Problem 1: Edge Detection

Motivation

Intuitively, edges can be found by identifying locations of sharply changing pixel intensities, i.e. large intensity gradients. A simpler edge detector such as **sobel** may achieve this by convolving the image with a high-pass filter, which averages the pixels on each side of the center pixel and calculates their difference. Typically, a gradient is calculated in the x direction and another in the y direction; then the two directional gradients can be combined into the full gradient with direction $\theta = atan(\frac{grad_y}{grad_y})$ and magnitude $r = \sqrt{(grad_y^2 + grad_x^2)}$. The 3by3 sobel filters in the x and y directions are:

$$\begin{bmatrix} 0.25 & 0 & -0.25 \\ 0.5 & 0 & -0.5 \\ 0.25 & 0 & -0.25 \end{bmatrix} \qquad \begin{bmatrix} -0.25 & -0.5 & -0.25 \\ 0 & 0 & 0 \\ 0.25 & 0.5 & 0.25 \end{bmatrix}$$

The magnitude r can then be thresholded to remove the smaller gradients which are likely noise. However, simply thresholding r turns out to be often overly simplistic and insufficient to distinguish between noise and edge.

Canny edge detector tries to address this by modifying sobel edge detector to include a gaussian filter pre-processing step, and non-maximum suppression and double-thresholding in postprocessing. Non-maximum suppression compares magnitudes along an edge gradient and only keeps the pixel whose gradient magnitude is the largest while setting all others to zero; this makes edges thin. Double-thresholding leverages the intuition that edges are often long lines of connected pixels, so it first uses a high magnitude threshold to confirm pixels that are definitely part of an edge, then it uses a low magnitude threshold to exclude pixels that are definitely not part of an edge, and finally for the ambiguous pixels, it keeps those that are connected with confirmed edge pixels and discards the rest as noise.

SE(Structured Edge) edge detector further refines edge detection by recognizing that features other than intensity such as texture and chrominance can also be indicative of edges. These features are harder to interpret with intuition, so SE uses machine learning techniques, which will be further explained in 1(c).

Methods

I first get the luminance matrix from RGB values using the given equation $Y = 0.2989R + 0.5870G + 0.1140B$. For **sobel** edge detector, I simply applied the filters given in motivation and $convolve()$ from hw1 to the luminance matrix to get $grad_x$ and $grad_y$ for each pixel, and then use the equations $\theta = atan(\frac{grad_y}{grad_y})$ and $r = \sqrt{(grad_y^2 + grad_x^2)}$ to get gradient directions and magnitudes. Since it's convolving with a filter of size 3by3, the image needs to be padded by width 1 before applying sobel filter. I'm using odd mirror padding in my solution since zero padding can create artificial gradients at the boundaries of the image. The magnitudes are then thresholded to get the edge map result. For thresholding, I first scaled the magnitudes to [0,255], and then construct its 256-bin cumulative histogram using hw1 codes; using the cumulative histogram, I then calculate a cutoff value by iterating 0→255 and getting the first bin with $bin_number/255 >= threshold$; all magnitudes below this threshold are considered noise.

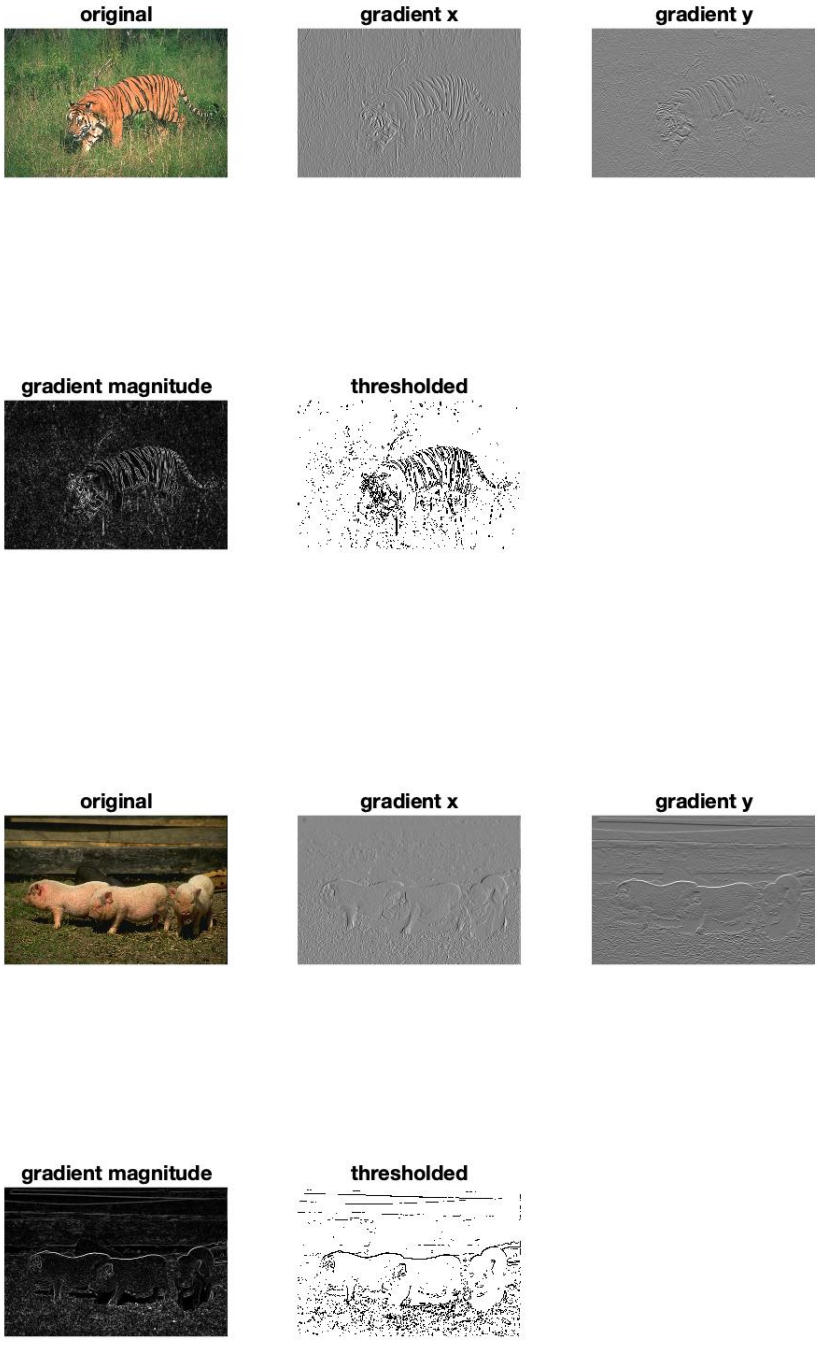
For **canny** edge detector, I used MATLAB's $edge(img, "canny", [threshold_{low}, threshold_{high}])$ function where the luminance matrix is again used as img.

For **SE** edge detector, I used the given **Structured Edge Detection Toolbox V3.0**, and essentially just copied codes in *edgesDemo.m*. I set *nms*(non-maximum-suppression) to true to get thinner edges. The codes return a probability map, which I threshold to get the edge map.

For evaluation, I calculate the precision and recall values using the $edgesEvalImg(E, G, "thrs", 1)$ function from the aforementioned toolbox. Although this function is supposed to take a probability map in E and does thresholding internally, I only give it already-thresholded binary edge maps and make it threshold at 0.5 for better compatibility with sobel and canny results.

Results and Discussion

(a)



The images above are obtained with cdf threshold percentage of 0.9, which means only 10% of the

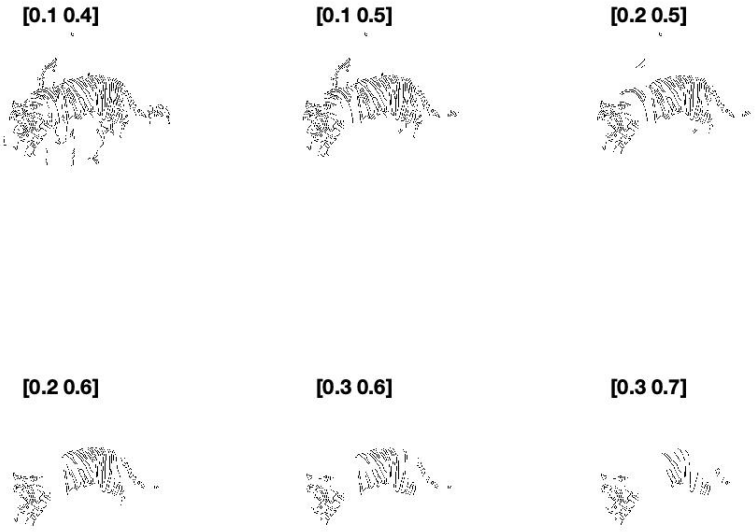
largest gradient magnitudes are counted as edges. Higher thresholds means higher precision but lower recall.

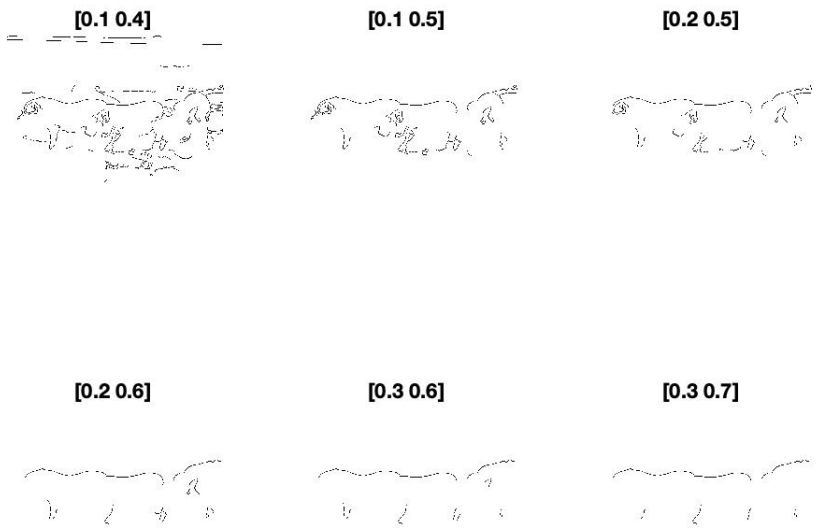
(b)

(1) For each pixel, NMS compares its gradient magnitude r with those of other pixels that are within a certain distance from the current pixel and are on the same line drawn along the current gradient direction θ; it keeps the pixel with the largest r and throws away the rest, which thins the edges.

(2) As explained earlier, the high threshold is used for confirming sure edges and the low threshold is used for excluding sure non-edges. So individually, increasing each would mean higher precision and lower recall. The two thresholds can also be discussed in terms of the gap between them. Larger gap between the low and high thresholds would mean leaving more candidate edges to be decided by their connections, so intuitively, this would mean less sporadic dots and more connected lines on the edge map.

(3)





The results mostly align with the explanations in (2), where higher thresholds suppress noise but also remove wanted edges. A fine example of larger gap between low and high thresholds causing more connected edges can be found between the pig images using [0.1 0.4] and [0.1 0.5] as thresholds. Generally speaking, canny edge detector results are much cleaner compared to sobel edge detector results.

(c)

(1) To sum up at a high level, SE edge detector tries to decide whether a pixel is part of an edge by looking at its surrounding patch and trying to match the patch to one of the "patterns" it has learned. The intuition here is that edges tend to follow certain patterns, and so long as the edge detector knows a discrete set of such patterns, and also how to match the patch surrounding a candidate edge pixel to these patterns, it will be able to distinguish the edge pixels. Thus, the problem can now be broken down to "how to get the set of candidate patterns" and "how to match a given patch against these patterns".

To get the set of patterns, SE considers 16by16 binary edge maps.[1] Since there are 256 pixels in the mask and every pixel can be {0,1}, there are 2^{256} such 16by16 maps possible, i.e. 2^{256} patterns. This is too large a number to handle, so SE instead uses an encoding defined by whether every pair of pixels in the 16by16 binary map belong to the same segment, recorded in a binary vector; there are $\binom{256}{2} = 32640$ ways to choose a pair of pixels from the 16by16 binary map, so the size of "pattern space" is reduced from 2^{256} to 32640. SE further reduces this number by only choosing a random subset of size m of such patterns, and then using either clustering or PCA to choose the k most representative patterns of this subset. It might seem SE is throwing away too much information but that's actually fine. The next paragraph explains why in more details.

To match a given patch at runtime against the set of patterns, SE uses random forest. Random

[1] It doesn't have to be binary but we consider the binary case for simplicity's sake.

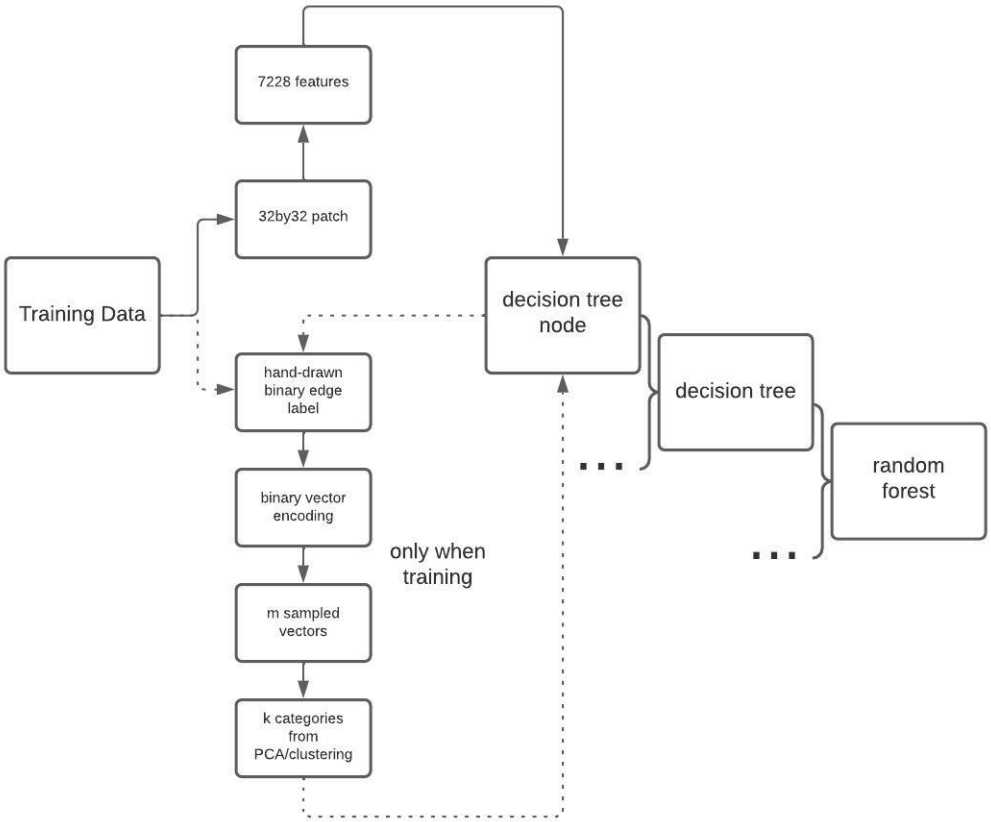
forest consists of decision trees, which in turn consists of decision nodes arranged in a tree structure. Given the input features, each node checks a single feature against some cutoff values and passes the input to one of its children depending on the result. Which feature to check and what cutoff values to use are learned in training. During training, the tree is given some labeled data (in this case patches and their corresponding binary vectors), and each node in the tree learns the feature and cutoff values that optimize "information gain" by splitting these data into groups that are similar within. Within-group similarity can be measured by variance in the continuous case, or Shannon entropy/Gini impurity in the discrete case. For example, the formula for Shannon entropy is

$$H(S) = -\Sigma_y p_y log(p_y)$$

where p_y is the possibility of a case belonging to the categorical label y occurring in the group. Intuitively, this measure is the best when all cases in the group belong to the same categorical label y. Knowing how decision trees are trained also helps understand why SE needs relatively small number of "candidate patterns" (imagine checking for the probabilities of occurrences for 2^{256} categorical labels, at every node!). In the last paragraph it was explained how SE reduced the number of candidate patterns from 2^{256} to $\binom{256}{2}$ with the binary vector encoding, from $\binom{256}{2}$ to m with random sampling, and from m to k with clustering or PCA, and there was concern that this might throw away too much information. Well, such aggressive procedure is actually fine because this is done for **each node**, of which there can be many in a decision tree, and there can be many decision trees in a random forest, so any bias created by sampling will likely cancel each other out.

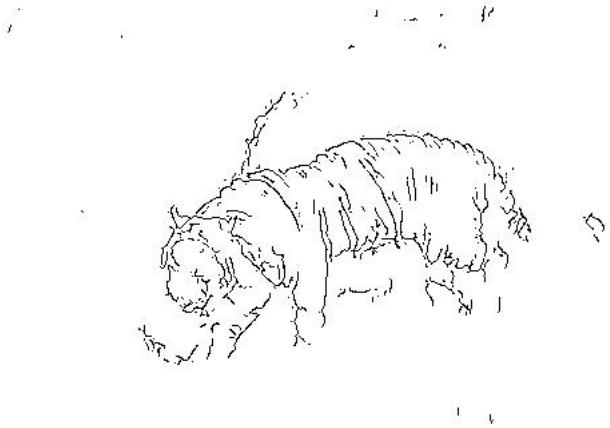
The features fed into the random forest as input has a high dimension at 7228 when the patch size is 32by32. The literature background here is too deep for me to explore, but suffice to say that there are generally two kinds of features. First there's the features associated with each individual pixel in the patch, including colors, gradient magnitudes, and orientations. Then there's the pairwise differences which checks each pixel against all other pixels.

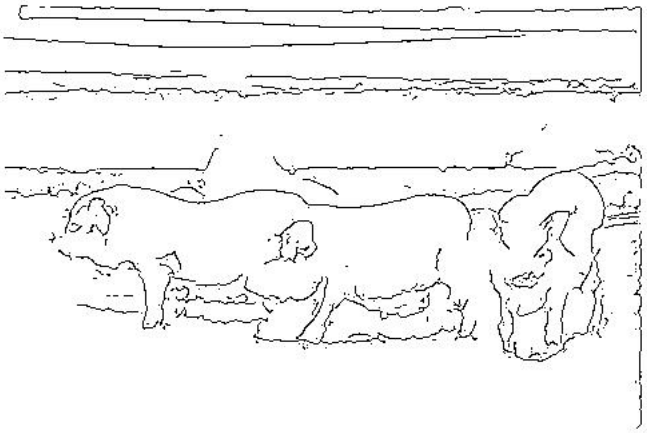
Finally, a flow chart to summarize SE detector:



(2) Please see (1) paragraph 3.

(3)





These images are generated with $nms = 1$ and all other parameters the same as in *edgesDemo.m*. The probability map is then thresholded with value 0.1. Please see methods for justifications. Compared to the results of canny detector, SE detector's results tend to include fewer messy noises such as tiger stripe edges, which aligns better with how people generally focus more on the contour and less on the inner details.

(d)

(1)

	P	R	F
gt1	0.0964	0.8955	0.1741
gt2	0.1052	0.9213	0.1889
gt3	0.1188	0.9269	0.2107
gt4	0.4751	0.9399	0.6312
gt5	0.1101	0.7405	0.1918
avg	0.1811	0.8848	0.2793

Table 1: Sobel on tiger image

	P	R	F
gt1	0.1574	0.7919	0.2627
gt2	0.1802	0.8548	0.2977
gt3	0.1970	0.8322	0.3186
gt4	0.7945	0.8512	0.8219
gt5	0.1897	0.6908	0.2976
avg	0.3038	0.8042	0.3997

Table 2: Canny on tiger image

	P	R	F
gt1	0.2833	0.6929	0.4021
gt2	0.3248	0.7494	0.4532
gt3	0.3597	0.7390	0.4839
gt4	0.8797	0.4584	0.6027
gt5	0.3439	0.6092	0.4397
avg	0.4383	0.6498	0.4763

Table 3: SE on tiger image

	P	R	F
gt1	0.1417	0.6805	0.2345
gt2	0.1472	0.6540	0.2403
gt3	0.2082	0.5776	0.3061
gt4	0.2593	0.5724	0.3569
gt5	0.2110	0.5997	0.3122
avg	0.1935	0.6168	0.2900

Table 4: Sobel on pig image

	P	R	F
gt1	0.3284	0.6612	0.4388
gt2	0.3437	0.6401	0.4473
gt3	0.4157	0.4835	0.4471
gt4	0.5158	0.4773	0.4958
gt5	0.4191	0.4993	0.4557
avg	0.4046	0.5523	0.4569

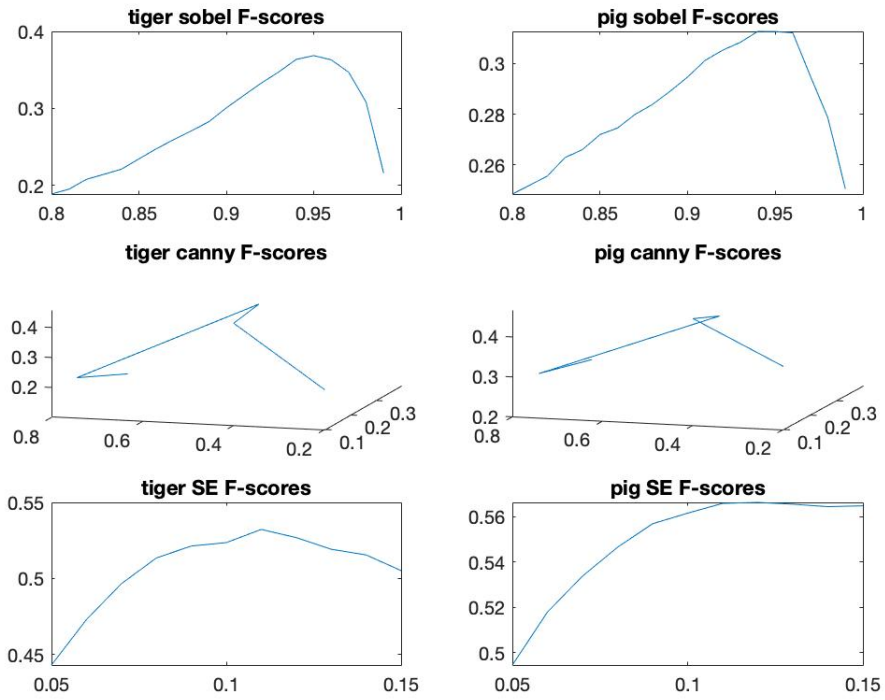
Table 5: Canny on pig image

	P	R	F
gt1	0.2740	0.8217	0.4109
gt2	0.2969	0.8234	0.4364
gt3	0.4738	0.8208	0.6008
gt4	0.6251	0.8616	0.7246
gt5	0.4558	0.8087	0.5830
avg	0.4251	0.8272	0.5511

Table 6: SE on pig image

The overall F scores calculated from the precision and recall scores averaged over ground truths, which is not the same as simply averaging the F scores calculated for each ground truth, are 0.3007, 0.4410, 0.5236, 0.2946, 0.4670, 0.5614, presented in the same order as the tables were presented. It is clear that in terms of F-scores, SE > canny > sobel. If we look deeper into the precision and recall scores, we may see sobel tends to give low precision but high recall scores, i.e. it tends to find more edges, but many are wrongfully identified non-edges. By comparison, SE gives lower recall but much higher precision scores; it might miss some edges, but those that it identify as edges are much more likely to be true positives. Canny is somewhere in the middle. The values in the tables shown above were calculated using the same thresholds as the ones I used for each edge detector, i.e. 0.9 for sobel, [0.1,0.4] for canny, 0.1 for SE.

(2)



The best F-scores are 0.3686, 0.4566, 0.5322, 0.3126, 0.4670, 0.5664, presented in the same order as tables in (1). It should be noted that threshold values don't quite have the same meaning for the three detectors. Due to sobel's tendency of giving false positives, I only tested it with thresholds in the high range. For both tiger and pig images, sobel's F-scores peak at about 0.95 threshold, i.e. when leaving only the top 5% magnitudes as edges. Thresholds lower than this give too many false positives and higher thresholds don't identify enough edges. It's harder to comment on canny's F-scores because it has two thresholds and I didn't test enough thresholds to fill the parameter space due to time constraints, but suffice to say the tradeoff between precision and recall is also present here and F-score peaks when the high threshold is around 0.4 and the low threshold around 0.15. For SE I only tested low thresholds because the probability map it returns tends to consist of small values; F-score peaks at around 0.12.

(3) Pig; the grassland in the tiger image has many places of interleaved green and yellow, which might be wrongfully identified as edges if we just look at the intensity gradient. Although this is also present in the pig image, it's to a lesser extent and concentrated mostly in the bottom half of the image.

(4) No. The design of the F-score equation is such that the nominator is more sensitive to change by factor than the denominator. Consider two pairs of P and R values $P = 4x, R = 2x$ and $P = 8x.R = x$; F-score for the former is $2\frac{8x^2}{6x} = \frac{8x}{3}$, and F-score for the latter is $2\frac{8x^2}{9x} = \frac{16x}{9}$. For a more extreme example, F-score will be zero if one of P or R is zero.
If $P+R = C$ where C is a constant, the F-score equation can be written as $2 * \frac{P(C-P)}{C}$. Since 2 and C are constants, maximizing $P(C-P)$ is the same as maximizing this equation. $\frac{\partial}{\partial P}P(C-P) = C - 2P$. $C - 2P = 0$ when $C = 2P$ and $R = C - P = 2P - P = P$.[2]

Problem 2: Digital Half-toning

Motivation

As the name implies, half-toning is the process of converting the continuous pixel values in an image to binary values for purposes such as printing. Ideally, the process should leverage the fact that human eyes are essentially low-pass filters that are capable of smoothing out abrupt changes in pixel values, and "trick" human eyes into seeing continuous intensities out of binary intensities by having more evenly-spread absent pixels in darker regions, and present pixels in brighter regions.
In this assignment four kinds of half-toning are attempted. Fixed-thresholding simply sets all pixel below a fixed threshold as absent and all those above as present. Random thresholding makes a slight modification by randomly generating a threshold for every pixel. These two are overly simplistic and are mostly there for reference. Dithering makes use of a thresholding matrix. The thresholds roughly span the [0,255] range, and high and low thresholds are interleaved to help present and absent pixels spread more evenly, so human eyes can better average the pixels to get a more pleasant sense of continuous intensity. Finally, whereas the three aforementioned approaches focus on changing the threshold, error diffusion keeps the same threshold across the entire image, but modify the pixel values themselves according to whether a pixel's neighbors are rounded up or down. For example, if the threshold is 128 and a pixel has intensity 129, this pixel will be rounded up to 255, which obviously creates a large (or small, technically) negative error of -126. We talked about how human eyes can average neighboring pixels, so this negative error can be compensated by tuning down the pixels around, hopefully creating more absent pixels in the region and cancelling out the extra intensity.

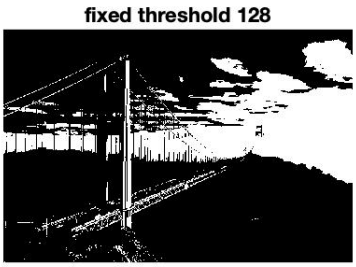
Methods

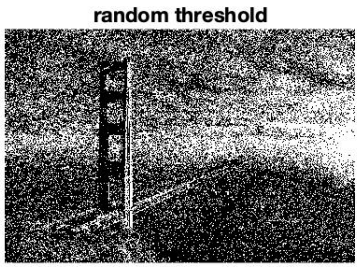
Fixed and random thresholdings are straightforward. For dithering, the dithering matrix formula is obviously calling for a recursive definition, so I created a recursive function that takes the dithering matrix size as parameter and fills its four quadrants by calling itself with half the input size. I hope I can be excused for using MATLAB's matrix manipulation here, since the question specification only asks for trying dithering matrices of 3 fixed sizes, which I could have just hard-coded. After generating the dithering matrix, I get the threshold matrix by applying the given equation to the dithering matrix. It should be noted that no rounding or clipping is done at this step, since only the "final" image needs to be of the unsigned 8-bit integer format. For error diffusion, I implemented serpentine scanning by scanning $1 \rightarrow 600$ on odd rows and $600 \rightarrow 1$ on even rows. Diffusion matrices are also "flipped" left and right on even rows by traversing its columns in the reverse order. On the image edges, if an error diffusion goes out of bounds, it is thrown away for simplicity. All else was clearly stated in the hw instructions and done accordingly.
NOTE: images shown here are scaled down for display. If they don't look satisfactory, please run prob_2.m and check the originals.

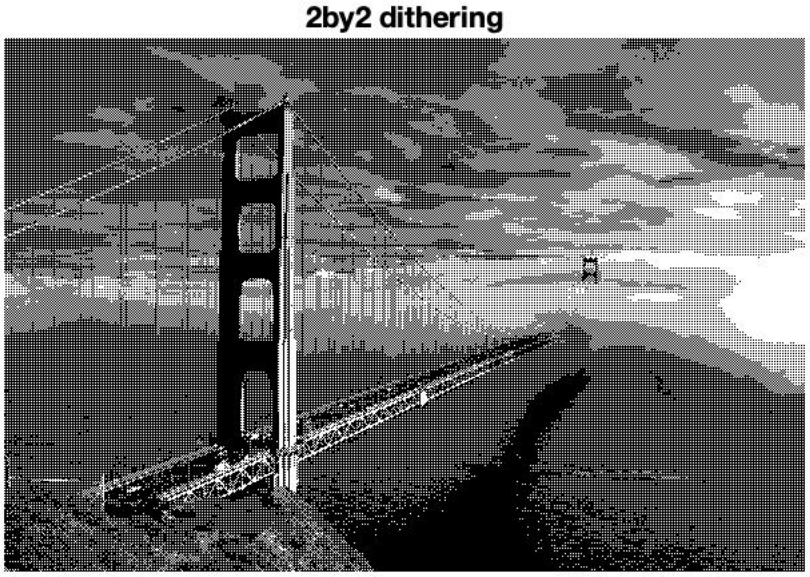
[2] second derivative is -2 so this stationary point is confirmed maximum

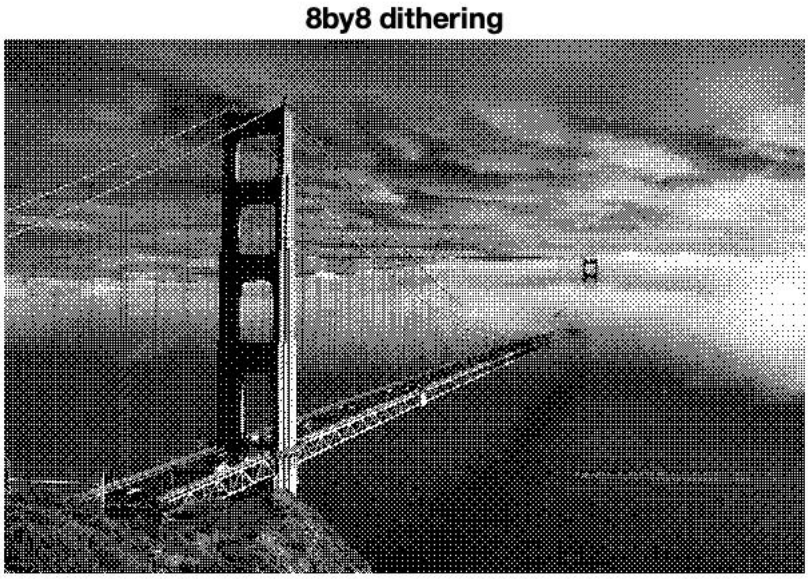
Results and Discussion

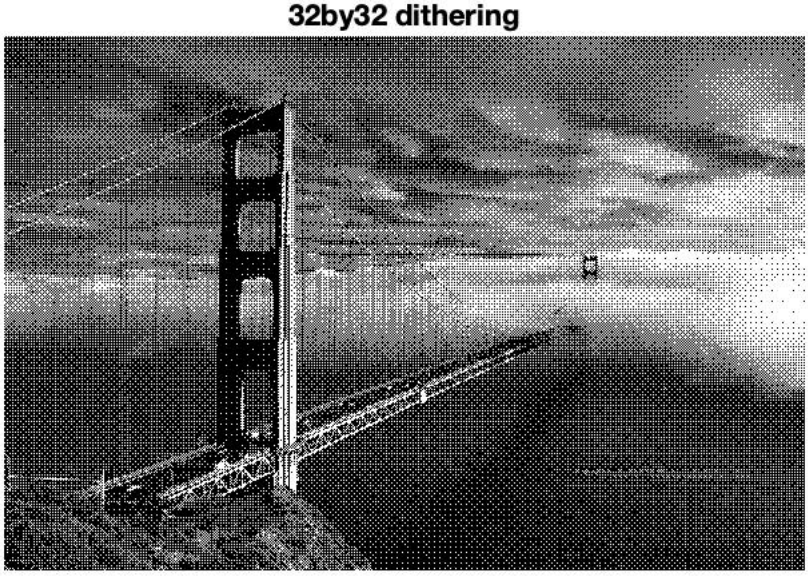
(a)











95.6250	159.3750
223.1250	31.8750

Table 7: I_2 thresholds

As expected, results for fixed and random thresholding are not great. For fixed thresholding, large continuous areas of salient white or dark are present, providing little opportunity for the human eye

85.6641	149.4141	101.6016	165.3516	89.6484	153.3984	105.5859	169.3359
213.1641	21.9141	229.1016	37.8516	217.1484	25.8984	233.0859	41.8359
117.5391	181.2891	69.7266	133.4766	121.5234	185.2734	73.7109	137.4609
245.0391	53.7891	197.2266	5.9766	249.0234	57.7734	201.2109	9.9609
93.6328	157.3828	109.5703	173.3203	81.6797	145.4297	97.6172	161.3672
221.1328	29.8828	237.0703	45.8203	209.1797	17.9297	225.1172	33.8672
125.5078	189.2578	77.6953	141.4453	113.5547	177.3047	65.7422	129.4922
253.0078	61.7578	205.1953	13.9453	241.0547	49.8047	193.2422	1.9922

Table 8: I_8 thresholds

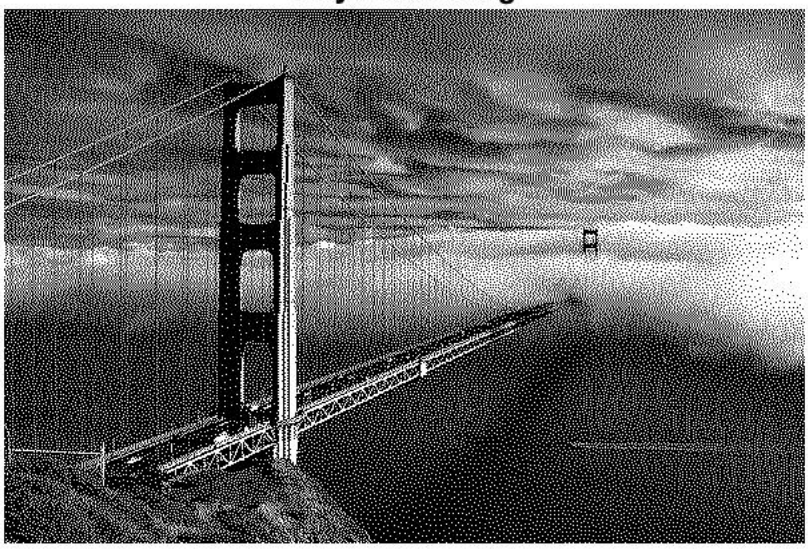
85.0415	148.7915	100.9790	164.7290	89.0259	152.7759	104.9634	168.7134
212.5415	21.2915	228.4790	37.2290	216.5259	25.2759	232.4634	41.2134
116.9165	180.6665	69.1040	132.8540	120.9009	184.6509	73.0884	136.8384
244.4165	53.1665	196.6040	5.3540	248.4009	57.1509	200.5884	9.3384
93.0103	156.7603	108.9478	172.6978	81.0571	144.8071	96.9946	160.7446
220.5103	29.2603	236.4478	45.1978	208.5571	17.3071	224.4946	33.2446
124.8853	188.6353	77.0728	140.8228	112.9321	176.6821	65.1196	128.8696
252.3853	61.1353	204.5728	13.3228	240.4321	49.1821	192.6196	1.3696

Table 9: I_{32}(1:8,1:8) thresholds

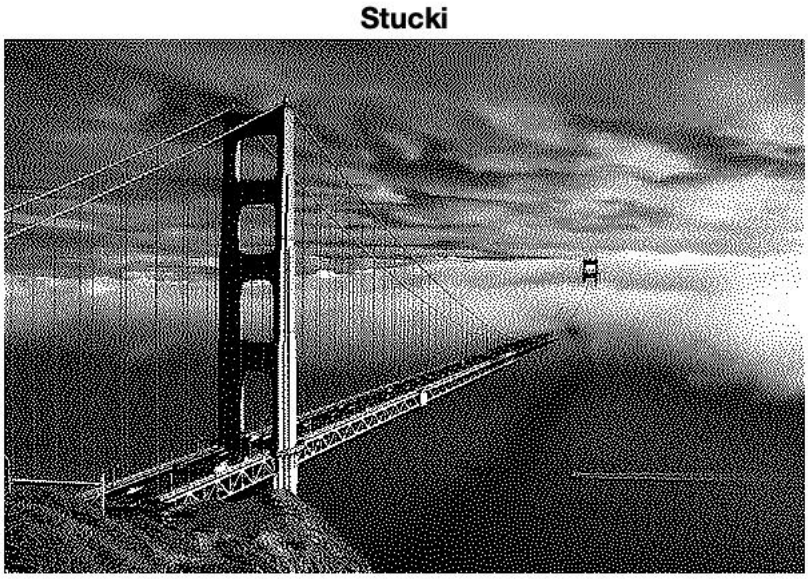
to average. Random thresholding improves this slightly since the randomness in thresholds disrupts the continuous similar-intensity areas and, although rough, achieves an effect where brighter areas have more present pixels that are somewhat evenly-spread. However, this randomness also creates salt-pepper noise. Dithering gives much more pleasant results. The grid-like texture is intentionally created by dithering matrix to arrange the present and absent pixels in a more even and orderly way, though one might argue it still looks somewhat uncomfortable. Among the three dithering results, I_2 is the best at keeping the edges that are present in the original image, since it processes fewer pixels at a time and thus doesn't allow as much information to "leak" among pixels; however, there's also the least variety in intensity since the thresholding matrix is small and cannot accommodate many combinations of pixel patterns. I_{32} is the opposite, with the most blurred edges (though still not obvious), but also the most variety in intensities. I_8 is in the middle.

(b)

Floyd-Steinberg



JJN

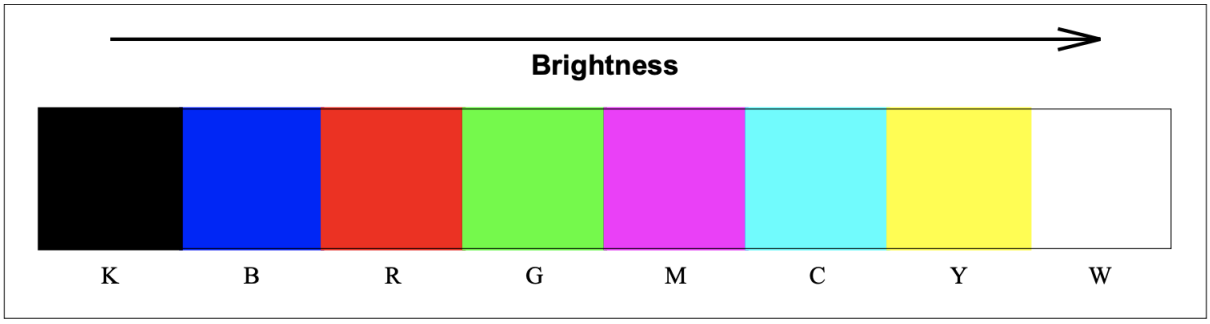


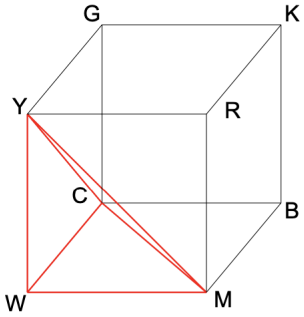
First of all I think JJN and Stucki give better results than Floyd-Steinburg. I'm probably stretching it but if you zoom a bit closer you can still see small areas of continuous white or dark that are not quite broken up in Floyd-Steinburg. This might be because of the smaller matrix size and therefore less information leaked to neighboring pixels. This is less obvious in JJN and Stucki results since they have larger matrices. Although theoretically this would also blur the edges, the blur is actually not obvious, perhaps because 5by5 is not much larger than 3by3 and by its very definition, half-toning relies on damaging resolution to some extent. Actually, I think I'll go a step further and claim that even larger error diffusion matrices will give even better results. It's a very close call between JJN and Stucki, but I'd pick Stucki since it better preserves the mirror of the bridge in the water. I think this is because it diffuses the error less aggressively with smaller weights and thus causes less blur.
For potential improvements, I'd say try larger diffusion matrices as explained above. Also, instead of throwing away error diffusion that goes out of the image bounds, maybe there can be some way to distribute the out-of-bounds-error across the neighborhood by for example, having a special variant of the diffusion matrix for pixels on the image bounds that folds the original matrix and adds out-of-bounds weights to their symmetrical counterparts.

Problem 3: Color Half-toning with Error Diffusion

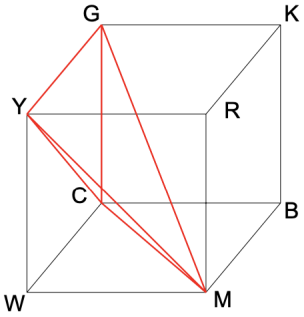
Motivation

Color Half-toning is an extension of the grayscale half-toning to rgb images. The most obvious method of doing color half-toning is simply doing monotone half-toning on each of the rgb channels independently. However, this violates the principle of MBVC (Minimal Brightness Variation Criterion), which is self-explanatory by name. It turns out that if each color channel is half-toned independently, a lot of pixels will end up becoming white or dark, creating unpleasant artifacts for the eye. MBVQ (Minimal Brightness Variation Quadrant) half-toning was designed to address this issue by separating the color space into constraining quadrants. Pixels that fall into a quadrant can only be rounded to one of its four vertices, which prevents large brightness variations.

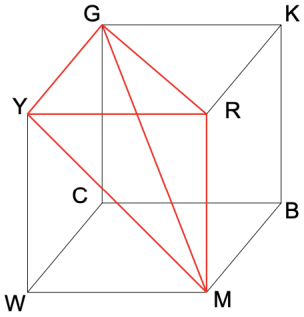




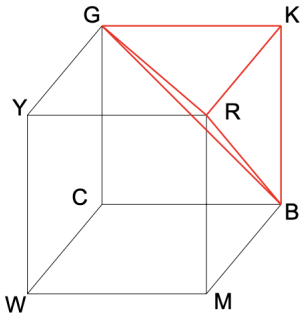
CMYW



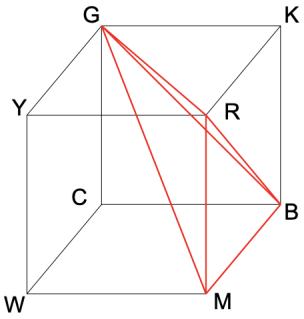
MYGC



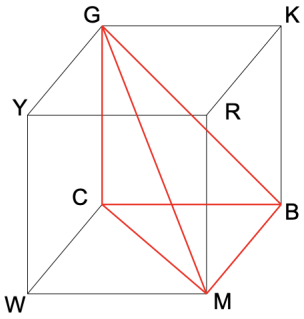
RGMY



KRGB



RGBM



CMGB

It should also be noted that when printing, RGB images need to be converted into CMY format since printing uses subtractive coloring. The convertion can be easily done with the equation

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 255 \\ 255 \\ 255 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

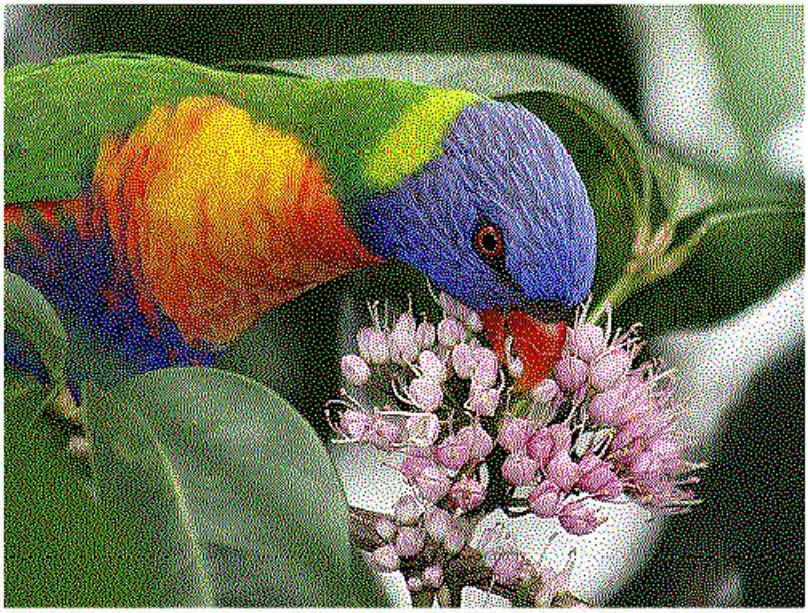
Methods

For separable error diffusion, the image is first converted into CMY color space; then Floyd-Steinberg error diffusion is done independently for each of the CMY channels using 128 as threshold. Finally the 3 processed CMY image is converted back to RGB for display.
For MBVQ error diffusion, I mostly follow [1] by first finding the quadrant and then using the provided

codes to find the closest vertex. The rgb values from the closet vertex are then subtracted from the real rgb values to get the errors in each of the rgb channels, and the current pixel is rounded to the closet pixel. The rest is the same as separable color diffusion. No RGB-CMY conversion is done for MBVQ error diffusion.

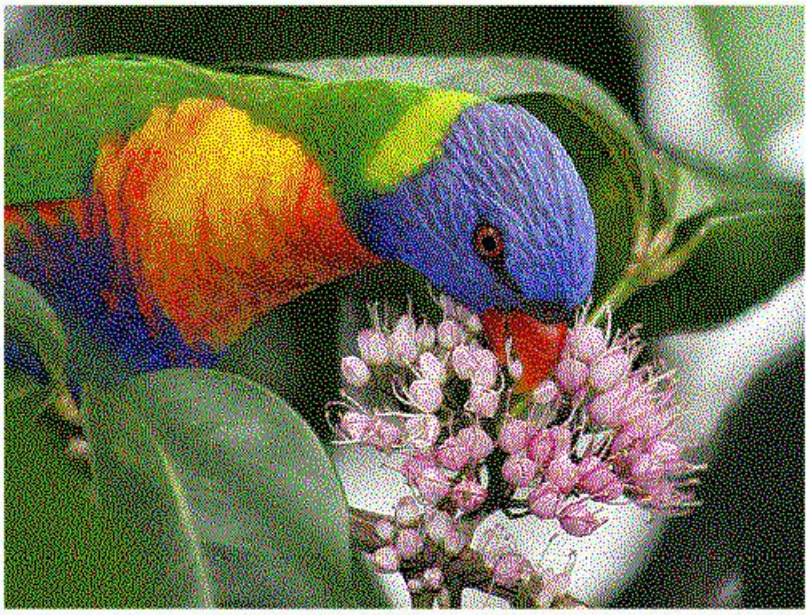
Results and Discussion

(a)



Compared to the MBVQ result, this image looks a little whitish, and indeed, if you zoom closer to the pedals for example, you can see that there are many more white/black pixels. As mentioned earlier, this was the motivation for MBVQ.

(b)



Please see paragraph (2) for how MBVQ mitigates brightness variation by constraining quantization to sub-quadrants. Again, the resulting image contains fewer white/black pixels and therefore looks better than the seperable diffusion result.

References

[1] Doron Shaked et al. "Color diffusion: error diffusion for color halftones". In: *Color Imaging: Device-Independent Color, Color Hardcopy, and Graphic Arts IV*. Ed. by Giordano B. Beretta and Reiner Eschbach. Vol. 3648. International Society for Optics and Photonics. SPIE, 1998, pp. 459–465. DOI: 10.1117/12.334589. URL: https://doi.org/10.1117/12.334589.