CS131 Computer Vision: Foundations and Applications (Fall 2015) Problem Set 0

This homework is intended to test your knowledge about the major prerequisites for the class. For some problems, you may need to use Google and/or Wikipedia to brush up on topics with which you're unfamiliar. You are also encouraged to review the Linear Algebra and Matlab tutorials.

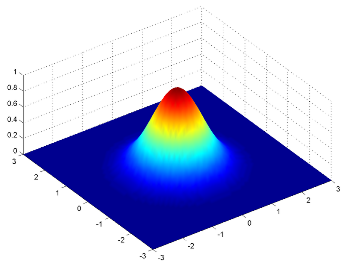
1 Calculus Review

(a) The height of a projectile at time t is given by

$$y(t) = v_0 t - 5t^2, \tag{1}$$

where v_0 is a constant. Using this information, find the peak height (the max value of y) in terms of v_0. *Hint: Take the derivative of $y(t)$ with respect to t, and then solve for the value of t that makes the derivative equal 0. Use this to give the peak height (the max value of y) in terms of v_0.*

(b)



The formula for the 2D Gaussian function above is

$$f(x,y) = e^{\dfrac{-(x-x_0)^2-(y-y_0)^2}{2\sigma^2}}. \tag{2}$$

We will use calculus to find the peak (though the peak might be fairly obvious in this example, please answer all steps below, to demonstrate that you can find it with calculus).

(b.i) Treat y as a constant, and give the derivative with respect to x.

(b.ii) Set the above derivative equal to 0 and solve for x. This maximizes the value of x for any given y. *Hint: find the value of x which makes one of the factors in the derivative zero. We can decide this is a maximum and not a minimum by looking at the given plot. We could have also used the second derivative* *to confirm that there is negative curvature at that point.*

(b.iii) Plug the maximizing x from part (b.ii) back into the original equation. Then find the derivative of that expression with respect to y.

(b.iv) Set the above derivative equal to 0 and solve for y. Where is the peak of the 2D Gaussian function? What is its value?

2 Image Manipulation

Remember, you can use the MATLAB "doc" function to learn how to use a command (e.g. "doc rgb2gray"). Also, the problems below can be easily solved without writing a single 'for' loop, if you use matrix math in your MATLAB code (a.k.a. "vectorization").

(a) The provided **imageManip.m** script reads in the provided u2dark.png photo and converts it to grayscale using `rgb2gray`. Use the script and your own code to calculate the following statistics: What is the average pixel value of the resulting grayscale image? What are the min and max values? (There are several ways to calculate these quantities in MATLAB. Review the MATLAB tutorial if you need help.)

(b) We would like to bring the image to a more typical average brightness. Add an offset and apply a scaling factor to all pixels, so that the minimum pixel value becomes 0 and the max pixel value becomes 255. *(Cameras often do a similar function automatically.)* Include the final image in your report, as well as the MATLAB code you used to produce it.

(c) Next, we would like to double the contrast of the pixels in the middle brightness range. Specifically, take your result from part (b) and replace each pixel's intensity i with a new intensity i', where

$$i' = 2 * (i - 128) + 128$$

Threshold i' so that $0 \leq i' \leq 255$ (you can use the `uint8` function). Include your MATLAB code and the resulting contrast-boosted image in your report. Compare the image to part (b). What is the downside of increasing contrast in this way?

3 Edge Detection

An "edge" is a place where image intensity changes abruptly. Edges can indicate the borders of objects.

(a) The intensity changes associated with vertical edges can be detected by calculating:

horizontal gradient at a pixel ≈ (the pixel) - (pixel on its left)

for every pixel in the image. Open `edgedetector.m` and edit the `DetectVerticalEdges()` function to do this. Run the `edgedetector()` function to display your calculated gradients as an image. Compare the original image to the image of gradients. Verify that the vertical edges were detected and are visible as very bright or very dark areas in the gradient image. However, the gradient image also shows some tiny bright/dark spots that indicate "edges" in the water. What caused these tiny "edges?" Include your explanation and your gradient image in your report.

(b) We will now assess the effects of "blurring" the gradient image. The simplest blurring operation is the box blur. In a box blur of width n, each blurred pixel is computed as follows:

$$blur(x, y) = \frac{1}{n^2} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} img(x+i, y+j)$$

In `edgedetector.m`, edit the `BoxBlur()` function to do this, and run the `edgedetector()` function to view the results. What was the effect of the blur on the large edges, compared to the tiny "noise" edges? Why? Include your explanation and your blurred edge image in your report.

4 Transformation Matrices

Note: In Problem 3 above, you had to handle the fact that MATLAB images don't use Cartesian coordinates. You don't need to think about that detail for this problem; just use standard Cartesian coordinates.

(a) Recall that transformation matrices can be used to rotate, scale, and/or skew vectors. For example, the rotation matrix

$$A = \begin{bmatrix} \sqrt{2}/2 & -\sqrt{2}/2 \\ \sqrt{2}/2 & \sqrt{2}/2 \end{bmatrix}$$

will rotate a vector 45 degrees counterclockwise (CCW). Points may be thought of as vectors from the origin, so we could equivalently say that it will rotate a point by 45 degrees CCW about the origin. For a vector

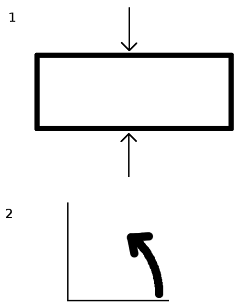
$$X_0 = \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$$

please describe (in words) the effect of the transformation

$$X_1 = AAX_0$$

In other words, how does X_1 relate to X_0? *Hint: when considering a series of matrix transformations, it is best to think of them as being applied one after the other, moving from* ***right to left****.*

(b)



Continue to use X_0 and A as defined above. Design a series of matrix multiplications which will **first** scale a vector X_0 by 1.0 in the x direction and 0.5 in the y direction, and then rotate the result by 45 degrees counterclockwise, to produce a vector which we will call X_2. Give a formula for X_2 in terms of X_0, the matrix A from above, and any other necessary matrices which you define.

(c) Use matrix multiplication to produce a **single** 2x2 matrix T which will accomplish the rotating-and-scaling transformation from (b) above. In other words, give a 2x2 matrix T such that $X_2 = TX_0$, with X_2 as defined above. You may use Matlab to multiply matrices. The MATLAB script **imageTransform.m** was provided with the homework. Open it and add your transformation matrix where the comments indicate. Run the script to see MATLAB apply your transformation matrix to every pixel in the image. Include the transformed image in your report, as well as your matrix T. *If the output is not what you expect, double-check your work from (b) and above.*

(d) *We will use 2D homogeneous coordinates* $[x, y, 1]^T$ *in this problem.* Build a transformation matrix T to translate a vector by +2 in the x direction (and not perform any rotation). Also build a rotation matrix R to rotate a vector 90 degrees CCW. Include T and R in your report. Given the 2D homogeneous vector

$$p_0 = \begin{bmatrix} 7 \\ 0 \\ 1 \end{bmatrix}$$

use MATLAB to apply the transformation $p_1 = TRp_0$ and $p_2 = RTp_0$. Give the values of p_1 and p_2 and explain why they differ from each other.

(e) Assume the unknown matrix R is a rotation matrix. *Remember that for rotation matrices, the transpose of the matrix has the reverse effect, which means* $R^T = R^{-1}$. *And this means* $R^T R = I$ *and* $RR^T = I$. Also, assume the unknown matrix A is invertible. Use these facts to solve for Y in the matrix expression

$$RYA = B. \tag{3}$$

Remember, Y and B are not rotation matrices, and you can't assume anything in particular about them.

5 SVD for Image Compression

Singular Value Decomposition (SVD) can be effectively used to compress images. Suppose I is the pixel intensity matrix of a large image $n \times n$. The transmission (or storage) of I requires $O(n^2)$ numbers. Instead, one could use I_k, that is, the top k singular values $\sigma_1, \sigma_2, \ldots, \sigma_k$ along with the left and right singular vectors $u_1, u_2, \ldots, u_k$ and $v_1, v_2, \ldots, v_k$. This would require using $O(kn)$ real numbers instead of $O(n^2)$ real numbers. If k is much smaller than n, this results in substantial savings.

In this problem, you will explore SVD compression on the `flower.bmp` image we have provided. In addition to your answers to each question, you should also submit your Matlab code and required plots where necessary. *Hint*: You may find the Matlab `svd` command particularly useful for this problem.

(a) Use MATLAB to read in `flower.bmp` and convert it to grayscale and 'double' format. Apply SVD and give the top 10 singular values. Generate a plot for all singular values versus their rankings (the `diag` command may be helpful to format the values). What do you notice from this plot?

(b) Verify that you can reconstruct and display the image using the three SVD matrices (note that the `svd` command returns V, not V^T). Then, perform compression by using only the top k singular values and their corresponding left / right singular vectors. Let k = 10, 50, and 100. Reconstruct and print the compressed images for the three different values of k. Briefly describe what you observe.

(c) Instead of transmitting the original (grayscale) image, you can perform SVD compression on it and transmit only the top k singular values and the corresponding left / right singular vectors. This should be much smaller than the original image for low values of k. With this specific image, will we still save space by compressing when $k = 200$? Show why or why not.

(d) Let e_{ij} be the per-pixel error, i.e. the difference between the original image and the compressed version at coordinates row i and column j. Explain why this expression gives the upper bound on the per-pixel error:

$$e_{ij} \leq \rho \sum_{i=k+1}^{r} \sigma_i, \tag{4}$$

where r is the rank of the original matrix (i.e., the total number of singular values), σ_i is the i*th* singular value, k is defined above, and ρ is a constant that you must fill in (in other words, ρ is just a simple constant number, which you can arrive at through understanding what is being added up by the formula, and how it relates to the worst-case error). *For full credit, you must determine ρ and explain how the formula is calculating worst-case error.*

(e) Explain how you could use the formula in (d) to choose the smallest possible value of k such that the per-pixel error at any pixel is guaranteed to be no greater than ξ, for some small tolerance $\xi > 0$.

6 SVD application - Matrix Norm

Similarly to vectors, we may define a norm (a magnitude measurement) for matrices. A matrix norm $\|\cdot\|$ is any function with the following properties:

(i) $\|A\| \geq 0$, with equality iff $A = 0$ (zero matrix).

(ii) $\|cA\| = |c|\|A\|$, for any $c \in \mathbb{R}$.

(iii) $\|A + B\| \leq \|A\| + \|B\|$.

A common matrix norm is the 2-norm:

$$\|A\|_2 = \max_x \sqrt{(Ax)^T(Ax)},$$

where x is a **unit vector** (i.e. of length 1). Here, the notation

$$\max_x(expression)$$

means "the maximum value of (*expression*), for any value of x" and the expression $\sqrt{x^T x}$ for a column vector x is another way of writing the length (you should go through the multiplication yourself to make sure you understand this fact). In other words, we can see that $\|A\|_2$ measures the maximum "magnifying power" of A: the max amount A can stretch any unit vector x under multiplication.

(a) At first glance, it looks like it would be hard to calculate $\|A\|_2$. But we can prove that $\|A\|_2 = \sigma_{\max}$, the largest singular value of A. Please prove this using the properties of the Singular Value Decomposition of A. *Hint: here is one possible strategy you might follow to prove this:*

- *Rewrite the formula for $\|A\|_2$ using the knowledge that any matrix may be expressed as its SVD: $A = U\Sigma V^T$.*
- *Use the properties of transpose and of rotation matrices to eliminate the V's.*
- *Note that, for any rotation matrix R and unit vector u, Ru produces another unit vector (call it w), and $u^T R^T$ produces w^T. You can use this line of reasoning to eliminate U (as an alternative to this reasoning and the next step, you could skip straight to defining the unit vector which will maximize the expression you have).*

- *At this point, you should have an expression with only Σ's and unit vectors. Using the properties of Σ, you should be able to easily state the unit vector that will maximize this expression, which will allow you to complete the proof.*

(b) Another commonly used norm is the *Frobenius norm*:

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2}, \quad \text{where } A \text{ is a } m \times n \text{ matrix} \tag{5}$$

If we square it to eliminate the square root, the same operation can be calculated as

$$\|A\|_F^2 = \text{trace}(A^T A)$$

From class, remember that the trace of a matrix is the sum of its diagonal elements. If desired, you can verify that the above expressions match by working through yourself how the diagonal elements of $A^T A$ are calculated. Your task is to prove that:

$$\text{trace}(A^T A) = \sigma_1^2 + \sigma_2^2 + \cdots + \sigma_r^2,$$

the sum of the squares of the singular values of A. *Hints*:

- *Use some of the SVD properties that you used in (a).*
- *One property of the trace is that it is "basis independent," which means, if we have an invertible matrix B,* $\text{trace}(B^{-1}AB) = \text{trace}(A)$.
- *Using the above properties, you should be able to reduce the expression to a matrix whose diagonal elements are $\sigma_1^2 + \sigma_2^2 + \cdots + \sigma_r^2$.*