the end points, even under perspective projection. Each pair of endpoints on the top edge generates an estimate for the top left and top right corners by projecting the line that passes through both endpoints and finding the corners at ratio zero and one.

The medians of all x and y estimates are used as the coordinates of the corners. This method is robust because it uses many estimates and ignores outliers.

4.7 Camera distance and angle

The camera distance d is calculated from w_{barcode}, the length of the scan line across the barcode, compared to w_{image}, the length of the diagonal of the entire input image. It requires a measured camera angle γ for the diagonal. For example, this angle is 46° for the prototype that uses a half image of the iPhone camera. Here the values x_1, x_2, y_1 and y_2 denote the horizontal and vertical coordinates of the endpoints of the scanline across the barcode.

$$w_{\text{image}} = \sqrt{x_{\text{image}}^2 + y_{\text{image}}^2} \qquad w_{\text{barcode}} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

$$d = \frac{1}{\tan\beta} \qquad \beta = \gamma \frac{w_{\text{barcode}}}{w_{\text{image}}}$$

The camera angle α is estimated from the ratio between the heights of the left and right sides of the barcode. These heights h_l and h_r are measured perpendicular to the top and bottom edges from the middle of the left and right edges of the barcode area. The angle is zero if the heights h_l and h_r are identical. It increases if the left side h_l appears larger than the right side h_r. The factor 1.2 is an adjustment constant that was determined experimentally.

$$\alpha = \arctan(1.2 \log \frac{h_l}{h_r})$$

4.8 Robust sampling

For digital processing, the continuous input scan line is sampled at discrete points, and a binary number is stored for each point. This discretization occurs already in the image sensor in the digital camera. For improved accuracy, the scan line is re-sampled from the image pixels at sub-pixel intervals with bilinear interpolation, and the brightness values are converted from 8-bit unsigned char (byte) values to 32-bit floating point representation. The result is a linear array that can be indexed by natural numbers.

Digital camera images always contain some level of noise, and sometimes the sampling line will suffer from printing inaccuracies or specks of dirt. The robustness can be improved by sampling several parallel lines and taking the median of all brightness values from corresponding samples for each index. Figure 4.3 shows an example for the results of robust sampling from 20 parallel lines.

4.9 Exact calibration

If the size of the camera blur is smaller than the distance between bars, there will be separate peaks for each bar in the gray value input array. In this case, it is possible to distinguish all 30 bars of the EAN-13 or UPC-A barcode. Their locations can be used to determine exactly the left and right edges of the barcode, and to calibrate the camera angle for perspective projection. This functionality is implemented in the source file `thirty.c`.

The peak detector uses an adaptive threshold array, which is computed exactly as in section 4.2, but here the input is taken from the array representing the camera input f_{cam}, which was generated from the raw camera data with sub-pixel sampling.

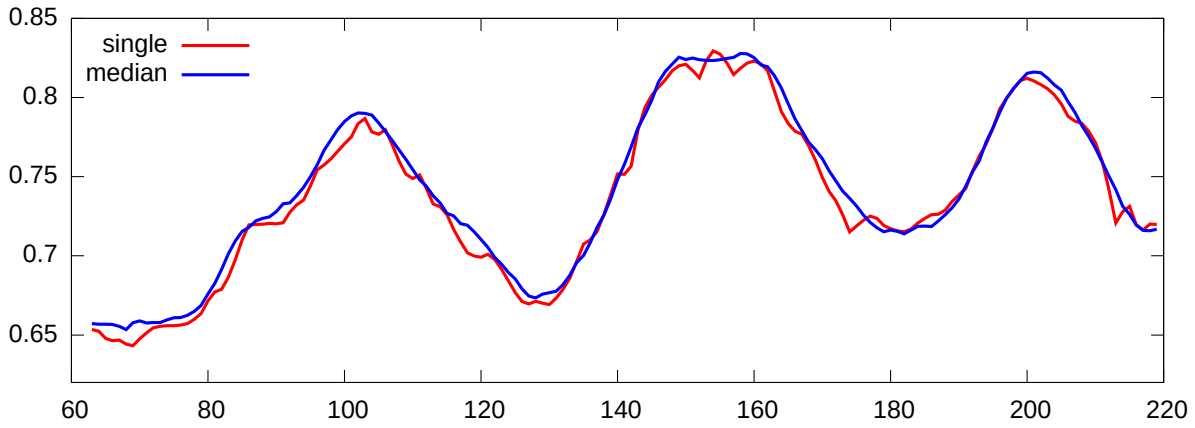


Figure 4.3: Sampling improvement by median of parallel lines

Peaks p_i and valleys v_i are located as the highest or lowest value between the intersections of the input and the threshold curve. If 30 peaks and 29 valleys are found, the locations of the left, right and middle guards (double bars) can be determined. For precision, we use the average of the locations of all peaks and valleys in each guard.

$$l_{\text{guard}} = \frac{p_1 + v_1 + p_2}{3} \approx v_1$$
$$m_{\text{guard}} = \frac{v_{14} + p_{15} + v_{15} + p_{16} + v_{16}}{5} \approx v_{15}$$
$$r_{\text{guard}} = \frac{p_{29} + v_{29} + p_{30}}{3} \approx v_{29}$$

The total number of modules in an EAN-13 barcode is 95, and the width of the left and right guard is 3 modules. So the expected position of the center of the left guard is 1.5 modules, and the distance between the centers of the outer guards is 92 modules. We can now extrapolate the exact positions of the left and right edge of the barcode.

$$l = l_{\text{guard}} - 1.5 \cdot \frac{r_{\text{guard}} - l_{\text{guard}}}{92}$$

$$r = r_{\text{guard}} + 1.5 \cdot \frac{r_{\text{guard}} - l_{\text{guard}}}{92}$$

If the barcode was captured perpendicular to the camera axis (no perspective distortion), the center of the middle guard is exactly half-way between the outer edges. But if it falls left or right, the camera angle can be determined with the bisection method, using the function $q(u)$ as presented in section 3.3. The position variables (start, middle, stop, attempt) are indices for the input array, so their type is integer. The bisection method produces one bit of accuracy in each iteration, and therefore converges in less than 10 iterations if the input contains 1000 samples.

```
function camera_angle(start, middle, stop, distance):
    lower = -60.0 # angle degrees
    upper = 60.0 # angle degrees
    for counter from 1 to 10:
        angle = (lower + upper) / 2
        projected = s(distance, angle, 95 / 2)
        if projected < middle:
            lower = angle # Search again in the right half.
        else if projected > middle:
            upper = angle # Search again in the left half.
        else: # Found the correct angle.
            break
    return angle
```

The exact calibration presented here produces very good decoding results because it provides good estimates for the locations of the edges between digits. Unfortunately, it works only if all 30 peaks and 29 valleys can be found in the input array. If the camera blur is severe, the blur kernel is larger than the distance between bars, and the distinct peaks and valleys are joined together. In this case, the camera angle must be estimated from the four corners of the barcode.

5 Decoding

The traditional method for decoding linear barcodes is to find the edge locations between bars and spaces. This approach fails when the blurring is severe, i.e. when the standard deviation of the blurring kernel is greater than the module size (the width of the thinnest bars or spaces). In this case, the separate peaks and valleys in the gray value curve disappear. The input consists of varying shades of gray, rather than black bars and white spaces. Even if the inverse of the blurring kernel could be found, the presence of noise makes it impossible to derive the original barcode from the blurry picture. Because the edge locations cannot be determined reliably, a different approach is required for robust decoding.

This work proposes to estimate the boundaries of the 12 digits in the barcode, and to guess each digit separately, as there are only 10 possible choices for each digit in a UPC barcode. EAN-13 uses two possible codes for the third through seventh digit to specify the extra first digit, so there are 20 possible choices. But many invalid guesses can be eliminated because there are only 10 valid combinations, see table 1.0(b).

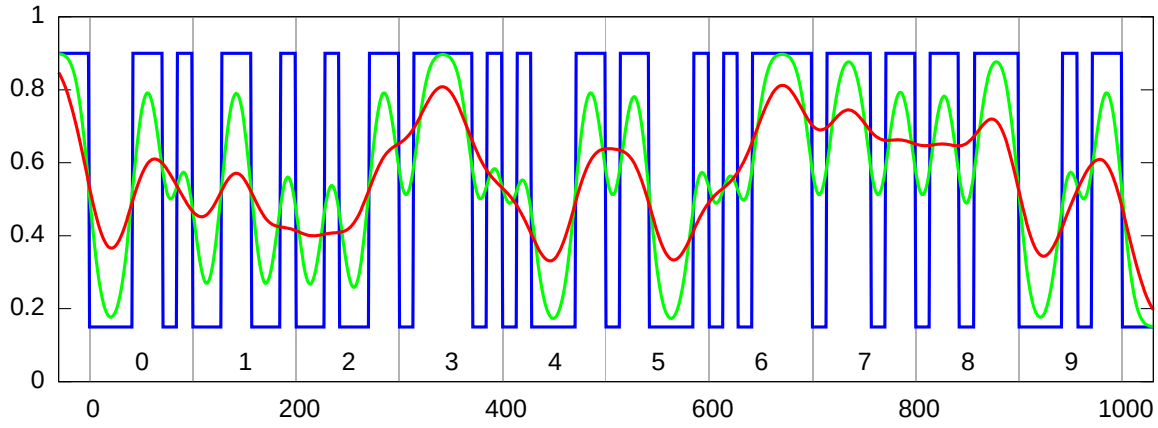
5.1 Guessing digits

After the parameters of the barcode model have been estimated, we can attempt to guess each of the digits by setting the corresponding seven bits, blurring with a Gaussian kernel, and comparing the result with the input.

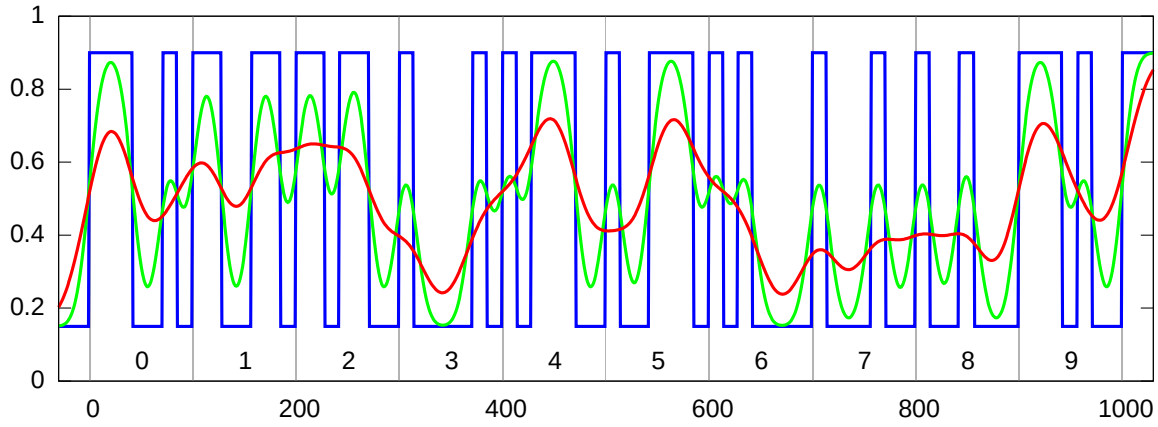
Figure 5.1 shows the three possible codes for all digits from 0 to 9. The blue graph is the bit pattern, showing the seven code bits for each digit. The module size (width of one bit) is roughly 14 samples in these graphs because each digit occupies 100 samples.

The green and red curves in figure 5.1 are the results of Gaussian blurring with a standard deviation parameter of $\sigma = 10$ samples and $\sigma = 20$ samples, respectively. This demonstrates the effects of blurring: if the standard deviation is smaller than the module size, the distinct peaks and valleys are preserved, but they are joined together for larger amounts of blur.

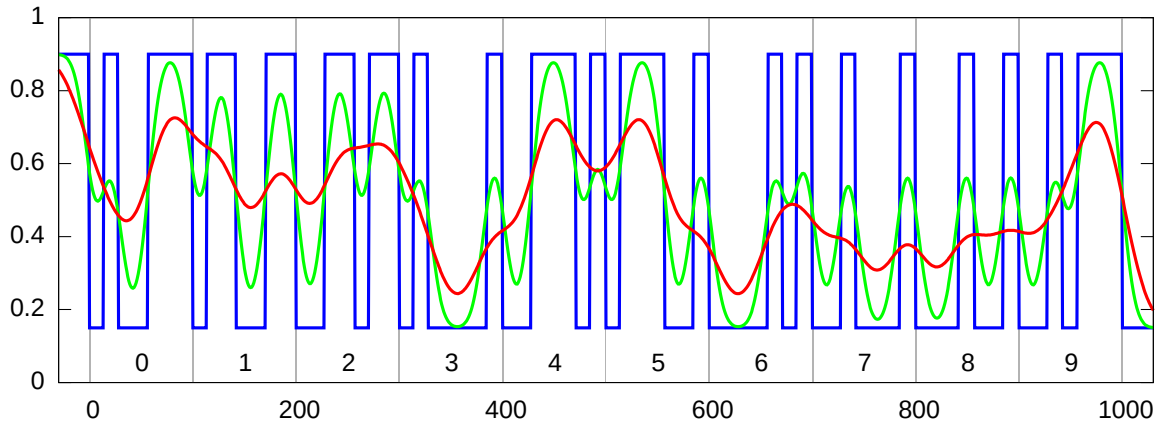
The blurring requires additional input on the left and right of each digit, half the width of the Gaussian kernel. Some of these bits may be unknown because those neighboring digits have not been guessed yet. But the left, right and middle guard bars are known, as well as the first and last bit in each digit. The six digits on the left side of the middle guard all start with a space (white) and end with a bar (black), and conversely on the right side. The inner five bits can be set to gray, as the average between the black and white brightness levels. The result of this is shown in figure 5.2. It is even better (but less obvious when plotted) to copy the input gray values as initial guesses for the inner five bits. For iterative adjustment, each additional pass can use the bits that were guessed in the previous pass.



(a) A code



(b) C code: upside-down mirror image of A code



(c) B code: left-to-right mirror image of the C code for each digit

Figure 5.1: Three possible codes for each digit

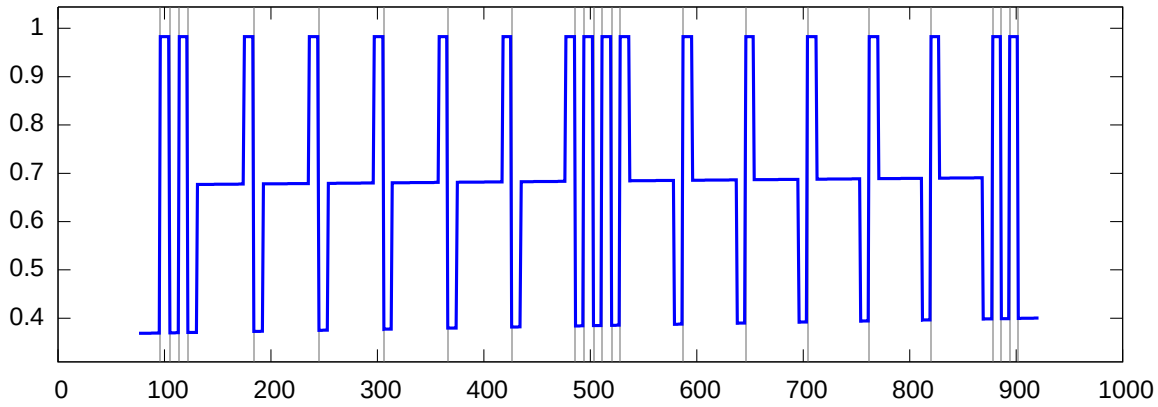


Figure 5.2: Known bits before guessing digits

5.2 Error detection and correction

5.2.1 Checksum

The last digit of the EAN-13 (and thus also UPC-A) is a check digit that can be used to detect when a barcode was decoded incorrectly. The checksum is calculated as a weighted sum of all other digits modulo 10, with different weights for the digits at odd and even positions.

$$d_{13} \equiv 10 - \sum_{i=1}^{6} (3d_{2i-1} + d_{2i}) \mod 10$$

If only one digit is incorrect, there will be a checksum mismatch. If several digits are incorrect, it is possible that the checksum will report a false match because the separate errors may cancel each other out. But statistically, 90% of incorrect barcodes will be detected.

5.2.2 Codes A and B

The limited number of choices for the first digit of the EAN-13 barcode can also be used to reject invalid combinations of A and B codes for the digits d_3 to d_7. The possible combinations are listed in table 1.0(b). For example, there is no case where d_3 to d_7 would all use the B code, so that result would have to be incorrect. Some of the digits would need to be changed to generate a valid result.

5.2.3 Error correction

The result of digit guessing can be invalid because of a checksum mismatch or because the codes for the first digit do not match one of the expected combinations. In this case, the error correction algorithm will find a valid result by making few changes with the smallest possible additional error between the input and the simulated barcode.

The digit guessing algorithm produces two outputs: a best match for each digit, and a list of possible changes. Each of these changes specifies the position of the digit, the alternative digit value, and the additional error compared to the best match. To find the best valid result, we would first have to test all combinations of changes for validity, then we could select the combination with the smallest additional error out of all the valid ones.

Let n denote the length of the list of possible changes. It is bounded from above by the total number of choices for digits in the barcode. There are 20 choices for each of the digits d_3 to d_7, and 10 choices for each of the other digits.

$$n \leq 5 \cdot 20 + 7 \cdot 10 - 12 = 158$$

Let k denote the number of changes to consider together. More than 12 changes need not be considered because a digit would be changed more than once, but the error would be smaller for only one change. The number of alternative barcodes for a fixed k can be computed with the binomial coefficient:

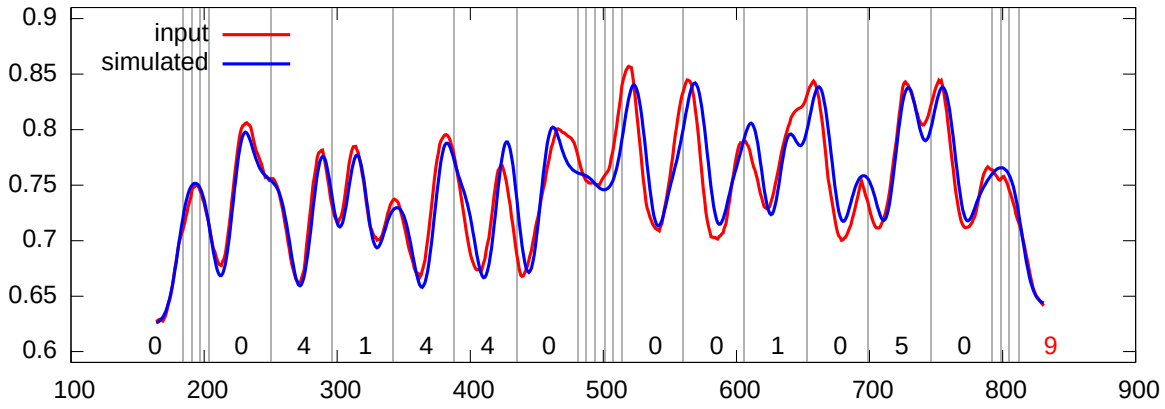
$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

If we consider alternative barcodes with $1 \leq k \leq m$ changes (m for maximum number of simultaneous changes), we get the following summation:

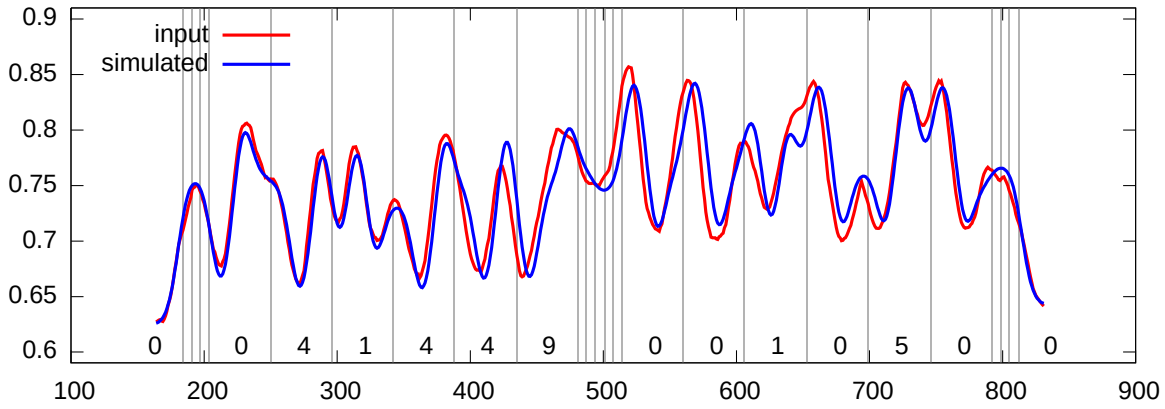
$$\sum_{k=1}^{m} \binom{n}{k} = \sum_{k=1}^{m} \frac{n!}{k!(n-k)!}$$

For $n = 158$ and $m = 12$, there are more than $3 \cdot 10^{17}$ alternative barcodes. The magnitude of this number makes an exhaustive search impossible. Therefore we restrict the search to a subset where the best solution will likely be found.

Most of the 158 alternatives will have large additional errors because they do not match the input very well. We can remove these useless entries by sorting the array of alternatives with the C standard library function `qsort`, and truncating it to a reasonable length. The current implementation considers only the best 20 alternatives.



(a) Before error correction



(b) After successful error correction

Figure 5.3: Successful error correction

Additionally, the maximum number of changes can be reduced because the errors add up, so more changes will produce higher additional error. The current implementation considers combinations of up to three changes. To find the best barcode with up to three changes out of 20, we must check 1350 combinations for validity. This is possible with a fast recursive algorithm, presented below in pseudocode.

```
function best_valid_barcode(changes_array, max_changes, digits):
    if the digits are valid:
        return 0.0
    for each change in changes_array:
        apply change to the input digits
        error = change.error
        if max_changes > 1:
            error += best_valid_barcode(
                rest of changes_array, max_changes - 1, digits)
        if the digits are valid and error < best_error:
            best_error = error
            best_digits = digits
    digits = best_digits
    return best_error
```

Figure 5.3 shows an example of successful error correction. Because of perspective distortion, the digits in the middle of the barcode have some horizontal offset. The seventh digit was decoded incorrectly, as shown in the first graph (a). The computed checksum digit (9) is displayed in the bottom right corner and colored red, because it is different from the last decoded digit (0). The second graph (b) shows the same barcode after the error was corrected automatically.

5.3 Iterative adjustment

The correctness of the digit guessing results depends heavily on accurate estimates for the following parameters of the blurry barcode model: the left and right edge of the barcode, the camera distance and angle for perspective distortion, the width of the Gaussian blurring kernel, and the black and white brightness levels.

It is difficult to estimate these parameters directly, because they are influenced by each other, and also by the decoded digits. For example, changing the amount of blur would also influence the left and right edge positions of the barcode, and incorrect digits produce different illumination levels. But it is possible to start with rough estimates and then refine the parameters with an iterative approach. The digit guessing (and error correction) is performed three times.

```
function decode_iterative(input):
    simulator.input = input
    estimate_parameters(simulator) # Initial rough estimates.
    guess_digits(simulator) # First attempt.
    correct_errors(simulator)
    adjust_parameters(simulator, 50%) # Careful.
    guess_digits(simulator) # Second attempt.
    correct_errors(simulator)
    adjust_parameters(simulator, 100%) # Confident.
    guess_digits(simulator) # Final attempt.
    correct_errors(simulator)
```

The percentage argument to `adjust_parameters` controls the amount of adjustment. Fifty percent means to take the average of the old and new estimate. This helps reduce errors if the first pass of `guess_digits` produces too many wrong digits. The `correct_errors` function call performs automatic error detection and correction based on the checksum digit, see section 5.2.

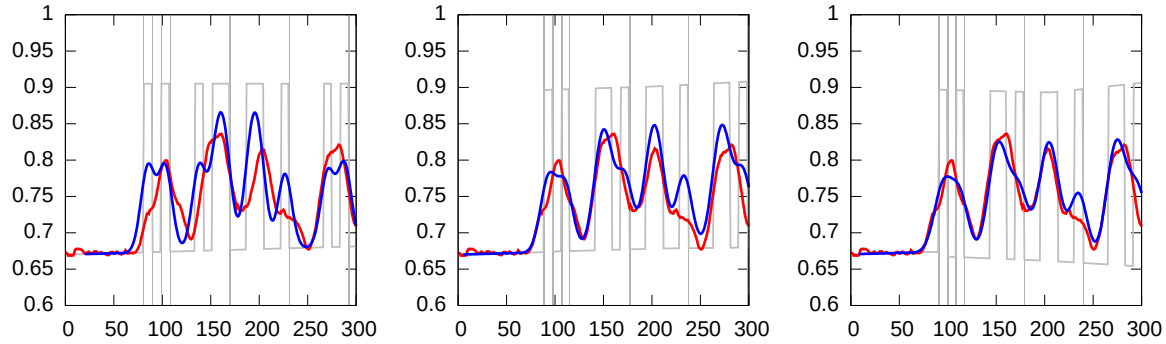
It would be possible to adjust the parameters in smaller steps and repeat the digit guessing until the parameters become stable. However, the test suite indicates that three passes are sufficient in most cases, and this seems to be a good trade-off between runtime and accuracy.

5.3.1 Position

The positions of the left and right edge of the barcode can be adjusted with the "ordinary least squares" (OLS) method. It is applied by moving the simulated barcode slightly left and right, and finding the offset with the smallest sum of squared deviations between the input and the simulated gray values for the area around the guard bar on each side.

```
function adjust_start(input, simulator, start):
    lowest_error = sum_of_squares(input, simulator, offset=0)
    best_offset = 0
    for offset from -10 to 10:
        if offset == 0:
            continue
        error = sum_of_squares(input, simulator, offset)
        if error < lowest_error:
            lowest_error = error
            best_offset = offset
    return start + best_offset
```

An example of the iterative adjustment of the start position is shown in figure 5.4. Again, the input brightness curve f_{cam} is plotted in red, the guessed bit pattern f_{guess} in gray, and the result of the simulation f_{sim} in blue. The light gray vertical bars indicate the left guard bars and the edges between digits. The left edge is estimated at $l = 81$ for the first pass and subsequently adjusted to $l = 89$ and then $l = 91$.



(a) First pass: initial estimate (b) Second pass: adjusted (c) Third pass: adjusted again

Figure 5.4: Iterative adjustment of left side and blur kernel size

5.3.2 Gaussian blurring

The width of the blur kernel is adjusted in a similar manner by varying the standard deviation parameter σ in small steps and finding the least sum of squared deviations over the entire barcode area. Examples for this adjustment are shown in figures 5.4 (σ increasing) and 5.5 (σ decreasing).

5.3.3 Perspective

The camera distance and angle are not adjusted in the current version of the prototype because it may actually deteriorate the decoding. If the input image contains little blur (standard deviation of Gaussian kernel smaller than module size), the angle can be calibrated exactly, see section 4.9. In this case, there is no need for adjustment. But if the blur is severe, the position of the middle guard bars generally cannot be found because they do not correspond with distinct peaks and valleys. If the digits adjacent to the middle guard are not guessed correctly in the first attempt, comparing the input with the simulation does not provide a good indication of the central guard position.

5.3.4 Non-uniform illumination

The f_{black} and f_{white} arrays are adjusted by comparing the extreme values of the input and the simulated barcode. The barcode area is divided into seven segments, and adjustments are computed for each of the eight boundaries between segments by finding the highest and lowest brightness values in the adjacent segments. The 5th and 95th percentile of the brightness values are used to ignore very narrow spikes.

```
function adjust_brightness(input, simulator, black, white):
    array adjust_white[0 to 7]
    array adjust_black[0 to 7]
    # Find adjustments for boundaries.
    for boundary from 0 to 7:
        # Consider segment from previous to next boundary.
        start = boundary_index(max(0, boundary - 1))
        stop = boundary_index(min(7, boundary + 1))
        # Compute 5th and 95th percentile for input.
        input_values = copy(input, start, stop)
        qsort(input_values)
        input_white = input_values[5 * (stop - start) / 100]
        input_black = input_values[95 * (stop - start) / 100]
        # Compute 5th and 95th percentile for simulator.
        sim_values = copy(simulator, start, stop)
        qsort(sim_values)
        sim_white = sim_values[5 * (stop - start) / 100]
        sim_black = sim_values[95 * (stop - start) / 100]
        # Store adjustments.
        adjust_white[boundary] = input_white - sim_white
        adjust_black[boundary] = input_black - sim_black
    # Linear interpolation between boundaries.
    for boundary from 1 to 7:
        start = boundary_index(boundary - 1)
        stop = boundary_index(boundary)
        width = stop - start
        for x from start to stop:
            black[x] += (
                (x - start) * adjust_black[boundary] / width +
                (stop - x) * adjust_black[boundary - 1] / width)
            white[x] += (
                (x - start) * adjust_white[boundary] / width +
                (stop - x) * adjust_white[boundary - 1] / width)
```

Figure 5.5 shows an example for the iterative adjustment of the illumination estimates. The thick black curves at the top and bottom of each graph represent the functions f_{black} and f_{white} respectively, and the thin gray line shows f_{guess} alternating between them. Like in the other graphs, the input f_{cam} is red and the result of the simulation f_{sim} is blue.

5.4 Confidence

Because of error correction, the algorithm presented here always finds a solution with a matching checksum digit. However, this solution is not always correct. If the barcode was not located properly, the simulation may approximate only some part of the barcode area, or the locations of edges between digits may be wrong. A separate method is required to detect if the simulation was successful.

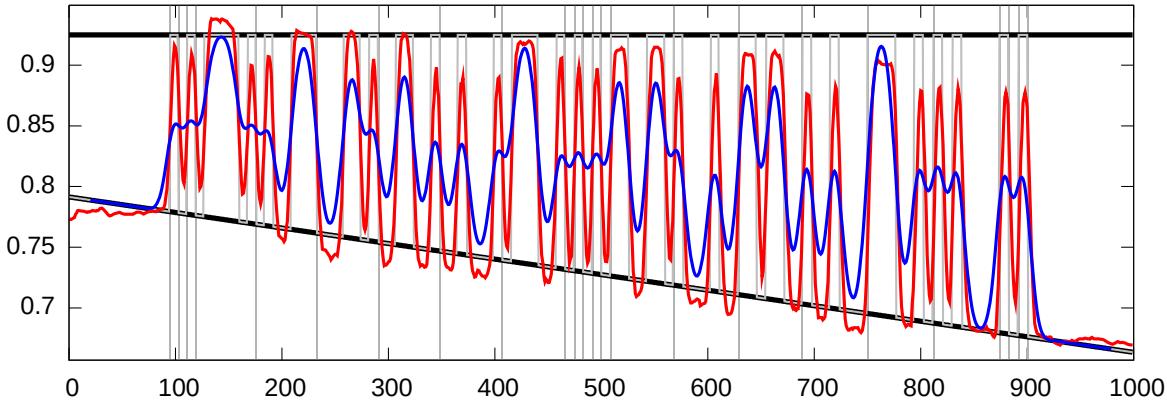
I propose a simple measure of confidence that is based on the sum of squares residual, as described in section 3.6. If all digits are set to the best candidates (with the smallest residual), the confidence measure is defined to be 1.0, or 100%. For each digit that had to be changed during error correction, the confidence measure is reduced. The reduction is greater if the changed digit has large residual compared to the best candidate.

$$c = 1.0 - \sum_{i=1}^{12} 0.1 \frac{e_i}{b_i}$$

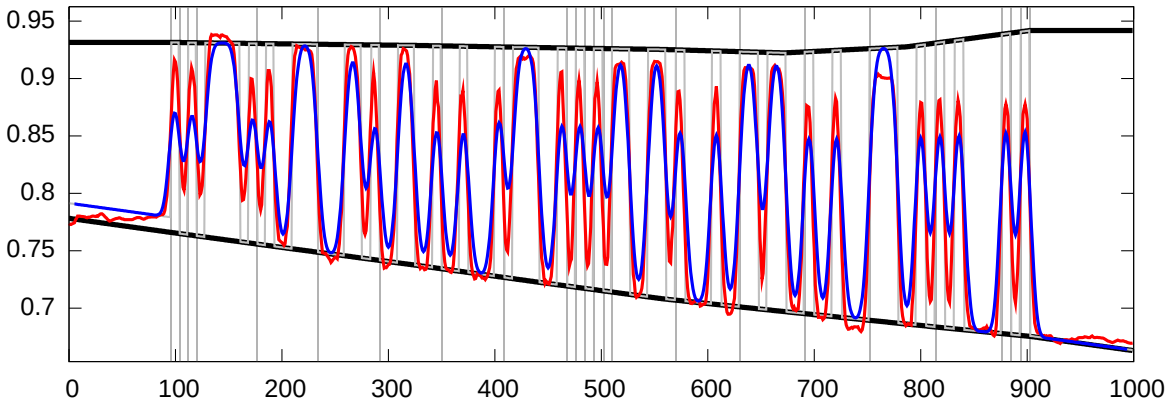
In this equation, b_i denotes the residual of the best candidate for the ith digit, and e_i denotes the residual for the same digit after error correction.

5.5 Multiple images

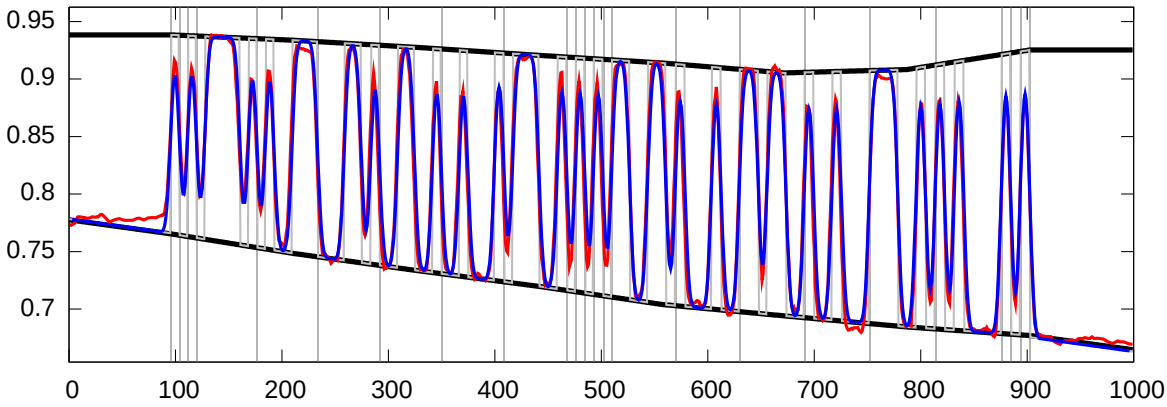
If the decoder is embedded in an interactive front-end with input from a live camera, multiple images can be analyzed to improve the confidence of the recognition. If consecutive results indicate the same digits, they are more likely to be correct. This information could be integrated with the confidence measure above and remains an interesting topic for future research.



(a) First pass: initial estimates



(b) Second pass: adjusted 50%



(c) Third pass: adjusted 100%

Figure 5.5: Iterative adjustment of non-uniform illumination and blur kernel size

6 Implementation details

The prototype decoder software for this research project was written in C (using standard ANSI C99) because of speed and portability. The decoder is directly linked with three different front-end applications described below. It is developed in a separate folder called `decoder` which contains the following modules (and corresponding `*.h` header files) with a total of 2772 physical source lines of code, according to David A. Wheeler's SLOCCount.

`blur.c`	`dir8.c`	`peaks.c`	`sections.c`
`changes.c`	`encoder.c`	`perspective.c`	`simulator.c`
`checksums.c`	`gnuplot.c`	`points.c`	`thirty.c`
`corners.c`	`gradients.c`	`quietzones.c`	`threshold.c`
`decoder.c`	`illumination.c`	`roi.c`	`trace.c`
`dir16.c`	`minmax.c`	`scanlines.c`	

6.1 OpenCV

The prototype implementation uses OpenCV, a computer vision library that was originally developed by Intel. OpenCV is available under a BSD license and free for commercial and research use. The book by Bradski and Kaehler (2008) provides a good introduction to the library.

Each pixel image is stored in an `IplImage` structure, which includes a pointer to the raw pixel data and a description of the image properties like dimensions, samples per pixel and bits per sample. The OpenCV library contains many useful functions to manipulate these images.

OpenCV also contains a simple cross-platform GUI library with support for displaying and updating results in a window on the screen, and direct input from webcams, e.g. the iSight camera on top of the MacBook display.

6.2 Interactive processing

Most of the prototype development was done on Mac OS X. The interactive prototype front-end uses the built-in iSight webcam and processes live video at up to 10 frames per second. This front-end allows direct experimentation with fast round-trip between implementation and testing. Figure 6.1 shows a screenshot of the interactive front-end. It can be configured in the source code to display different processing stages. This example shows the RGB camera input

image in the top left image, the ridge and valley traces on the red channel in the middle, and the detected barcode area as a red frame on the right, with green frames for potential quiet zones. The bottom half of the display is a plot of the brightness values in the input (in red), the adaptive threshold (in orange), and the result of the blurry barcode simulation (in blue).

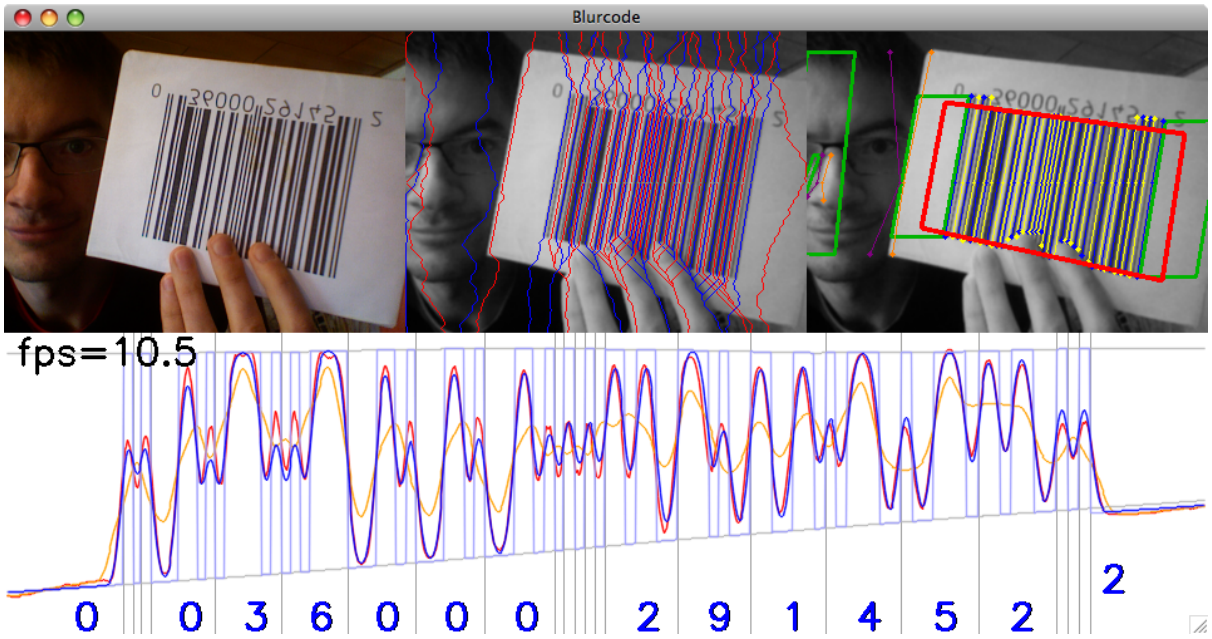


Figure 6.1: Screenshot of the interactive prototype on a MacBook

6.3 Still pictures

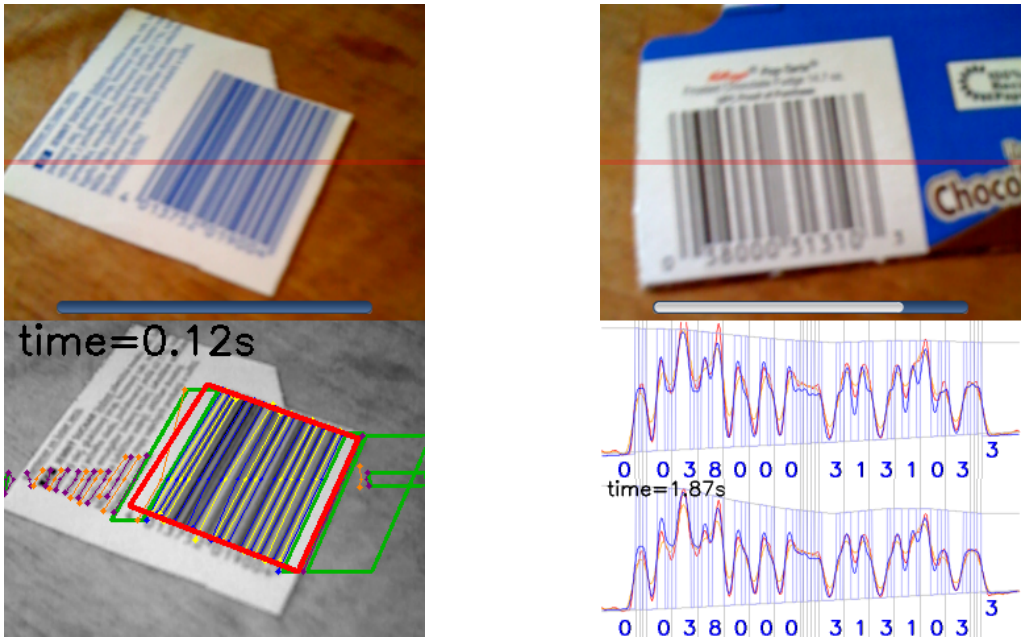
A similar program was developed to process saved pictures in batch mode. It is run from the command line, and the names of the input files are passed as command line arguments. The processing is exactly the same as for the interactive prototype, but the results are written to PNG files in the output folder for inspection. The program skips existing output files, so it will only process new files that have been added to the test suite. But if it is called with the command line option `--force`, it will re-process even existing output files, which is useful for testing after the decoder has been improved.

6.4 iPhone prototype

The same decoder software was embedded into an iPhone application prototype. Software for the iPhone is written in Objective C, but standard C code can be included, so the `decoder` folder containing the C modules listed above is simply linked into the Xcode project.

The iPhone application starts the camera preview which is normally intended to let the user take pictures at the full resolution of 1.92 megapixels. Unfortunately, the capturing and encoding of the image takes several seconds before the application can start to process it. A better solution is to hide the buttons in the camera preview window and use a timer to capture the live preview image from the screen.

The screen resolution of the iPhone is 320×480 pixels (aspect ratio 2:3), but the camera preview is only 320×426 pixels (aspect ratio 3:4). However, a typical barcode at full screen is very close to the camera and thus extremely blurry. The decoder works faster and still reliable at half resolution, so only the top half of the captured screen (320×240 pixels) is used for decoder input. The bottom half can be used to display the intermediary results of image processing in the decoder, or two graphs of the blurry barcode simulator, including the resulting digits.



(a) Results of bar tracing and quiet zone detector

(b) Graphs of input and simulation for iterations 2 and 3

Figure 6.2: Screenshots of the prototype running on the Apple iPhone

The decoder runs in a background thread, so that the live camera preview continues to be updated, and a progress bar indicates the processing stages.

7 Test results

To evaluate the reliability of the proposed method, it was tested with barcode images from several different cameras. This chapter presents the percentages of correct recognition for each set of test images, as well as results for partially correct recognition and reduced resolution. The prototype is also compared to four existing decoders with a smaller selection of test images.

7.1 Test suite

I collected a test suite of 466 images. It includes barcodes of grocery and retail products, books, CDs, and DVDs. The entire test suite in full resolution occupies 231 megabytes. To reduce storage requirements and improve processing speed, all images were scaled down to a maximum size of 640×480 pixels and saved as JPEG images with 90% encoding quality. Some test images already had smaller dimensions, so they were not re-sampled. The median of the reduced file sizes is 50 kilobytes, and the entire reduced test suite fits in 25.1 megabytes.

Image source	Total	Correct	Success rate
Apple iPhone	112	66	58.9%
Apple iPhone with macro lens	113	93	82.3%
Sony Ericsson T650i	121	96	79.3%
PDAbar correct	76	64	84.2%
PDAbar not found	44	21	47.7%
All test images	466	340	73.0%

Table 7.1: Test results for five sets of test images

Table 7.1 shows test results for the prototype decoder on five sets of test images. The first two sets were captured with an iPhone using the built-in camera. For the second set, the sharpness of the images was improved by a Griffin Clarifi protective case with macro lens.

The third set of images was taken with a Sony Ericsson T650i camera phone with auto-focus lens. The focus mode on the phone was set to “macro” for most of the images in this set, but the automatic focus was not always accurate, so there are also several blurry images.

The last two sets of test images were provided by the developers of the PDAbar library (Kritzner, 2008). Their decoder correctly recognized the barcodes in the “correct” set but rejected the images in the other set because they are out of focus or contain reflections.

7.1.1 Automatic test bench

The filename of each test image contains the correct EAN-13 number. This makes it possible to compare the output of the decoder with the true digits, and automatically generate statistics for success and failure rates. For readability, the digits in the filename are grouped with hyphens in the positions of the EAN-13 guard bars, e.g. `0-123456-789012-filename.jpg`.

The command line front-end runs in batch mode and produces PNG images in the `output` folder to show the processing steps and results. It also stores the recognized digits as symbolic links in the `digits` folder, in the form `0-123456-789012-filename.jpg` → `0123456789012`. These symlinks are used to produce statistics with a Python script. It directly generated the LaTeX tables included in this chapter.

7.1.2 Partially correct recognition

Because the prototype decoder guesses each digit separately, it is possible that only some digits are recognized correctly. Common reasons for partially incorrect results include severe camera blur, non-uniform illumination and non-planar barcode surface. Table 7.2 shows the percentages of partially correct results for each of the five sets of test images. The prototype never produces 12 correct digits and only one incorrect digit because of error correction.

Number of correct digits	0…3	4…7	8…11	13
Apple iPhone	11.6%	6.2%	23.2%	58.9%
Apple iPhone with macro lens	11.5%	4.4%	1.8%	82.3%
Sony Ericsson T650i	14.0%	1.7%	5.0%	79.3%
PDAbar correct	13.2%	0.0%	2.6%	84.2%
PDAbar not found	29.5%	13.6%	9.1%	47.7%
All test images	14.2%	4.3%	8.6%	73.0%

Table 7.2: Percentages of partially correct results

The first set of test images (iPhone without macro lens) produced a significant percentage of results with 8 to 11 correct digits. The barcode was correctly located in these images, but the camera blur was too severe for correct recognition. This indicates a good focus point for future development. Analyzing the results of multiple consecutive images may lead to correct results for these blurry images.

7.1.3 Low resolution

Almost all images in the test suite have a resolution of 640×480 pixels. Separate test runs were performed with the same test images, but with reduced resolutions. The scaling was performed in the batch processing front-end for the prototype decoder, using the OpenCV

function `cvResize` with the `CV_INTER_AREA` option. The other parameters of the decoder were not changed for the low resolution tests.

Maximum width in pixels	640	320	213	160	128
Apple iPhone	58.9%	44.6%	32.1%	21.4%	14.3%
Apple iPhone with macro lens	82.3%	77.9%	55.8%	39.8%	12.4%
Sony Ericsson T650i	79.3%	73.6%	44.6%	16.9%	1.6%
PDAbar correct	84.2%	85.1%	27.0%	15.1%	2.6%
PDAbar not found	47.7%	48.8%	31.0%	7.1%	2.3%
All test images	73.0%	67.2%	40.3%	22.5%	7.5%

Table 7.3: Test results for different resolutions

The success rate generally decreases with lower resolution, but the method is still usable at 320 pixels. The results for both sets of PDAbar test images are actually improved, because the resizing reduces problems with very grainy images.

7.2 Comparison with other solutions

A smaller set of 60 images was selected from the large test suite. It contains images of 20 UPC-A barcodes, 20 EAN-13 barcodes, and 20 ISBN-13 barcodes, some of which are accompanied by an EAN-5 barcode for the suggested retail price. Each of these 60 images was processed with the thesis prototype and four different existing decoders. I decided to use only 60 images because I did not have access to a batch processing interface for all of the other decoders, and manual testing was required. I tried to cover a wide variety of problematic test images, including non-uniform illumination, perspective distortion and camera blur. The test images were not selected because my prototype decoder could recognize them successfully, but I admit that the prototype was tuned after the test suite was finalized. The results and some example images are shown in appendix A.

7.2.1 Input for each decoder

For the thesis prototype and the open-source ZXing decoder, the batch mode front-end was used to process all images in the comparison test suite.

Arndt Kritzner and Steven Thielemann of Logic Way GmbH provided the results for their PDAbar decoder after I submitted my test images to them.

The iPhone application ItemShelf was tested by uploading the test images to the iPhone and opening each image in the application with the “Photo library scan” function. Some images were manually zoomed and panned using the built-in interface if the barcode was too small or not centered. Several images were successfully decoded but produced an error message

because the corresponding product could not be found on Amazon's web service. I assume that all digits were recognized correctly in these cases, but I could not verify it.

The Delicious Library software allows interactive scanning with the MacBook's built-in iSight camera. Each test image was presented on the screen of a second laptop, and then the display was moved in all directions and even rotated and tilted to allow the camera to capture it. This was performed in a dark room with few reflections. Camera blur was not an issue because the barcodes were displayed at full screen size, and the distance between camera and display was about 40 centimeters. It is possible that the success rate is overestimated because the display movements compensated some difficulties like barcode rotation.

7.2.2 Discussion

The percentages of successful recognition in the decoder comparison are weak indicators for the general success rate of each decoder, because of the limited number and arbitrary choice of test images. Most of the images in the comparison test suite suffer from severe camera blur because they were taken without a macro lens. The ZXing, PDAbar and ItemShelf decoders are not designed to work with such blurry images.

The set of images correctly recognized by the prototype decoder is not a superset of the respective sets for any other decoder in the comparison. In other words, each of the other decoders correctly recognized images that the prototype decoded incorrectly. For example, the prototype does not attempt to decode barcodes that are upside down (rotated 180°).

However, it is clear that the proposed method can correctly recognize barcodes with higher levels of blur and perspective distortion than existing solutions, even considering the limitations above.

8 Summary

This work presents a solution to the problem of decoding linear barcodes from blurry camera images under processing constraints.

A mathematical model is proposed for simulating the brightness values on a scan line across the blurry barcode. Camera blur is simulated by convolution with a Gaussian kernel. The model for perspective distortion is based on the angle between the barcode and the projection plane, and the distance between the camera and the barcode. Non-uniform illumination is simulated with two continuous functions for the black and white brightness level across the barcode area.

The bright and dark stripes of the barcode are traced to find the top and bottom edges of the barcode, using an adaptive threshold to detect good starting points. A robust method calculates the four corners of the barcode from the endpoints of the traces. The four corners are used to extract brightness values from the camera image with sub-pixel sampling from several parallel lines, and to estimate the camera distance and angle. An iterative algorithm adjusts the remaining parameters of the simulation, including the locations of the left and right edges of the barcode, the width of the blur kernel, and the brightness functions for non-uniform illumination.

The decoder selects the digits for which the simulation best matches the camera input with the ordinary least squares (OLS) method. Error detection and correction is performed using the explicit checksum digit and the limited number of possible code combinations for the first six characters.

A software prototype was implemented in C, with three different front-ends. One version runs on a MacBook and processes live images from the built-in camera, at 10 frames per second. A stand-alone decoder processes images in batch mode. The third version runs directly on the iPhone and continuously processes images from a live camera preview, taking less than two seconds per image.

The prototype was tested with several hundred test images from different cameras, including the iPhone camera, with and without a macro lens. The decoder recognizes most EAN-13 and UPC-A barcodes at normal size using the iPhone's built-in camera without a macro lens. However, a good macro lens (available as part of a protective case) improves the results for smaller barcodes.

Compared to three existing decoders, the proposed method significantly improves the rate of successful barcode recognition for blurry images. The prototype results show only moderate improvement over the interactive scanner used in the Delicious Library application. In this comparison, the main advantage of the proposed method is that it does not require a vector processor and can be implemented directly on a mobile device like the Apple iPhone.

8.1 Future research

Future development of the decoder may include support for the smaller but structurally similar barcode symbologies UPC-E and EAN-8. It may be possible to extend the digit guessing approach to other linear symbologies, but it may be less accurate if there are more than 20 choices for each character.

The detection of incorrect decoding results still needs to be improved. The confidence measure could use information from multiple consecutive images when a live camera feed is available. This is the case for the interactive prototype front-ends on both the MacBook and the iPhone.

For commercial use, it would be good to improve the decoder performance on the iPhone to one image per second or more. The prototype is not optimized for speed, so it is possible that the next version will be significantly faster. For example, the processing of brightness curves could be changed from floating point to integer arithmetic.

Some general types of barcode capture problems cannot be solved with the current implementation. The geometric model for perspective projection is designed for planar barcodes because it is difficult to determine complex geometry from a single blurry picture. The prototype fails to locate the digit boundaries correctly if the barcode is printed on a curved surface and the middle guard bars are significantly closer to the camera than the outer guards, e.g. on a bottle. The question is if a more detailed geometric model would introduce too many parameters that cannot be adjusted automatically.

A Decoder comparison

UPC barcodes are primarily used in North America. Most of these were taken in Seattle, USA.

Image	ZXing Decoder 1.3	ItemShelf Lite 1.5	Logic Way PDAbar 1.08.03.25	Delicious Library 2.0.7	Thesis Prototype 1.0
(a) 0010126000747				✓	✓
(b) 0010126000747				✓	✓
(c) 0015700050545					✓
(d) 0022924048111				✓	✓
(e) 0034449002127		✓		✓	✓
(f) 0041449001050					✓
(g) 0042000062008	✓	✓	✓	✓	✓
(h) 0070617006115				✓	✓
(i) 0070617006115	✓	✓			✓
(j) 0070617006115	✓	✓			✓
(k) 0071101001838				✓	✓
(l) 0072251001556				✓	✓
(m) 0082839390415					
(n) 0090497299995				✓	✓
(o) 0123456789012		✓	✓	✓	✓
(p) 0187471000002				✓	✓
(q) 0645177436780				✓	✓
(r) 0740500962308					✓
(s) 0854622001453					✓
(t) 0891356000260				✓	✓
Success rate	15%	25%	10%	65%	95%

Table A.1: Decoder comparison for UPC-A barcodes

(b) 0010126000747

(g) 0042000062008

(p) 0187471000002

Figure A.1: Test images for UPC-A barcodes

Most of the EAN-13 barcode test images were taken of grocery and retail products in Germany. The entire comparison test suite includes 18 test images from Logic Way GmbH. 15 of those images were recognized correctly by their PDAbar decoder.

Image	ZXing Decoder 1.3	ItemShelf Lite 1.5	Logic Way PDAbar 1.08.03.25	Delicious Library 2.0.7	Thesis Prototype 1.0
(a) 2215812003752	✓	✓	✓	✓	✓
(b) 4000521642402				✓	✓
(c) 4001475105609	✓	✓	✓	✓	✓
(d) 4002448038061		✓	✓	✓	✓
(e) 4002590108001	✓	✓	✓	✓	
(f) 4003274006347				✓	✓
(g) 4005808890576	✓	✓	✓	✓	✓
(h) 4005906002352				✓	✓
(i) 4006067006142					✓
(j) 4011200563901	✓	✓	✓	✓	✓
(k) 4011439302418	✓	✓	✓	✓	✓
(l) 4015400112600					✓
(m) 4030400200461	✓	✓	✓	✓	✓
(n) 4052400033559	✓		✓	✓	✓
(o) 4300175167314	✓	✓	✓	✓	✓
(p) 4300175172806		✓	✓		✓
(q) 4306188124276					✓
(r) 6005809655035	✓		✓	✓	✓
(s) 8013663000753	✓	✓			
(t) 9008700120258				✓	✓
Success rate	55%	55%	60%	75%	90%

Table A.2: Decoder comparison for EAN-13 barcodes

(g) 4005808890576

(l) 4015400112600

(o) 4300175167314

Figure A.2: Test images for EAN-13 barcodes

Locating blurry ISBN-13 barcodes properly is slightly more difficult because they are often accompanied by a smaller EAN-5 barcode on the right, and the quiet zone between them is narrow. The interactive camera scan of Delicious Library has an advantage here because the user positions the barcode according to green boxes on the live camera display.

Image	ZXing Decoder 1.3	ItemShelf Lite 1.5	Logic Way PDAbar 1.08.03.25	Delicious Library 2.0.7	Thesis Prototype 1.0
(a) 9780061044434					✓
(b) 9780312364267				✓	✓
(c) 9780312369088				✓	✓
(d) 9780399154362					
(e) 9780425212929				✓	✓
(f) 9780525950363					✓
(g) 9780553560244				✓	
(h) 9780553574661				✓	✓
(i) 9780596006754	✓	✓	✓	✓	✓
(j) 9780596006754	✓	✓	✓	✓	
(k) 9781405403849	✓	✓	✓	✓	✓
(l) 9781433249129				✓	✓
(m) 9781582294551				✓	✓
(n) 9781590585214				✓	✓
(o) 9783772350825				✓	
(p) 9783794142170				✓	✓
(q) 9783811859838					✓
(r) 9783893609482	✓	✓	✓	✓	✓
(s) 9783899901153		✓	✓	✓	✓
(t) 9788420532318			✓	✓	✓
Success rate	20%	25%	30%	80%	80%

Table A.3: Decoder comparison for ISBN-13 barcodes

(c) 9780312369088

(j) 9780596006754

(q) 9783811859838

Figure A.3: Test images for ISBN-13 barcodes

Bibliography

R. Barnes, J. Hudgins, and A. Muse. Shopsavvy. Android Market, 2009. URL http://www.biggu.com/. Last checked Apr. 14, 2009.

G. Bradski and A. Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, Inc., Cambridge, MA, 2008. ISBN 978-0-5965-1613-0.

J. Brown. Zbar barcode reader (formerly Zebra). SourceForge, 2009. URL http://sourceforge.net/projects/zbar/. Last checked Apr. 21, 2009.

T. Burton. Barcode writer in pure postscript. Project Homepage, 2009. URL http://www.terryburton.co.uk/barcodewriter/. Last checked Apr. 11, 2009.

J. F. Canny. A computational approach to edge detection. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986. ISSN 0162-8828.

R. O. Duda and P. E. Hart. Use of the Hough transformation to detect lines and curves in pictures. *Commun. ACM*, 15(1):11–15, 1972. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/361237.361242.

S. Esedoglu. Blind deconvolution of bar code signals. *Inverse Problems*, 20:121–135, Feb. 2004. doi: 10.1088/0266-5611/20/1/007.

GS1. General Specifications. Version 9.0, Issue 1, Jan. 2009. Available from local GS1 member organisations.

C. Hockenberry. What the iPhone specs don't tell you... furbo.org, Aug. 2007. URL http://furbo.org/2007/08/21/what-the-iphone-specs-dont-tell-you/. Last checked Apr. 11, 2009.

P. Hough. Method and means for recognizing complex patterns. U.S. Patent 3,069,654, Dec. 1962.

E. Joseph and T. Pavlidis. Peak classifier for bar code waveforms. In *11th IAPR International Conf. Pattern Recognition Methodology and Systems*, volume 2, pages 238–241, Sept. 1992. ISBN 0-8186-2915-0. doi: 10.1109/ICPR.1992.201763.

E. Joseph and T. Pavlidis. Deblurring of bilevel waveforms. *IEEE Trans. Image Processing*, 2 (2):223–235, Apr. 1993. ISSN 1057-7149. doi: 10.1109/83.217225.

J. Kim and H. Lee. Joint nonuniform illumination estimation and deblurring for bar code signals. *Optics Express*, 15:14817–+, 2007. doi: 10.1364/OE.15.014817.

A. Kritzner. PDAbar-Bibliothek. Logic Way GmbH, Schwerin, 2008. URL http://www.logicway.de/pages/pda-barcode.shtml. Last checked Apr. 14, 2009.

X.-J. Lu, G.-L. Fan, and Y.-K. Wang. A robust barcode reading method based on image analysis of a hierarchical feature classification. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference*, pages 3358–3362, Beijing, China, Oct. 2006. ISBN 1-4244-0258-1. doi: 10.1109/IROS.2006.282512.

T. Murakami et al. ItemShelf Lite. iPhone App Store, 2009. URL http://itemshelf.com/. Last checked Apr. 21, 2009.

E. Ohbuchi, H. Hanaizumi, and L. A. Hock. Barcode readers using the camera device in mobile phones. In *CW '04: Proceedings of the 2004 International Conference on Cyberworlds*, pages 260–265, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2140-1. doi: http://dx.doi.org/10.1109/CW.2004.23.

S. Owen, D. Switkin, et al. ZXing. code.google.com, 2009. URL http://code.google.com/p/zxing/. Last checked Apr. 14, 2009.

T. Pavlidis, J. Swartz, and Y. P. Wang. Fundamentals of bar code information theory. *Computer*, 23(4):74–86, 1990. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.55471.

R. Shams and P. Sadeghi. Bar code recognition in highly distorted and low resolution images. In *IEEE International Conf. Acoustics, Speech and Signal Processing*, volume 1, pages I–737–I–740, Honolulu, HI, USA, 2007. IEEE Computer Society. ISBN 1-4244-0727-3.

S. Shellhammer, D. Goren, and T. Pavlidis. Novel signal-processing techniques in barcode scanning. *IEEE Robotics & Automation Magazine*, 6(1):57–65, Mar. 1999. ISSN 1070-9932. doi: 10.1109/100.755815.

W. Shipley. Delicious Library. Delicious Monster, Seattle, 2009. URL http://www.delicious-monster.com/. Last checked Apr. 14, 2009.

K.-Q. Wang, Y.-M. Zou, and H. Wang. Bar code reading from images captured by camera phones. *IEE Conference Publications*, 2005(CP496):42, Nov. 2005. doi: 10.1049/cp:20051487.

K.-Q. Wang, Y.-M. Zou, and H. Wang. 1D bar code reading on camera phones. *Int. J. Image Graphics*, 7(3):529–550, 2007.

T. Wittman. Lost in the supermarket: Decoding blurry barcodes. *SIAM News*, 37(7), Sept. 2004.

X.-W. Xu, Z.-Y. Wang, Y.-Q. Zhang, and Y.-H. Liang. A skew distortion correction method for 2d bar code images based on vanishing points. *Machine Learning and Cybernetics, 2007 International Conference on*, 3:1652–1656, Aug. 2007. doi: 10.1109/ICMLC.2007.4370412.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Johann C. Rocholl)