Recursive Method to Detect and Segment Multiple Rectangular Objects in Scanned Images

MSR-TR-2004-01 January 5, 2004

Cormac Herley

***Abstract*—Rectangular objects frequently appear in certain classes of images. Scanned images of photographs, receipts or other small objects are obvious examples. It is often desirable to detect the existence, number and location of such rectangular objects to single them out for separate treatment. While image segmentation is generally a hard problem, we find that by considering one dimensional projections of the image it is possible to recursively simplify the segmentation problem efficiently and robustly. We outline the algorithm and its components and discuss its performance on real data from user scanned images.**

***Index Terms*— image segmentation, scanned photographs**

I. Introduction

The problem we address is to examine an image and determine whether it contains rectangular objects. If so we wish to determine their vertices. The rectangles can be of any sizes, at any positions and orientations and there can be any number of them in the image (subject merely to some constraint on the minimum size of an object). We assume that they do not overlap with each other, and are distinguishable from the background in that at least a majority of pixels in the interior of each rectangle differ from the background color by more than a threshold amount (in some suitable metric). An obvious application is to scanner images, so that multiple photographs, receipts, or business cards might be segmented and stored automatically. We are primarily interested in objects on a constant color background. For example, most scanners have a constant black, white or grey color, but we also accommodate any other background that is predominately of one color, even where there are marks, or noise or other distortions on the background. An example is shown in Figure 1.

Attention to scanned images in the literature is less common than one might imagine. There are various examinations of orientation detection and deskewing of scanned documents [4-6], cancellation of show-through on duplex documents [7], and segmentation when only a single object is present [3]. The author has been unable to find a single approach to the multiple rectangle segmentation problem in the recent image processing literature. Thus, while the problem we address is of some practical importance, it appears to have received little attention in the published literature. We wish to be very clear at the outset that we are not solving or addressing the compound document analysis problem: we do not seek to separate images from text on scans of magazine pages for example. We also do not seek to identify buildings, doors or other rectangular objects within images; these are interesting and hard problems, but beyond the scope of this work.

Given that we seek rectangles the obvious approaches might be to seek lines and/or corners in the image and match them up four at a time to form objects. While this is feasible when only a single object is present it can become very complicated as the number of objects increases. Some of the reasons that the multiple object case is harder are:

- Detecting either lines or corners at arbitrary positions and orientations can be computationally expensive. The Hough transform [2], a standard method for determining the existence of lines in an image, requires many operations and is often error prone.

- Certain images will have one or more edges that are hard to distinguish from the background (e.g. photographs of snow scenes scanned on a white background or underexposed images on a dark background). Any algorithm that is based on line detection is likely to fail in these cases.
- There is often sharing of lines among objects: two or more objects at the same orientation can often be confused as one. In Figure 1, for example, the leftmost edge of the two photos on the left are collinear, and might easily be confused as a single line rather than two. Problems of shared collinear edges are more common than might be imagined, as users tend to align objects when placed on the platen.

Segmentation of images is in general a hard problem, the two-dimensional nature of the data makes looking for even well defined objects like rectangles difficult or computationally expensive. Our solution hinges on the fact that by taking appropriate one dimensional projections of the data the problem often becomes simple.

Notation: we call an image Im(i, j). For convenience we denote by Im(i_0:i_1, j_0:j_1) the sub-image that consists of rows i_0 through i_1 and columns j_0 through j_1 inclusive. We will assume that images have three color planes, and the red, green and blue values at location (i, j) are Im(i, j).R, Im(i, j).G and Im(i, j).B respectively. We will denote the background color of the image as Bg. We will use the symbol $|\mathrm{Im}(i, j)|_{rgb}$ to denote a norm. For example $|\mathrm{Im}(i, j) - \mathrm{Bg}|_{rgb}$ is a scalar quantity representing the difference between an image pixel and the background. A simple example metric would be one that transforms the color data to grayscale, and then takes the absolute difference. We will explain reasons for considering more complicated norms in Section V D.

In the next section we show that when an image contains only a single rectangular object the vertices are easily determined. In Section III we give an example of how a recursive divide and conquer technique can simplify the multiple object case to many single object cases, and give the core of the algorithm. In Sections IV and V we examine implementation details, and how our algorithm fares when faced with real data that deviates from our idealized assumptions.

II. SINGLE OBJECT CASE

If an image, or sub-image, contains only a single rectangular object, then many of the difficulties listed in the previous section are not a factor. Key to our method for solving the multiple object case will be an efficient and robust scheme for the single object case. Consider the example shown in Figure 2. We assume that the background color Bg has been estimated, and that at least a majority of the pixels in the interior of the rectangle differ from Bg by more than a threshold amount. We call any pixel for which $|Im(i,j) - Bg|_{rgb} > T$ a data pixel, and all others background pixels. We defer until Sections IV and V all discussion of how Bg can be estimated, how a threshold is chosen, and an appropriate metric. The simplest possible case is shown in Figure 2, where a constant color rectangular object is shown on a constant color background. Since there are only two colors, there is no difficulty distinguishing between data and background pixels, so determining Bg and a suitable threshold is not difficult.

Next, suppose we calculate the number of data pixels in the j-th row P(j), and the i-th column Q(i). That is: P(j) is the number of pixels for which $|Im(i,j) - Bg|_{rgb} > T$. Clearly $0 \leq P(j) \leq i_{max}$, since P(j) is positive, and cannot exceed the number of columns, and similarly $0 \leq Q(i) \leq j_{max}$, since Q(i) cannot exceed the number of rows. These functions have been plotted in Figure 2. Observe that for the first and last few rows we have P(j) = 0, since there are no data pixels at the top and bottom of the image. P(j) becomes non-zero at rows a and d (the rows that contain the corners of the rectangular object), and ramps from there to it's maximum width between rows b and c. Similarly Q(i) has a trapezoidal shape as also shown in Figure 2. Elementary geometry gives that the max of P(j) is equal to x Cos(t) and of Q(i) is y cos(t), where x and y are the dimensions of the rectangle, and t the angle at which it is oriented. The corners of the rectangle are the four points (g,a), (h, c), (f,d) and (e,b) which correspond to the inflexion points of the trapezoids P(j) and Q(i).

There is actually a second possible rectangular object that would produce the same P(j) and Q(i) functions, as shown in Figure 3. This rectangle with coordinates (h,b), (g,d), (e,c) and (f,a) is at the same position as the first, but oriented at angle –t, and is the only other possible rectangle that would generate the observed trapezoids. We can determine which of the two orientations is correct by checking whether the region with vertices (g,a), (h, c), (f,d) and (e,b) contains more data pixels than the one with vertices (h,b), (g,d), (e,c) and (f,a). If so the orientation is t, and otherwise it is –t. Call these numbers S(t) and S(-t) respectively.

Thus, when the image, or sub-image, contains only a single rectangular object we have a means of solving the problem:

- Calculate P(j) and Q(i)
- Determine the knee-points of these trapezoids
- Determine the vertices of the two possible rectangles (at angles t and –t)
- Determine whether a rectangle at t or –t is present; i.e. is S(t) > S(-t) ?

Issues such as difficulties in determining the knee-points and the effect of departures from the ideal case shown in Figure 2 will obviously have to be considered, but we defer them to Section V.

Note that the absence of a rectangular object in the image is easily determined: the total number of data pixels in the image is Σ_j P(j) = Σ_i Q(i). Call this number S(Im). Recall that S(t) is the number of data pixels in the estimated rectangle and orientation t. If the larger of S(t) and S(-t) is not approximately equal to S(Im) then the object present is not a rectangle or the assumption that $|Im(I,j) - Bg|_{rgb}$ < threshold for a majority of the interior pixels is invalid. Thus failure is easily detected.

To summarize the foregoing: a pseudo-code representation of examining an image for a single rectangular object goes as follows:

```
function singleObject(Im){
        [P, Q] = getProjections(Im);
        [a,b,c,d] = getKnees(P);
        [e,f,g,h] = getKnees(Q);
        for i = 1, length(P){ sigTotal += P(i);}
        sigPlus = pixelsIn([a,b,c,d], [e,f,g,h],1);
        sigMinus = pixelsIn([a,b,c,d],[e,f,g,h],-1);
        if (sigPlus > max(sigMinus, sigTotal * 0.9))
                return plus;
        if (sigMinus > max(sigplus, sigTotal * 0.9))
                return minus;
        return 0;
}
```

where pixelsIn([a,b,c,d],[e,f,g,h],tilt) returns the number of data pixels interior to the rectangle positively or negatively oriented depending on whether tilt is 1 or -1. This function returns “plus” if the rectangle with vertices (h,b), (g,d), (e,c) and (f,a) is a fit, “minus” if the rectangle with vertices vertices (g,a), (h, c), (f,d) and (e,b) is a fit, and “zero” otherwise.

We should note at this point that many approaches are likely to be successful in the case where only a single object is present [3]. Explicitly searching for corners or lines may be quite feasible. The method presented here is quite robust, but also has the advantage that it leads naturally to our extension for multiple objects.

III. Multiple objects

As mentioned in Section I, when the image consists of multiple rectangles the situation becomes a great deal more complicated than when only one rectangle is present. Nonetheless we'll follow the approach started in Section II and calculate the quantities P(j) and Q(i). These will now consist of the sums of the trapezoids generated by each of the individual rectangles. Our general approach is best illustrated first by an example.

A. Example

See for example Figure 4, where three rectangles are present. In the ideal case it might be possible to estimate the parameters if one knew, or guessed, that three trapezoids were present. But such an approach is unlikely to be robust when faced with real data, and will become very complicated as the number of rectangles (and hence trapezoids) increases.

Observe from Figure 4 however, that P(j) contains a gap at row j_0, that is at this location where $P(j_0)=0$. This indicates that there is no image data at this location. Since there is only background data on this row, there is no possibility that any part of a rectangular object crosses this row. There may be objects in the sub-image above this row $Im(0:i_{max}, 0:j_0)$ and the sub-image below $Im(0:i_{max}, j_0+1:j_{max})$, but no objects cross it. Thus the problem can be decoupled to examine the portions of the image above and below row j_0 separately. This is an important simplification as it allows us to split the problem in two.

This is shown in Figure 5. Let’s calculate the quantities P(j) and Q(i) over the two parts of the image the rows above j_0 and the rows below. We see from Figure 5 (a) that the part above consists of a single rectangle so P(j) and Q(i) end up being simple trapezoids. Thus this subproblem is solved by routine singleObject() as introduced in Section II. The part below j_0 consists of two rectangles and P(j) is the sum of two trapezoids. The handling of this sub-image is shown in Figure 5 (b). Observe that now there is a gap in Q(i) at location i_1 (i.e. $Q(i_1)=0$) indicating again that this sub-image be broken into even simpler sub-images by taking those columns to the left of i_1 (i.e. $Im(0:i_1,j_0:j_{max})$) and those to the right (i.e. $Im(i_1+1:i_{max},j_0:j_{max})$). Those sub-sub-images each contain a single rectangle and their P(j) and Q(i) functions are simple trapezoids, allowing them to be solved using routine singleObject().

The example shown in Figures 4 and 5 illustrates that a gap in the P(j) or Q(i) function of the image will allow us to break the problem into sub-problems, and even these sub-problems can often be similarly decomposed. In this example the simplification carried us all the way to sub-images that each contained only a single rectangle and could be solved using function singleObject().

It is not the case that the simplification always leads to sub-images that each contain a single rectangle. However, every simplification makes further simplifications more likely. For example, in the case above, the gap in the Q(i) function for $Im(0:i_{max}, j_0:j_{max})$ only became visible after performing the initial split, that is the Q(i) function for $Im(0:i_{max},0:j_{max})$ didn’t contain such a gap. We examine the completeness of the algorithm and performance on scanned data in Sections III C and D below.

B. *Recursive divide and conquer*

Our approach will be to calculate P(j) and Q(i) for a given image. If gaps are found, we simplify and re-apply to the sub-images until no further simplifications are possible. At the lowest level we have sub-images for which the P(j) and Q(i) functions contain no gaps. We then use routine singleObject() and if a single rectangle is found add it to the global list. Otherwise we decide that no rectangle has been found in the given sub-image. In either case we proceed with the other sub-images until no more sub-images remain. Because we apply much the same processing to an image and its sub-images the overall algorithm is efficiently implemented recursively. In pseudo-code here is the heart of our approach:

```
function procMult(Im){
[P, Q] = getProjections(Im);
[gP, gQ] = getGaps(P, Q);
if ( #(gaps in P) + #(gaps in Q) == 0)
    singleObject(Im);
else{
    for m = 1 to #(gaps in Q)
                for n = 1 to #(gaps in P)
            procMult(Im(gQ(m):gQ(m+1), gP(n):gP(n+1))
        }
}
```

Where the called functions are as follows:

getProjections(Im) is a routine to calculate P(j), Q(i) over sub-image Im, getGaps(P,Q) determines position of any gaps in P(j), Q(i), and singleObject(Im) examines P(j) and Q(i) for a single rectangular object and adds to the global list if found.

C. Experimental

To test the segmentation efficacy of the algorithm we applied it to a collection of images of scanned objects. A total of twelve users scanned photographs, business cards and business receipts on standard 8.5 by 11 flatbed scanners; ten different scanners were used in all. We processed a total of 139 images, 92 of which consisted of multiple photographs, and 47 of which were business cards, receipts or both. The users were asked to scan their objects, but were not given any details about the workings of the algorithm, its assumptions or restrictions. In only 3 of the 139 cases did the algorithm fail to segment simplify to sub-images containing only single objects. In all three of these cases objects were placed so close together as to allow no clearance. Clearly in the vast majority of cases the algorithm accurately and robustly segments real data from realistic scanning scenarios.

We demonstrated the algorithm simplifying a single example in Section III A, but this of course does not establish that the algorithm will in general simplify all multiple rectangle images. In fact, it is a simple matter to construct images made up of multiple rectangles for which the algorithm utterly fails. See for example Figure 8 where several examples of images that cannot be simplified are shown. Compare with the images shown in Figure 9, all of which are simplified to single object sub-images by the algorithm.

Thus, it is clear, that while the algorithm never makes a segmentation more complex, it does not simplify every image. How then to explain the near universal success on segmenting real data from actual scans as detailed above ? Since we are primarily interested in data from actual scans it is difficult to build an accurate statistical model of how people place objects on scanners. An examination of the images in Figure 8, where the algorithm fails to simplify, indicates that in these images the rectangular objects are often placed at arbitrary angles, and the area of the scanner platen is inefficiently used. An examination of the 139 scans from actual users reveals several tendencies:

- objects are almost always oriented so that edges are parallel to one of the platen edges
- objects are often arranged in grids to make most efficient use of space.

While it is difficult to model user preferences accurately we performed a Monte Carlo simulation to test how likely an composite rectangular image is to be simplified by algorithm procMult(). We automatically generated images consisting of rectangular objects on a constant color background. Each image was 1100 by 850 pixels, and the rectangular objects 400 by 600 (thus preserving the relative sizes of common US photo prints and flatbed scanner platen). Positions for up to three rectangular objects were chosen at random, subject to the constraints that they made an angle of no more than 2 degrees with the platen edge, and did not overlap. In certain cases only two objects could be placed without overlapping. The images so generated were used as input to procMult() to determine which could be simplified. In 100 trials 89 of the images were simplified completely by procMult(). While this simulation is somewhat simplistic it does help explain the far higher success rate that we observed on actual scanned data.

D. Complexity

The algorithm is inherently very efficient. In the recursion the number of times procMult() will be called is determined by the structure of the image, while singleObject() is called as many times as there are rectangular objects. For example, the images shown in Figures 1 and 4 required 8 and 5 calls to procMult() and 5 and 3 calls to singleObject() respectively. Calls to procMult() merely involve getting the projections P(j) and Q(i); we show in Section V C below that all of the required projections can be computed at the cost of two passes through the image data. Finding the gaps involve negligible computation, since they involve only one dimensional data. The bulk of the computation then is spent in the calls to singleObject(). Each such call requires:

- Calculating projections and kneepoints

- Determining the S(t) and S(-t)
- Improving the estimate of the corner locations.

The first point requires operations only on one dimensional quantities, and is a slight burden. Determining S(t) and S(-t) involves only one pass through the sub-image. The major burden is generally in improving the estimate of the corner locations, since (as pointed out in Sections V A and V B) a matched filter or least median square method is generally used here to determine the exact corner location once the knee-point method has determined their approximate location. This is also one of the few operations that should be performed on the full resolution data (see Section V F). The complete algorithm, including estimating corner locations, on a 600 dpi image containing 10 objects takes less than a second on a PIII 750 MHz machine, which of course is generally less than the time required to perform the scan.

IV. Estimation of The background color

In Sections II and III above we assumed that the background color, Bg, was known, that we chose a threshold, T, and that there was negligible confusion in classifying pixels as either background or data pixels by simply classifying any pixel for which $|Im(I,j)-Bg|_{rgb} < T$ as background. If the background color is not estimated accurately the pixel classification can be erroneous, gaps between trapezoids are not found, and algorithm procMult() fails to simplify the problem. Figure 6 shows an example of the effect of background estimation on the P(j) function. In Figure 6 (a) the P(j) function for the image in Figure 1 is shown, using a grayscale estimate of background color of 255. Pixels that are within a threshold T=10 of this value are classified as background. Figure 6 (b) shows the P(j) function of the same image, but in this case the background has been estimated as gray level 230. The gap that was so important to the simplifications exploited by algorithm procMult() has disappeared. While this is an extreme case of background misestimation, it illustrates the importance of accurate estimation of the background color. It might be imagined that Bg might be estimated once for a particular scanner, and would not have to be re-estimated. However, users will occasionally place a colored sheet behind photos, and thus the background should be measured for each image.

Equally, it is important to choose a threshold T so that the number of misclassified pixels are small. For example, in an image such as shown in Figure 1, the background is saturated white (i.e. gray level 255). The variance of the background color (as measured by scanning with the photos removed) is 2.97. Clearly a smaller value of T can be used than might be possible if the variance were larger.

In many cases the background color of a scan will be the dominant color in the histogram. Figure 7, for example shows the grayscale histogram of the scanned image in Figure 1. There is a large peak at gray level 255; in fact fully 31% of pixels in the image have this gray level. If every case were this simple background estimation would be a trivial issue. Consider the histogram in Figure 7 (b), however: it consists of the histogram of the image in Figure 10. Here the situation is complicated by the facts that:

1. there are several peaks in the histogram, and it is unclear which represents the background.
2. the background gray level itself occurs in many of the pixels interior to the photographs.

Rather than attempt to estimate Bg globally we use a method tuned to our segmentation algorithm. Recall that the algorithm simplifies by finding gaps, and that gaps are row or columns which contain almost entirely background pixels. Unless there are several rows and/or columns dominated by a single color, algorithm procMult() is unable to simplify. Thus we lose nothing by considering as candidates only those colors that account for a majority of pixels along at least a number of rows and columns. Even in cases such as Figure 10 this simple observation prunes the list of possible background colors to one. The background color may still not be well separated from data pixels, i.e. Bg may occur in many of the interior pixels, but catastrophic errors are prevented.

```
getBackground(Im){
    for i = 0, #rows-1{
        [maxClr, maxVal] = histogram(Im(i, 0:$j_{max}$));
        if (maxVal > 0.9*$j_{max}$)
            candClr[cnt] = maxClr;
            candVar[cnt++] = variance(Im(I,0:$j_{max}$));}
    for j = 0, #cols-1{
        [maxClr, maxVal] = histogram(Im(0:$i_{max}$, j));
        if (maxVal > 0.9*$i_{max}$)
            candClr[cnt] = maxClr;
            candVar[cnt++] = variance(Im(0:$i_{max}$, j));}
    [B, T] = mostCommonClr(candClr);
}
```

where [maxClr, maxVal] = histogram(Im); returns the most common color in Im as maxClr, and the number of pixels of that color as maxVal, and mostCommonClr(candClr) returns the most frequently occurring color in the list of candidate colors, and T, a threshold that is proportional to the variance. Since we have isolated the rows and columns that contain only background pixels we can estimate the background value, and variance with some confidence.

It is a simple matter to verify that any image that can be simplified by algorithm procMult(), will have the correct background color as a candidate color in the above procedure. In the case that more than one color dominates several rows and columns (an event that did not occur in a single one of the 139 test images mentioned in Section III C) we can choose the most common color, or even run the algorithm successively using each of the possible background colors.

V. Implementation issues

A. *Estimating Knee-points of Trapezoids*

While the method of Section IV almost always estimates the background color Bg correctly there is still the possibility that Bg is a color that occurs frequently in the interior of the rectangles. The image in Figure 10 is an example; even once we have accurately estimated Bg, there is the possibility that this color occurs frequently in the interior of the rectangular objects. In this case it is to be expected that classification will be imperfect, and the trapezoids mentioned in Section II and III will deviate considerably from their ideal shapes. This of course makes estimation of the knee-points of the trapezoids, and hence the vertices of the rectangular objects themselves error prone. An example is shown in Figure 12 where the P(i) and Q(j) functions of an actual image are shown; it is not hard to see that robust estimation of the knees might be difficult.

Again a simple observation helps: while pixels interior to a rectangle are sometimes classified as background the converse is seldom true. That is, while P(j) may often be lower than the true number of data pixels on row j, it will almost never be higher. This has the consequence that it is usually easy to estimate points a and d accurately, while b and c can be hard, and similarly e and h tend to be easy, while f and g are hard.

Once a and d have been determined doing a least mean squared, or least median squared fit of a trapezoid to the data P(i) and Q(j) will often give accurate estimates of the knee-points. Errors can occur however when the interior contains many background pixels. We have found that better results are found with the following approach.

B. *Accurately determining corners*

As explained in Section II, the knee-points of the trapezoids defined by P(j) and Q(i) for the single object case give the vertices of the single rectangular object. As we have seen noise, estimation errors and the presence of background color pixels in the interior of the rectangle make estimation of the knee-points only approximate. It is important to improve this estimation.

While we dismissed the possibility of exhaustively searching for corners in Section I, once their approximate location has been determined, even a crude method such as matched filtering may be adequate to improve the estimate of the location.

C. *Efficient calculation of P(j) and Q(i)*

Each call to procMult() requires calculation of the functions P(j) and Q(i) over that sub-image. This would require at least two complete passes through the sub-image. This is so, since each sub-image has already been processed by each higher layer of the recursion. An alternative is to use one pass through the entire image to pre-calculate an array that can be used to calculate P(j) and Q(i) for any sub-image. For example to calculate globalP:

```
for i = 0, #rows-1{
        for j = 0, #columns – 1{
                sum = 0;
                if ( |Im(i, j) – Bg| > Threshold)
                        sum++ ;
                globalP(i, j) = sum;}}
```

Similarly to calculate globalQ. Calculating these arrays requires the same amount of computation that is required to calculate P(j) and Q(i) of the overall image, but once done the P(j) and Q(i) functions for any sub-image Im(i_0:i_1, j_0:j_1) can be determined by

```
getProjections(Im(i_0:i_1, j_0:j_1)){
        for j = j_0, j_1
                P(j) = globalP(i_1, j) – globalP(i_0, j);
        for i = $i_0$, $i_1$
                Q(i) = globalQ(i, $j_1$) – globalQ(i, $j_0$);
}
```

Depending on the number of sub-images processed as procMult() recursively descends the saving in computation can be considerable.

D. Choice of metric

Our main requirement of the metric $|Im(i,j)|_{rgb}$ is that it allow us to clearly distinguish between data pixels and background pixels. An obvious choice is to transform color values to luminance grayscale and use, for example, the L_1 norm in the grayscale space. However, since many different colors can transform to the same gray level this can be a poor choice. For example, interior pixels can be incorrectly classified as background pixels if the color is different but the luminance is the same as Bg. We prefer choices where all three values have to be similar, for example

$|Im(i,j) - Bg|_{rgb} = \max(Im(i,j).R – Bg.R, Im(i,j).G – Bg.G, Im(i,j).B – Bg.B)$,

has the effect of forcing a pixel to match Bg in all color planes to be classified as a background pixel. However many other choices work equally well.

E. Alternatives to P(j) and Q(i)

In Section II we introduced the P(j) and Q(i) functions without justification. These are merely one dimensional projections of the image data which broke the two-dimensional segmentation problem into one-dimensional problems. Other one-dimensional projections might serve just as well. For example P(j) to be the distance between the rightmost and leftmost data pixel positions is also a good choice. In practice we have found this to be much more sensitive to noise.

F. Processing on Low Resolution Images

We have assumed that all processing is performed on an image Im(i,j). However, much of the processing can be performed on a low resolution copy of the image. For example, if an image is to be scanned at 600 dpi, it probably suffices to do most of the proceesing at 75dpi or so, reducing the complexity by a factor of $(600/75)^2 = 64$. Estimating the background color, calculating P(j) and Q(i), seeking gaps, and estimating the corners based on the trapezoid positions can all be done on a 75 dpi version of the image to reduce computational complexity. Accurate estimation of the corners (Section V B) can be done, starting from a low resolution estimate on the 600 dpi data.

G. Detecting non-rectangular objects

Our algorithm is designed to find and segment rectangular objects, and uses the fact that the projection of a rectangle is a trapezoid. Other geometric shapes such as disks, triangles also have simple easily identified projections, and the algorithm could be easily adapted to find them.

VI. CONCLUSION

We have introduced a method to automatically segment images comprised of rectangular objects on an approximately constant color background. The method is fast and efficient. It is robust in that it handles deviations from the ideal case well. It has been tested on a wide variety of images generated by users on real scanners, with very high success. The question of characterizing the class of images for which the algorithm fails remains open.

ACKNOWLEDGMENT

The author would like to thank Po Yuan, Issa Khoury and Shaun Ivory for helpful discussions.

REFERENCES

[1] C. Herley, “Recursive Method to Extract rectangular Objects from Scans,” in Proc. IEEE Intl. Conf. Image Proc. 2003, Barcelona.

[2] A. Jain, “Fundamentals of Image processing,” Prentice-Hall.

[3] R. L. de Queiroz, Personal Communication.

[4] A. Chaudhuri and S. Chaudhuri, “Robust Detection of Skew in Document Images,” IEEE Trans Image Proc, vol 6, No. 2, pp 344-349, February 1997.

[5] A. Hashizume, P. S. Yeh and A. Rosenfeld, “A method of detecting the orientation os aligned components,” Pattern Recognition Letters, col. 4, pp. 125-132, 1986.

[6] D. S. Lee, G. R. Thomas and H. Wechsler, “Auotmated page orientation and skew angle detection for binary documents images,” Pattern Recognition, pp. 1325-1344, Oct. 1994.

[7] G. Sharma, “Show-through Cancellation in Scans of Duplex Printed Documents,” IEEE Trans. On Image Proc., vol. 10, No. 5, pp 736-759, May 2001.

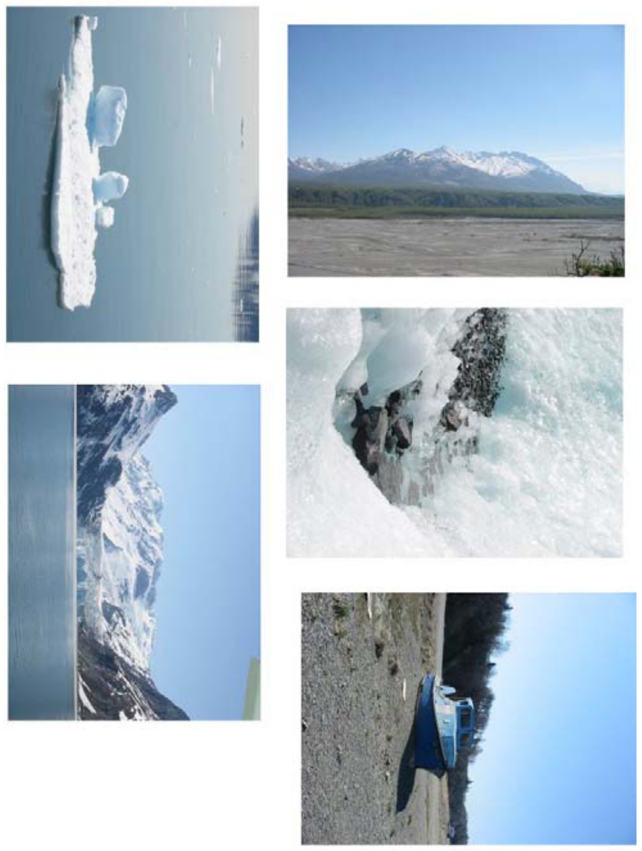

Figure 1: Example image consisting of several rectangular objects on constant color background.

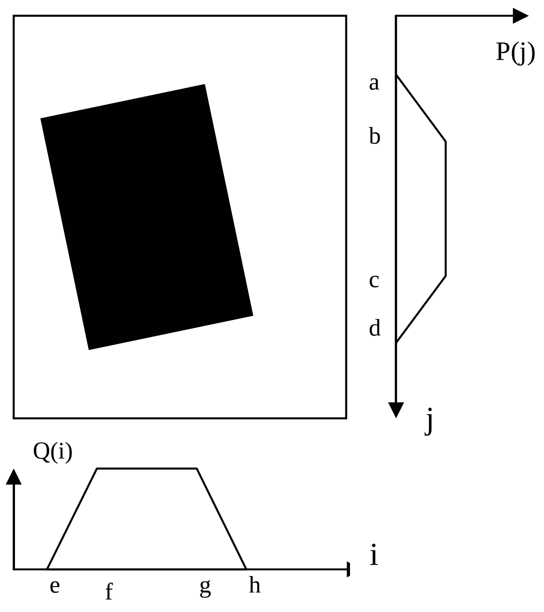

Figure 2: A single rectangular object. We graph P(j) and Q(i), the number of data pixels per row and per column respectively. Observe that the knee points of the trapezoids give the vertices of the rectangular object in the image.

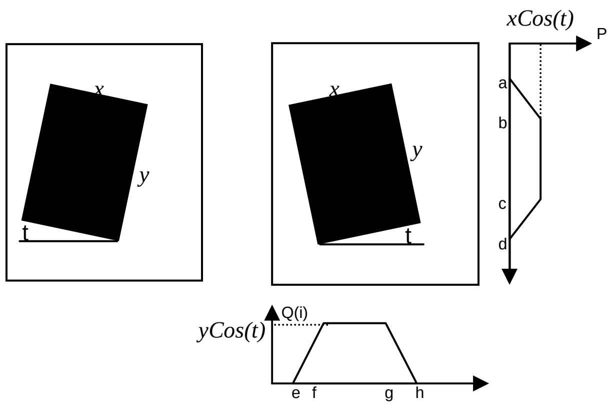

Figure 3: Two rectangular objects at different orientations give the same P(j) and Q(i) functions. To distinguish between these two possibilities one can explicitly check which rectangular object is present.

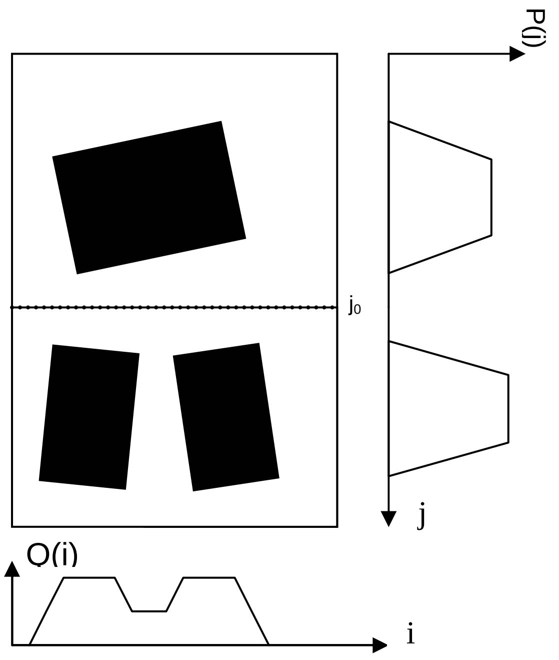

Figure 4: Example of three rectangular objects. The P(j) and Q(i) functions are now the sums of trapezoids. Observe that P(j) has a gap at row j_0, indicating that the problem can be simplified into sub-problems above and below this row.

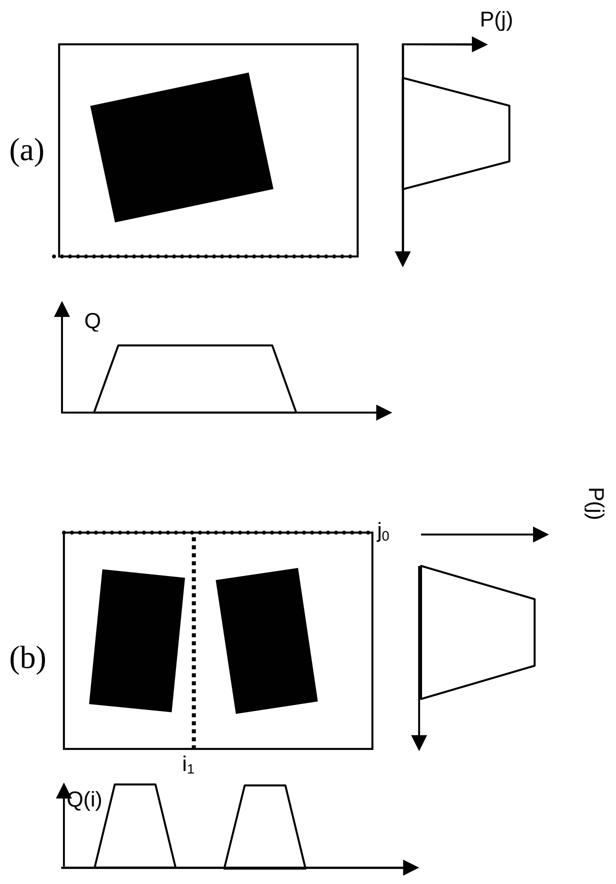

Figure 5: decoupling the example of Figure 3 by splitting above and below row j_0. We recalculate P(j) and Q(i) for the new sub-images. (a) We find the sub-image above j_0 is now the single object case (b) The sub-image below j_0 now has a new split in Q(j) at column i_1 (allowing further simplification).

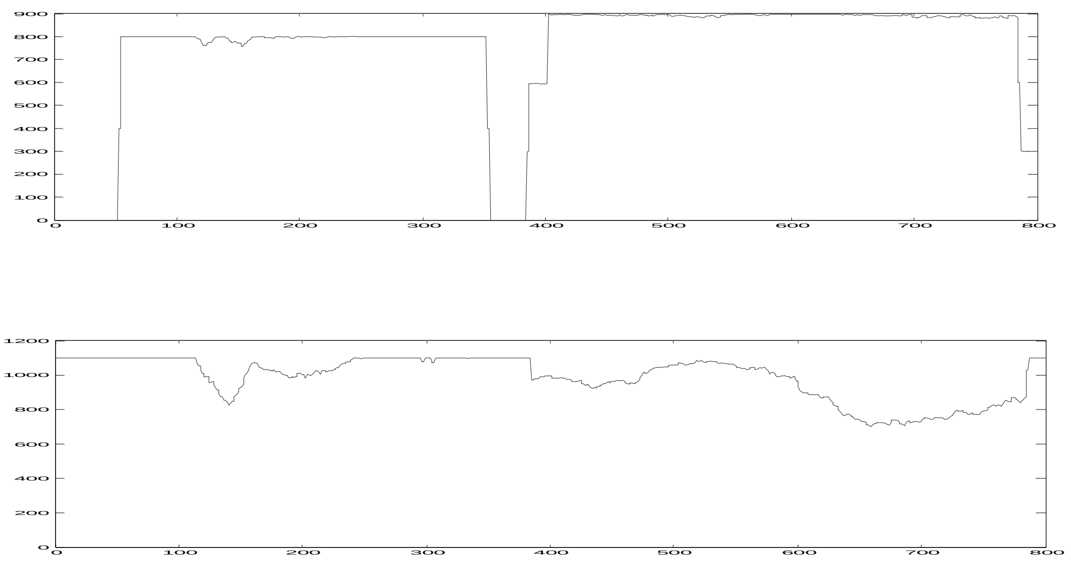

Figure 6: Importance of background estimate in calculating P(j) and Q(i). Illustrated is the Q(i) function for the image in Figure 1. Top shows Q(i) when the correct estimate for the background color is used; observe the gap which indicates the clearance between the two images on the left and the three on the right of Figure 1. The bottom graph shows the result of using an incorrect estimate. Clearly, incorrect background estimation can have disastrous effects on the ability of algorithm procMult() to exploit gaps.

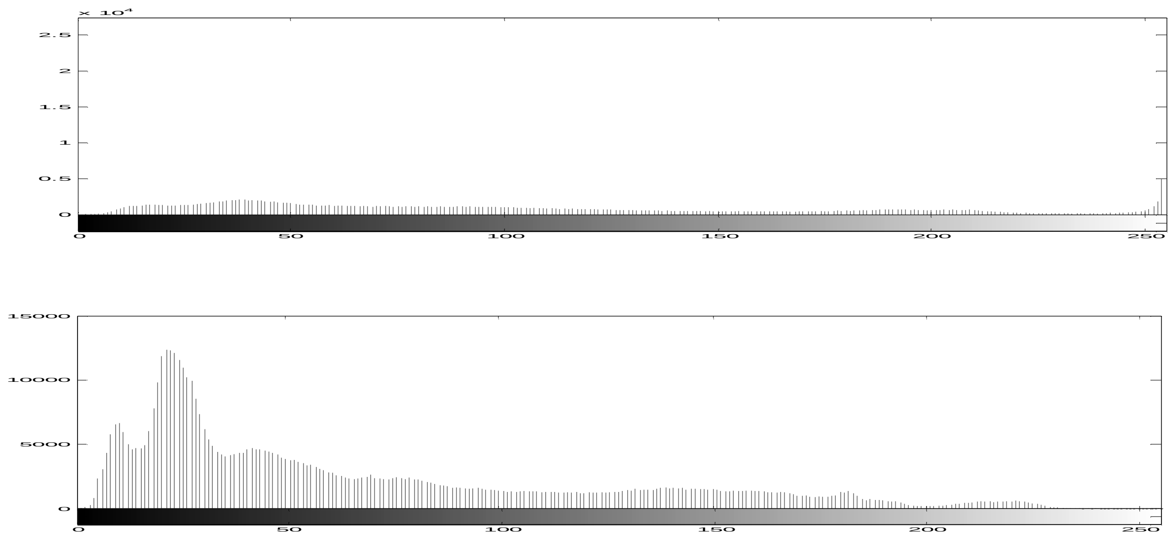

Figure 7. Grayscale histograms of two scanned images. Top: histogram of image in Figure 1. Bottom: image consists of four underexposed photos on a dark background. Note the peak at gray level 255 in the upper case allows easy estimation of the background color, while it is hard to determine the exact background color in the lower case.

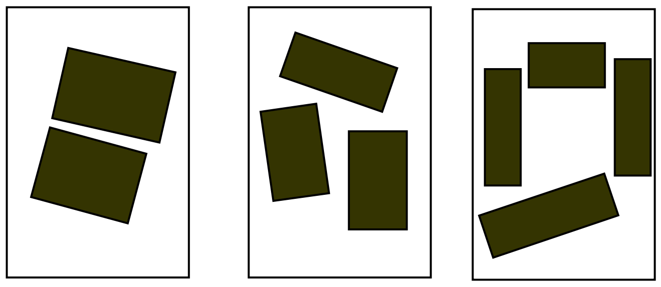

Figure 8. Examples of image that cannot be simplified with algorithm procMult(). In each case gaps in the P(j) and Q(i) functions are not found.

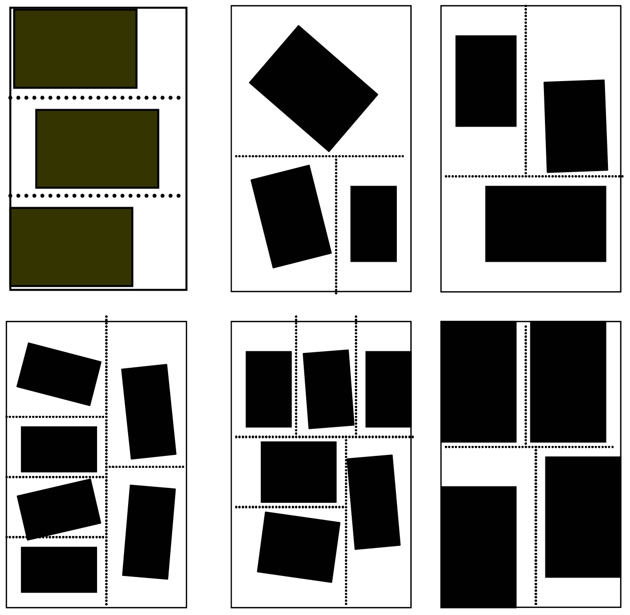

Figure 9: Examples of six images containing rectangular objects. Each of these is simplified all the way to sub-images containing single objects by algorithm procMult(). The dashed lines indicate the splits generated by the algorithm.

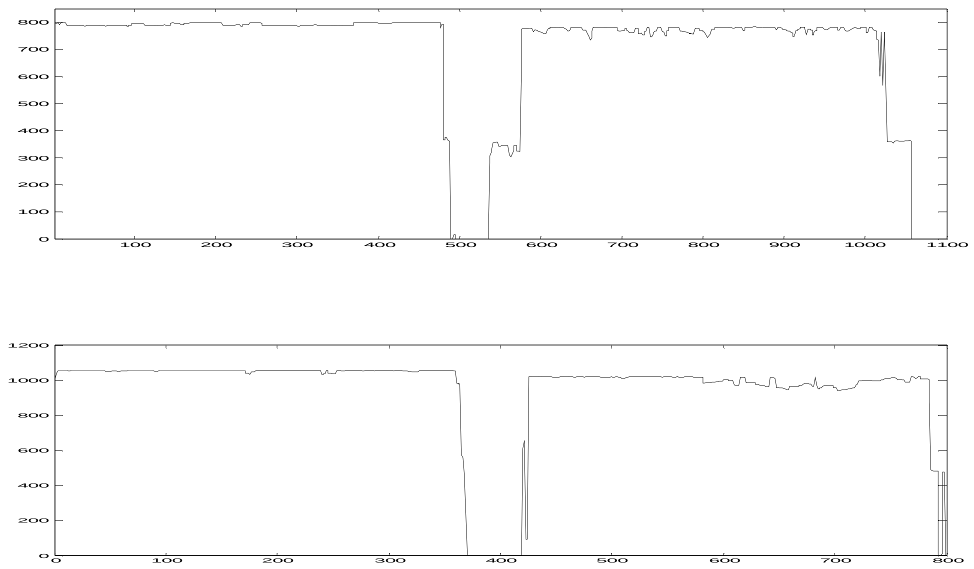

Figure 11: The P(j) and Q(i) projection functions for the image shown in Figure 10.

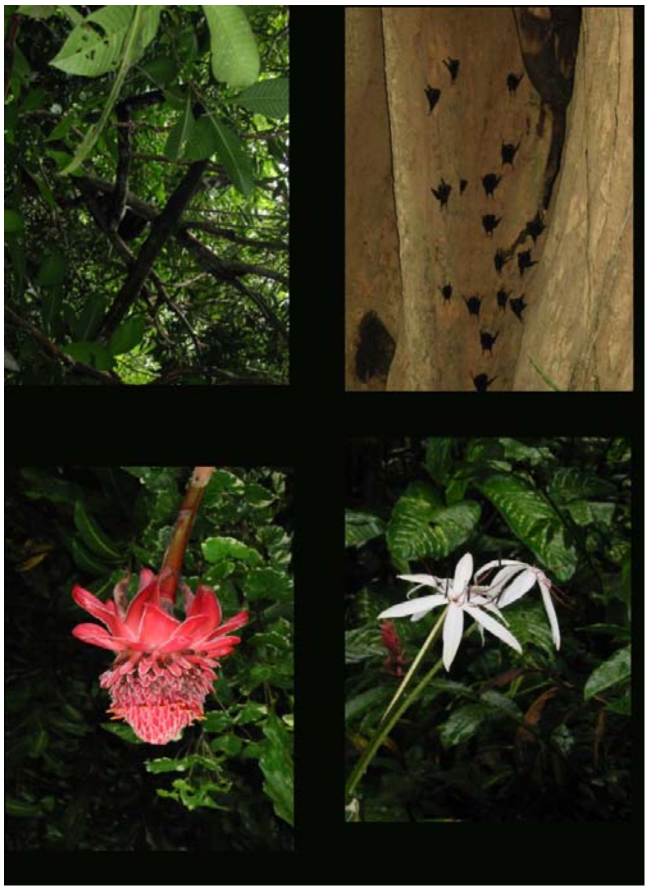

Figure 10. Image consisting of four dark photos on a black background. The background color is more difficult to estimate when it occurs frequently in the interior of the objects.

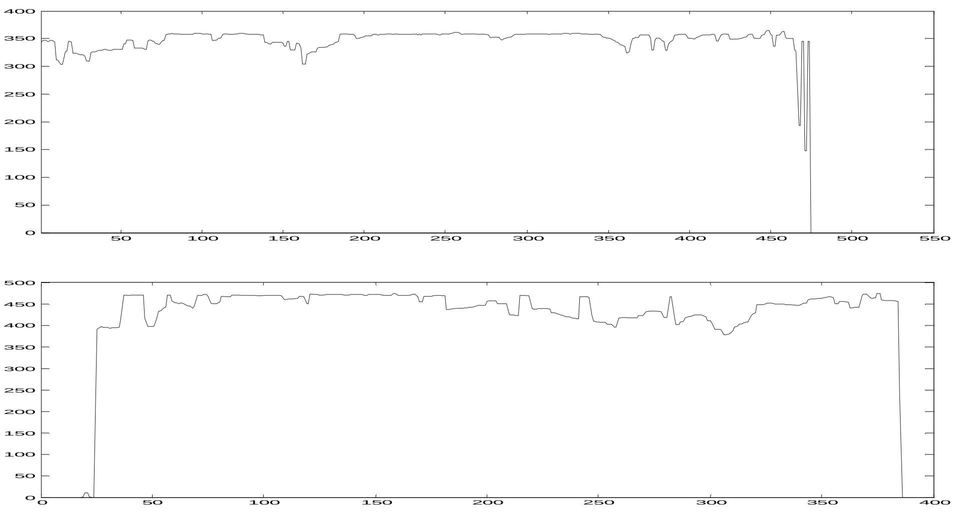

Figure 12. The P(j) and Q(i) projection functions for one of the sub-images of the image shown in Figure 10. The sub-image contains only a single photo; nonetheless the projection functions deviate considerably from the ideal trapezoidal shape. This illustrates the need for robust estimation of object corners.