Quiz 1

Solutions to all Questions

Ahmad Choudhry
Feb 15, 2025
CE 4TN4
Instructor: Dr. Sharma
Teaching Assistants: Da Ma & Sora Alghziwatalkhawaldh

Question 1 - Problem 2.7

Given: A flat area with center at (x_0, y_0) is illuminated by a light source with intensity

$$i(x, y) \;=\; K\,\exp\!\big[-\big((x-x_0)^2 + (y-y_0)^2\big)\big].$$

Assume the reflectance is constant and equal to 1, and let $K = 255$. When this image is digitized with k bits of intensity resolution, the number of possible intensity levels is 2^k. Since the maximum intensity is 255, the quantized increment in intensity (the difference between adjacent digital levels) is

$$\Delta G \;=\; \frac{255+1}{2^k} \;=\; \frac{256}{2^k}.$$

We are told that an abrupt change of 8 intensity levels (i.e. $\Delta G = 8$) is detectable by the eye. Hence we set

$$\Delta G \;=\; \frac{256}{2^k} \;=\; 8 \quad\Longrightarrow\quad 2^k \;=\; \frac{256}{8} \;=\; 32,$$

which gives

$$k \;=\; 5.$$

Thus, a resolution of $k = 5$ bits (or fewer) produces only $2^5 = 32$ levels, so an 8-level jump between neighboring pixels is visible, leading to false contouring.

Question 2 - Problem 2.12

Statement: Develop an algorithm for converting a one-pixel-thick 8-path to a 4-path.

An 8-path allows diagonal adjacency. To convert it into a 4-path (which only allows vertical/horizontal adjacency), we can systematically replace every diagonal step in the path by two successive 4-adjacent steps. A common way to do this is:

1. *Scan* the path from one endpoint to the other (or in any orderly manner).
2. *Identify* each pair of consecutive pixels (p_i, p_{i+1}) that are diagonally adjacent (i.e., differ by 1 in both x and y coordinates).
3. *Replace* that diagonal adjacency by two 4-adjacent links:

$$p_i \;\to\; \begin{cases} \text{(an intermediate horizontally or vertically adjacent pixel),} \\ \quad \text{then to } p_{i+1}. \end{cases}$$

4. Carefully define all possible "neighborhood" patterns (local 3×3 patterns around each diagonal) to ensure that the replacement does not introduce any unintended breaks or overlaps.

This procedure yields a valid 4-connected path of the same overall shape (still one-pixel thick), but with only horizontal and vertical moves.

Question 3 - Problem 2.15

Statement (simplified): We have the following 4×4 grid of intensity/label values (shown here with row/column layout), and two special points p and q:

$$\begin{array}{cccc} 3 & 1 & 2 & 1\,(q) \\ 2 & 2 & 0 & 2 \\ 1 & 2 & 1 & 1 \\ (p)\,1 & 0 & 1 & 2 \end{array}$$

(a) Let $V = \{0, 1\}$. Compute the lengths of the shortest 4-, 8-, and m-paths between p and q. If no path exists for a given adjacency, explain why.
(b) Repeat for $V = \{1, 2\}$.

Solution outline:

(a) Case $V = \{0, 1\}$.

- *4-path:* In order for a 4-path of pixels to exist from p to q, every pixel on that path must (i) be 4-adjacent to its neighbors, and (ii) have a value in $\{0, 1\}$. By inspection of the grid, one finds that it is impossible to move from p to q *only* through $\{0, 1\}$-valued pixels with purely up/down/left/right steps. Hence, **no 4-path exists** in this case.

- *8-path:* If we allow diagonal steps, we *can* find a path consisting entirely of 0's and 1's. One such path has length 4. (See the book/diagram for a specific example of diagonal connections.) This is the **shortest 8-path**, and it is unique.

- *m-path:* An m-path is a path that allows 8-adjacency except that we must avoid the "diagonal bridging" of two 4-adjacent pixels with the same value if that bridging crosses a pixel with a different value. Even so, a valid m-path with length 5 can be found. It is unique in this case.

(b) Case $V = \{1, 2\}$.

- *4-path:* Now we look for a path of 4-adjacent pixels all belonging to $\{1, 2\}$. It turns out that such a path *does* exist, and its *shortest length* is 6. However, there is *more than one* such 4-path of length 6, so the shortest 4-path is not unique.

- *8-path:* Allowing diagonal connections among pixels in $\{1, 2\}$ yields shorter paths. A shortest 8-path is found with length 4. Again, that path is not unique: there are multiple diagonal ways to get from p to q in 4 steps.

- *m-path:* A shortest m-path in this set also has length 6, and it is *not* unique. The presence of diagonal bridging rules and multiple possible routes means there can be more than one minimal m-path.

Question 4 - Problem 2.16

(a) Give the condition(s) under which the D_4 distance between two points p and q is *equal* to the length of the shortest 4-path between these points. **(b)** Is this path unique?

Recall that the D_4 distance (sometimes called the city-block distance) between points

$$p = (x_1, y_1), \quad q = (x_2, y_2)$$

is defined by

$$D_4(p, q) \;=\; |x_1 - x_2| \;+\; |y_1 - y_2|.$$

A 4-path between p and q can move only in the horizontal or vertical directions. If there are *no obstacles* (i.e. we are free to step through every pixel on a simple rectangular route) and every intermediate pixel on that route is valid for the path, then a straightforward route of length $D_4(p, q)$ exists: simply move $\Delta x = |x_1 - x_2|$ steps in one horizontal direction and $\Delta y = |y_1 - y_2|$ steps in one vertical direction (in any order).

Hence, **the condition** under which $D_4(p, q)$ *equals* the length of the shortest 4-path is simply that the path is *unobstructed* and can proceed purely by horizontal/vertical steps from p to q without having to detour. In that case, the shortest 4-path is indeed $D_4(p, q)$ in length.

Uniqueness? In general, *the path is not unique*. Even with no obstacles, there are multiple ways to order the horizontal and vertical steps. For example, to move 3 steps up and 4 steps right, one can choose any arrangement of those moves (e.g. up-up-up-right-right-right-right, or up-right-up-right-up-right-right, etc.) and all have the same total length $3+4 = 7$. Therefore, in most typical grids, there are multiple shortest 4-paths. But in theory, there can be instances when the path is unique dependent on the values along the steps.

Question 5: Resolution and Gray-Level Reduction

Reducing Resolution by Factors of 3 and 5

Reducing the *spatial resolution* of an image by an integer factor F involves sampling every F-th row and column. In practical terms, if the original image is of size $M \times N$, the down-sampled image will be approximately of size $\frac{M}{F} \times \frac{N}{F}$.

- **Nearest-Neighbor Method:** We create a new image whose pixel at (r, c) comes from the original at ($F \times r$, $F \times c$). This is the simplest approach and is typically done by "slicing" the row/column indices in code.

- **Implementation Note:** See the Python function `reduce_resolution(img, factor)` given in the submission. For instance, calling `reduce_resolution(lena, 3)` keeps only every 3rd row and column of the `lena` image, thus reducing resolution by a factor of 3.

Reducing Gray Levels by Factors of 2, 4, and 8

To reduce the *number of gray levels* from 256 to $256/F$ (for example, $F = 2, 4, 8$), we quantize each pixel into larger "bins."

- **Quantization Strategy:** One simple approach is

$$\text{new\_value} \;=\; \left\lfloor \frac{\text{old\_value}}{F} \right\rfloor \times F.$$

  This shrinks the number of distinct levels and slightly coarsens the image's shading.

- **Implementation Note:** The Python function `reduce_gray_levels(img, factor)` demonstrates this. For instance, calling `reduce_gray_levels(cameraman, 4)` will group pixel intensities in steps of 4, producing 64 possible levels (since $256/4 = 64$).

Example Results with "Lena" and "Cameraman"

We test these functions using:

- Factors of 3 and 5 for spatial resolution reduction.
- Factors of 2, 4, and 8 for gray-level reduction.

These are all included in the submission.

Question 6: Manual Rotation by 30° Using a Transformation Matrix

Rather than using a built-in library function, we implement rotation *manually* by applying the standard 2D rotation matrix from the table 2.2 in the textbook. For a counter-clockwise angle θ, the forward transform is

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v \\ w \\ 1 \end{pmatrix},$$

i.e.,

$$x = v\cos\theta \;-\; w\sin\theta, \quad y = v\sin\theta \;+\; w\cos\theta.$$

Here (v, w) are the *input* coordinates and (x, y) are the *rotated* output coordinates. To rotate about the image center (c_x, c_y) rather than $(0, 0)$, we first *shift* (v, w) so the center is at $(0, 0)$, apply the rotation, then shift back.

Algorithm Outline

1. **Read the input** (e.g., `Lena.tif`) and store in an array.
2. **Compute the center** (c_x, c_y) of the image, typically $\left(\frac{w}{2}, \frac{h}{2}\right)$.
3. **Allocate an output array** of the same or larger size.
4. **Inverse mapping:** For each pixel $(x_{\text{out}}, y_{\text{out}})$ in the *output* array, compute which (v, w) in the *input* array would map to it under the rotation. The inverse rotation for CCW is
$$v = x\cos\theta + y\sin\theta, \quad w = -x\sin\theta + y\cos\theta.$$
Add back (c_x, c_y) to account for the center shift.
5. **Nearest-neighbor sampling:** Round (v, w) to integer coordinates. If they lie within the input image bounds, copy that pixel into the output. Otherwise, set the output to a default (e.g. black).

Result

Running this procedure at $\theta = 30°$ on the Lena image yields a rotated version without relying on functions like `rotate` from PIL. The Python code implements this transformation directly from the matrix in the attached table.

Question 7: Finding Connected Pixels with Threshold T

We want all pixels "connected" to a given pixel p if:

1. They are within a chosen adjacency (often 4-adjacency).
2. The absolute difference in gray-level between a pixel and its neighbor is *less than or equal to* T.

The pseudo-code given is effectively a **flood fill** or **graph traversal** (BFS or DFS).

Pseudo-code Explanation

- **Initialize:** All labels $l(s)$ are set to 0, meaning unvisited (or not connected). Let $B = \{p\}$, where p is the chosen *seed* pixel.
- **Process Loop:** While B is not empty:
    1. Remove any element q from B.
    2. Label $l(q) = 1$.

3. Find neighbors r of q (in the 4-adjacent sense) whose label is currently 0 *and* whose intensity differs from q by no more than T.
4. Add those neighbors to B.

- **Result:** When the loop ends, all pixels marked with $l(r) = 1$ form the connected set (the region around p). We can then write the final labeled image with, for example, 1 mapped to white and 0 to black.

Practical Example

When running on an image called "bird.png" with threshold $T = 2$,

- We supply the seed pixel, say $(67, 45)$.
- All pixels that can be reached by 4-adjacency steps, such that each step never jumps more than 2 in gray-level difference from one pixel to the next, are labeled 1.
- All other pixels remain at 0.

Repeating this for $T = 3$ or $T = 4$ expands (or contracts) the connected region.

Implementation Note: In Python, one can use a queue (BFS) or recursion (DFS). Our sample function `connected_label` shows a BFS with a queue. It checks each neighbor's intensity difference and enqueues those within threshold. The final output is a label map of 0 and 1.

Question 8 Problem 3.2

We are given exponentials of the form $e^{-\alpha r^2}$ (with $\alpha > 0$) as the basis for constructing smooth intensity transformations. The goal is to devise functions $s = T(r)$ having the general shapes in Figs. 3.2(a)–(c), with parameters chosen so that each curve meets the "half-value" condition indicated in the figure. In all cases, r denotes the input intensity and s the output intensity.

(a) Shape as in Fig. 3.2(a)

A suitable model is

$$s \;=\; T(r) \;=\; A\, e^{-\,K\, r^2}.$$

According to Fig. 3.2(a), at $r = L_0$ the output value is $\frac{A}{2}$. Thus,

$$A\, e^{-\,K\, L_0^2} \;=\; \frac{A}{2},$$

which implies

$$e^{-\,K\, L_0^2} \;=\; \frac{1}{2} \quad\Longrightarrow\quad -\,K\, L_0^2 \;=\; \ln\!\left(\tfrac{1}{2}\right) \;=\; -\,\ln(2).$$

Hence

$$K \;=\; \frac{\ln(2)}{L_0^2} \;\approx\; \frac{0.693}{L_0^2}.$$

So the final form is

$$\boxed{s \;=\; A\,\exp\!\Big(-\frac{\ln(2)}{L_0^2}\,r^2\Big).}$$

(b) Shape as in Fig. 3.2(b)

Here the curve starts near 0 when $r = 0$ and approaches B asymptotically for large r. A convenient model is

$$s \;=\; T(r) \;=\; B\big(1 - e^{-K\,r^2}\big).$$

From the figure, at $r = L_0$ we have $s = \frac{B}{2}$. Therefore,

$$B\Big(1 - e^{-K\,L_0^2}\Big) \;=\; \frac{B}{2} \quad\Longrightarrow\quad 1 - e^{-K\,L_0^2} \;=\; \frac{1}{2} \quad\Longrightarrow\quad e^{-K\,L_0^2} \;=\; \frac{1}{2}.$$

Exactly as before, we get $K = \frac{\ln(2)}{L_0^2}$, so

$$\boxed{s \;=\; B\Big(1 - \exp\big[-\tfrac{\ln(2)}{L_0^2}\,r^2\big]\Big).}$$

(c) Shape as in Fig. 3.2(c)

A more general version of the previous model (allowing a nonzero lower asymptote C and upper asymptote D) is

$$s \;=\; T(r) \;=\; \big(D - C\big)\big[1 - e^{-K\,r^2}\big] \;+\; C,$$

which starts near C when $r = 0$ and asymptotically approaches D for large r. If a "halfway" constraint is given (e.g., $s = \frac{C+D}{2}$ at some $r = L_0$), it again leads to $e^{-K\,L_0^2} = \frac{1}{2}$ and thus $K = \frac{\ln(2)}{L_0^2}$.

Question 9 - Problem 3.3

We now want a *continuous* function for implementing the contrast-stretching transformation of Fig. 3.2(a), but normalized so that s ranges from 0 to 1 for r in the valid input range. In addition to a midpoint parameter m, we include a slope parameter E that controls how sharply the function transitions from high to low.

(a) A Suitable Continuous Function

A convenient choice is a (decreasing) logistic or exponential-style function that goes from $s = 1$ at $r = 0$ down to $s = 0$ at $r = L$ (the maximum intensity). One common *sigmoid* variant that fits the shape of Fig. 3.2(a) can be written:

$$s \;=\; T(r) \;=\; \frac{1}{1 + \exp\Big[\, E\,(r - m)\Big]}\,.$$

By adjusting $E > 0$, we control the slope: a larger E makes the curve steeper near $r = m$. Also, $r = m$ is where $s = \frac{1}{2}$. In typical usage:

- $r \in [0, L]$, for an L-level image (e.g. $L = 255$ for 8-bit).
- m is placed somewhere in $(0, L)$ to control the main "drop" from near 1 to near 0.

This is just one common formula that satisfies the requirement that the output be in $[\,0, 1\,]$ and that there be a parameter E for slope control. Other decreasing exponentials (such as $e^{-E\,r^2}$ suitably normalized) could also be used, provided s is in the range $[0, 1]$ and $s(m) = \frac{1}{2}$.

(b) Sketching a Family of Transformations as E Varies

For a fixed midpoint $m = \frac{L}{2}$, plotting

$$s \;=\; \frac{1}{1 + \exp\Big[E\,(r - L/2)\Big]}$$

for $E = 1, 2, 5, 10, \ldots$ yields a *family* of S-shaped curves that all pass through $r = \frac{L}{2},\ s = \frac{1}{2}$, but become more and more abrupt as E increases. Conversely, smaller E makes the transition more gentle.

(c) The Smallest Value of E for a "Binary" Result

In the context of an 8-bit image ($L = 255$), if $m = 128$ and we let $E \to \infty$ then

$$\exp\Big[E\,(r - 128)\Big]$$

is either extremely large or extremely small except right at $r \approx 128$. Numerically, once E is large enough so that for $r < 128$ we have $s \approx 1$ (to within machine precision), and for $r > 128$ we have $s \approx 0$, the function behaves like a *binary threshold* at $r = 128$.

The question asks: "What is the smallest E that makes the function effectively produce a binary image?" In floating-point arithmetic, we need s to round to 1 for $r < 128$ and to 0 for $r > 128$, up to the smallest positive number C that the machine can represent above 0. Roughly speaking, we want

$$\exp\big[E(128 - r)\big] \;\geq\; \frac{1}{C} \quad \text{for} \quad r < 128,$$

and

$$\exp\big[E(r-128)\big] \;\geq\; \frac{1}{C} \quad \text{for} \quad r > 128.$$

Solving these precisely can be done by taking logarithms and substituting the IEEE floating-point C on a given system. In practice, once E is large enough that the exponential saturates to 0 or 1 in the software, the output looks binary. Thus the minimal E depends on C (the machine epsilon or smallest representable positive above zero).

Problem 10: Piecewise Linear Contrast Stretching

We have a contrast-stretching transform that remains at 0 for input intensities below T_1, increases linearly from 0 to 255 for input intensities between T_1 and T_2, and remains at 255 for intensities above T_2. Mathematically:

$$Y = \text{stretch}(X, T_1, T_2) = \begin{cases} 0, & X < T_1, \\ 255\,\dfrac{X - T_1}{T_2 - T_1}, & T_1 \leq X \leq T_2, \\ 255, & X > T_2. \end{cases}$$

Here X denotes the input gray level (assumed in $[0, 255]$), and Y denotes the output gray level (also constrained to $[0, 255]$). The parameters T_1 and T_2 ($0 \leq T_1 < T_2 \leq 255$) control the location and slope of the "stretch" region.

Explanation

- **Behavior:** Intensities below T_1 are "pushed down" to black (0), while intensities above T_2 are "pushed up" to white (255). The range in between $[T_1, T_2]$ is linearly mapped to $[0, 255]$.
- **Usage:** By changing T_1 and T_2, we can adjust how much of the input range we are "stretching" or saturating. For instance, a narrower range between T_1 and T_2 means more aggressive contrast stretching for that segment.
- **Implementation:** The provided MATLAB function `stretch()` performs exactly this piecewise mapping. One simply calls

  ```
  Y = stretch(I, T1, T2);
  ```

  for an 8-bit grayscale image `I`.

Experiments

- In the code, we demonstrate running `stretch` on `einstein.tif` and `race.tif` using multiple (T_1, T_2) pairs. The resulting images will show different levels of contrast enhancement in specific ranges of the intensity scale.

Problem 11: Spatial Filtering on the "race" Image

We next apply several standard spatial filters to `race.tif`:

1. **Weighted Averaging (Smoothing):** Uses a 3×3 filter

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

   to blur the image slightly and reduce noise.

2. **Composite Laplacian Filter:** We convolve with

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix},$$

   which highlights edges. Subtracting some fraction of this result from the original image sharpens edges and fine detail.

3. **Sobel Operators:** We convolve with

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix},$$

   to detect horizontal and vertical edges. We may also form the gradient magnitude,

$$\sqrt{(f * S_x)^2 + (f * S_y)^2},$$

   to produce a single image showing overall edge strength.

Explanation and Observations

- **Smoothing** generally blurs the image, lowering high-frequency details.
- **Laplacian-based Sharpening** can recover some high-frequency detail, making edges crisper. One must choose a scaling factor (e.g. $\lambda = 1$) for how strongly to add or subtract the Laplacian from the original.
- **Sobel Edge Detection** reveals edges by approximating intensity gradients. The x-direction kernel S_x highlights vertical edges, while S_y captures horizontal edges. Combining them into a gradient magnitude displays the overall edge map.

Implementation Details:

- We load the original `race.tif` image in MATLAB, convert it to `double` to avoid integer overflow, and apply each filter via `imfilter`.

- For the Laplacian filter, we typically use `replicate` at the borders (which replicates edge pixels outward).
- For the gradient magnitude, we normalize the resulting values to the $[0, 1]$ range before writing to disk.