Digital Image Processing CS14
Final Exercises

General Instructions:

1. Video Requirements:
   - Prepare a short video file (5–15 seconds) in a common format such as MP4 or AVI.
   - Ensure the file is ready for processing.

2. Programming Setup:
   - Use a suitable programming environment, such as Python with OpenCV or MATLAB.
   - Ensure all necessary libraries and dependencies are installed before starting.

3. Project Setup:
   - Main Project Folder: Create a dedicated main folder on your computer to hold all the files related to these exercises (e.g., **DIP Video Exercises**).
   - Exercise Submission Folder: For *each* exercise you complete (**e.g., Exercise 1**), create a specific sub-folder inside your main project folder. Name this sub-folder exactly as **YourFullName-FinalExerN**, replacing **YourFullName** with your actual full name and **N** with the exercise number.
     a. Example for Exercise 1: **JuanDelaCruz-FinalExer1**
     b. Example for Exercise 2: **JuanDelaCruz-FinalExer2**
   - Place the corresponding Python script and the final output video for *that specific exercise* inside this folder.

4. Environment & Libraries:
   - Programming Setup: You need a functional programming environment. Python with OpenCV and NumPy is highly recommended and used in the provided code examples.
     - Ensure you have Python installed.
     - Install required libraries:
       a. OpenCV: **pip install opencv-python**
       b. NumPy: **pip install numpy**
   - System Capabilities: Your environment must support:
     - Reading video files frame-by-frame (common formats like MP4, AVI).
     - Accessing video properties: Frame Rate (FPS), Resolution (Width, Height), and Frame Count. (Be aware that reported frame counts are not always 100% accurate).
     - Performing image processing operations on individual video frames (pixel manipulations, filtering, transformations).
     - Writing processed frames back into a new output video file, typically preserving the original FPS and resolution unless the exercise specifies otherwise.

5. Input Video:
   - *Video Requirement:* You must use a video file as input for these exercises. Image sequences require code modification.
   - *Source File:* Prepare a short input video clip (5-15 seconds is usually sufficient for testing). Use common formats like **.mp4** or **.avi**.
   - *Placement:* Copy your input video file (e.g., my_test_video.mp4) into your main project folder or directly into the specific **YourFullName-FinalExerN** folder for the exercise you are working on. This makes it easier for the script to find.

6. Coding & Execution:
   - *Script Naming:* Save the Python code for each exercise as a separate **.py** file. Use the naming convention **YourFullName_ExerN.py.**

    a. Example for Exercise 1 code: **JuanDelaCruz_Exer1.py**
    b. Example for Exercise 2 code: **JuanDelaCruz_Exer2.py**
- *Customize Code:* Crucially, open each script and modify the placeholder filenames.
    - Change input_filename = "input_video.mp4" to match the actual name of YOUR input video file (e.g., **input_filename = "my_test_video.mp4"**).
    - Change output_filename = "output_exercise_N.mp4" to the desired name for your output video (e.g., **output_filename = "transformed_video_exer1.mp4"**). Make sure the output name is distinct for each exercise.
- Run: Execute the script from your terminal or IDE, ensuring you are in the correct directory (usually the folder containing the script and input video). Example command: **python JuanDelaCruz_Exer1.py**

7. Output:
    - *Video Transformation Output:* Each exercise will generate a new video file showcasing the applied transformation.
    - *Location & Naming:* The output video will be saved in the location specified by the **output_filename** variable (by default, the same directory as the script). Ensure this output video file is placed inside the correct submission folder (e.g., the generated **transformed_video_exer1.mp4** should be inside the **JuanDelaCruz-FinalExer1** folder).
    - *Format:* The default output is **.mp4** using the **'mp4v'** codec. If you face issues saving or playing the output, consider changing the **fourcc** code in the script (e.g., to **'XVID'**) and the output filename extension (e.g., to **.avi**). Codec support varies by OS.
    - *Verify:* Always open and play the generated output video to confirm the transformation was applied correctly and the video is not corrupted.

Remember to handle potential errors like file not found, incorrect video formats, and ensure you release the video capture and writer objects properly to avoid corrupted output files. Good luck!

Exercise 1: Temporal Grayscale Conversion & Contrast Adjustment

Objective: Convert a color video to grayscale, and dynamically adjust the contrast over the duration of the video (e.g., starting with low contrast and ending with high contrast).
Input: A short color video clip.

Transformation Steps:
1. Open the input video file and prepare an output video writer. Get video properties (FPS, width, height, total frames).
2. Loop through each frame of the input video.
3. For each frame: a. Convert the color frame to grayscale.
    a. (Common method: weighted average Y = 0.299*R + 0.587*G + 0.114*B).
    b. Calculate the current frame's position relative to the total duration (e.g., progress = current_frame_index / total_frames).
    c. Determine the contrast adjustment factor based on progress. For example, linearly interpolate a contrast multiplier alpha from 0.8 (low contrast) to 1.5 (high contrast): alpha = 0.8 + (1.5 - 0.8) * progress.
    d. Apply contrast adjustment. A simple method is new_pixel = alpha * (pixel - mean) + mean, where mean is the average pixel value of the grayscale frame (often 128 for 8-bit images, or calculate it). Clamp pixel values to the valid range [0, 255].
4. Write the processed (grayscale, contrast-adjusted) frame to the output video file.
5. Release video reader and writer resources after processing all frames.

Key Concepts: Video I/O, Frame Iteration, Color-to-Grayscale Conversion, Contrast Enhancement (Linear Stretching), Parameter Interpolation over Time.
Tools/Libraries: cv2.VideoCapture, cv2.VideoWriter, cv2.cvtColor (with cv2.COLOR_BGR2GRAY), basic NumPy/array math for contrast adjustment.

Output: A grayscale video where the contrast smoothly increases from the beginning to the end.
Challenge: Instead of linear interpolation, make the contrast pulsate (e.g., follow a sine wave pattern over time).

Answer the following Questions:

1. **Perceptual Impact of Grayscale:** The standard weighted average (0.299*R + 0.587*G + 0.114*B) approximates human luminance perception. How would using a simple average ((R+G+B)/3) or using only the Green channel alter the perceived details and structure in the video? In what types of scenes might these differences be most pronounced?

2. **Global vs. Local Contrast:** The linear contrast adjustment (cv2.convertScaleAbs or alpha*(pixel-mean)+mean) modifies contrast globally across the entire frame. How might this fail in scenes with both very bright and very dark regions? Discuss the conceptual approach and potential benefits/drawbacks of implementing a *temporally varying local* contrast enhancement (like adaptive histogram equalization - CLAHE) instead.

3. **Temporal Coherence:** Linearly interpolating the contrast factor alpha ensures smooth change. What visual artifacts (like flickering) might occur if the contrast adjustment was based on a per-frame metric (e.g., standard deviation of the current frame) without temporal smoothing? How could you design a system that adapts contrast based on content but maintains temporal stability?

4. **Information Loss:** Converting to grayscale discards color information. While adjusting contrast can enhance detail, in what ways does the combined process potentially lead to an overall loss of *distinguishable* information compared to the original color video, even if the grayscale contrast looks subjectively "better"?

5. **Computational Trade-offs:** Calculating the true mean pixel value for the contrast formula (beta = mean * (1 - alpha)) requires an extra pass over the frame data compared to using a fixed approximation like 128. Analyze the scenarios (e.g., video resolution, content variability, target platform) where the computational cost of calculating the exact mean might be justified or prohibitive.

Exercise 2: Applying a Moving Filter (e.g., Box Blur)

Objective: Apply a constant-size filter (like a box blur or Gaussian blur) to the video, but have the *effect* appear to move across the frame horizontally over time.
Input: A color or grayscale video clip.

Transformation Steps:

1. Open input video, prepare output writer, get properties.
2. Define the filter kernel size (e.g., kernel_size = 21 for a 21x21 blur).
3. Determine the width of the video frame (frame_width).
4. Loop through each frame (i) of the input video.
5. For each frame:
   a. Calculate the horizontal center position (cx) of where the *full* blur effect should be applied. This position should move across the screen. Example: cx = (frame_width / total_frames) * i.
   b. Create a copy of the original frame (output_frame).
   c. Apply the chosen filter (e.g., cv2.blur or cv2.GaussianBlur with the defined kernel_size) to the *entire* original frame (blurred_frame).
   d. Define a region around cx where the blurred version will be shown. Let's say the region width is blur_region_width (e.g., 100 pixels). Calculate the start (x_start) and end (x_end) columns for this region, ensuring they stay within frame bounds: x_start = max(0, int(cx - blur_region_width / 2)) x_end = min(frame_width, int(cx + blur_region_width / 2))

   e. Copy the pixels from blurred_frame within the column range [x_start, x_end) into the output_frame at the same location. The rest of output_frame remains the original frame's content.
2. Write the output_frame to the output video file.
3. Release resources.

Key Concepts: Video I/O, Frame Iteration, Image Filtering (Blurring), Masking/Region of Interest (ROI) processing, Spatial Manipulation over Time.
Tools/Libraries: cv2.VideoCapture, cv2.VideoWriter, cv2.blur or cv2.GaussianBlur, NumPy slicing for region copying.
Output: A video where a vertical band of blur sweeps horizontally across the frame from one side to the other.
Challenge: Implement a smooth transition (alpha blending) at the edges of the blurred region instead of a hard cut.

Answer the following Questions:

1. **Filter Choice and Motion:** Compare the visual effect of using a Box Blur versus a Gaussian Blur for the moving filter region. How does the filter's kernel shape interact with motion *within* the video content as objects pass through the blurred band? Which filter might produce more visually plausible motion blur artifacts?

2. **Boundary Artifacts and Blending:** The code creates a sharp boundary for the blurred region. How does this abrupt transition affect visual perception, especially with fast motion across the boundary? Detail the steps and calculations needed to implement alpha blending for a smooth transition – what information is needed for the blend weights?

3. **Perception of Movement:** How do the parameters (kernel_size, blur_region_width, speed of traversal across the frame) interact to influence the viewer's perception of the *effect* itself? Could manipulating these parameters create illusions of speed, focus shifts, or depth changes beyond just a simple moving blur?

4. **Masking Efficiency:** The current method applies the blur to the whole frame, then copies a region. Consider alternative approaches: could you define the ROI *first* and apply the blur only within that region? Or could you use image masks directly with blending operations? Analyze the potential computational efficiency and memory usage differences between these methods.

5. **Semantic Meaning:** In what practical or artistic scenarios could a spatially localized, moving blur effect be used meaningfully? How does this differ from a global blur or a blur that tracks a specific object? Discuss how the *artificiality* of the moving band affects its potential applications.

Exercise 3: Geometric Transformation - Gradual Rotation

Objective: Rotate the video content gradually around its center, from 0 degrees at the start to a specified angle (e.g., 90 degrees or 360 degrees) by the end. Handle the changing frame boundaries.
Input: A color or grayscale video clip.

Transformation Steps:

1. Open input video, prepare output writer, get properties (width `W`, height `H`, total frames `N`).
2. Define the final rotation angle (`final_angle`, e.g., 90 degrees).
3. Determine the center of rotation (usually `(W/2, H/2)`).
4. Loop through each frame (`i`) of the input video.
5. For each frame:
   a. Calculate the rotation angle for the current frame: `current_angle = final_angle * (i / (N - 1))` if N > 1.

   b. Construct the 2D rotation matrix using the **`current_angle`** and the center of rotation. (Libraries like OpenCV have functions for this: **`cv2.getRotationMatrix2D`**).
   c. Apply the rotation to the frame using an affine warp function (**`cv2.warpAffine`**). You need to specify the output size (`(W, H)`). This might clip corners or introduce black areas.
6. Write the rotated frame to the output video file.
7. Release resources.

Key Concepts: Video I/O, Frame Iteration, Geometric Transformations (Rotation), Affine Warping, Transformation Matrices.
Tools/Libraries: `cv2.VideoCapture`, `cv2.VideoWriter`, `cv2.getRotationMatrix2D`, `cv2.warpAffine`.
Output: A video where the content appears to rotate steadily around the center point. Black areas may appear in the corners as the original rectangular frame rotates within the output frame boundary.
Challenge: Modify the process to scale the frame *down* slightly as it rotates, ensuring no part of the original frame content is ever clipped by the output boundaries. This involves calculating the bounding box of the rotated rectangle.

Answer the following Questions:

1. **Interpolation Artifacts:** cv2.warpAffine uses interpolation (e.g., linear, cubic, nearest neighbor). Analyze the types of visual artifacts each method might introduce during rotation, particularly on sharp edges, fine textures, and areas of high contrast. How does the *speed* of rotation influence the visibility of these artifacts?

2. **Handling Induced Boundaries:** Rotation introduces empty areas. Filling with black is simple but visually stark. Compare the visual and computational implications of alternative boundary handling methods like reflection padding, edge pixel replication, or attempting a (computationally expensive) content-aware fill. Which methods preserve information best versus creating potentially distracting patterns?

3. **Accumulated Error:** When performing gradual rotation frame-by-frame based on the *original* frame each time, precision errors might be minimal. If, hypothetically, each frame were rotated based on the *previously rotated* frame, how would interpolation errors accumulate over time? What visual degradation would you expect to see?

4. **Beyond Affine:** Rotation is an affine transformation. How would the process differ if you wanted to apply a non-affine transformation, like a perspective warp (e.g., simulating looking at the video tilted away from the viewer) that changes gradually over time? What transformation matrix and warping function would be needed?

5. **Center of Transformation:** The code rotates around the frame center. How would choosing an off-center point of rotation change the required size of the output frame if you wanted to ensure *no* part of the original video content is ever clipped? How would you calculate this required output size based on the rotation angle and center?

Exercise 4: Simulating a Simple "Night Vision" Effect

Objective: Transform a color video to simulate a basic night vision goggle effect: convert to grayscale, apply a green tint, possibly increase brightness/contrast slightly, and add some noise.
Input: A color video clip (preferably one shot during the day or in good lighting for a clear transformation).

Transformation Steps:
1. Open input video, prepare output writer, get properties.
2. Loop through each frame of the input video.
3. For each frame:
   a. Convert the color frame to grayscale (**`gray_frame`**).

b. Create a new 3-channel (color) frame (`output_frame`) of the same size, initialized to black.
c. Copy the `gray_frame` data into the Green channel of `output_frame`. The Red and Blue channels remain zero (or very low intensity). (e.g., in OpenCV BGR format, `output_frame[:, :, 1] = gray_frame`).
d. *Optional:* Apply a slight brightness/contrast boost to the green channel before or after placing it in `output_frame`. (e.g., `green_channel = cv2.addWeighted(gray_frame, 1.2, np.zeros_like(gray_frame), 0, 10)` adjusts contrast and brightness).
e. *Optional:* Add synthetic noise (e.g., Gaussian noise) to the green channel to simulate sensor noise. Create a noise matrix of the same size as the frame and add it. Ensure values stay within [0, 255].

4. Write the `output_frame` (now appearing green and potentially noisy) to the output video file.
5. Release resources.

Key Concepts: Video I/O, Frame Iteration, Grayscale Conversion, Color Channel Manipulation, Brightness/Contrast Adjustment, Noise Addition.
Tools/Libraries: `cv2.VideoCapture`, `cv2.VideoWriter`, `cv2.cvtColor`, NumPy for channel manipulation and noise generation (`np.random.normal`).
Output: A video that looks predominantly green, simulating a monochrome night vision display, potentially with added noise.
Challenge: Add scan lines (thin horizontal dark lines) periodically to enhance the effect.

Answer the following Questions:

1. **Simulating Sensor Physics:** Real night vision involves photon amplification and specific sensor noise characteristics, often dependent on light levels. How does the simple approach (grayscale, tint, uniform noise) fail to capture the non-linear amplification and light-dependent noise found in actual devices? Propose refinements to the noise model and brightness/contrast adjustments to improve physical realism.

2. **Color Channel Information:** The effect discards most color information, mapping luminance to green intensity. Could specific color information from the original frame be subtly reintroduced (e.g., mapping very bright red lights to a slightly different shade of green or white) to add more detail without breaking the night vision aesthetic? What are the risks of making it look unrealistic?

3. **Temporal Artifacts:** This simulation processes frames independently. Image intensifiers often exhibit lag or persistence (ghosting). How could you simulate this temporal effect by incorporating information from the *previous* processed frame(s) into the current frame's calculation? What parameters would control the amount of lag?

4. **Beyond Core Effect:** Authentic night vision often includes artifacts like geometric distortion (barrel/pincushion), blooming around bright lights, or scintillation (sparkling). Choose one of these artifacts and outline a computational approach to simulate it and integrate it into the existing effect pipeline.

5. **Perceptual Cues:** The green tint and noise are strong cues for "night vision." How much can you deviate from this specific formula (e.g., different tint color, different noise type/level, adding scan lines) before the effect is no longer readily interpreted as night vision by a viewer? Where is the boundary between simulation and abstract stylization?

Exercise 5: Creating a Static Vignette Effect

Objective: Apply a vignette effect to the video, darkening the corners and edges while keeping the center brighter. The vignette mask itself remains static throughout the video.
Input: A color or grayscale video clip.

Transformation Steps:
1. Open input video, prepare output writer, get properties (width `W`, height `H`).

2. Pre-computation (before frame loop):
    a. Create a vignette mask. This mask should be the same size as the video frames (**H x W**) and contain values typically between 0.0 and 1.0. a. Create 2D arrays representing the normalized X and Y coordinates relative to the center (ranging from -1 to 1 or similar).
    b. Calculate the distance of each pixel from the center. A common approach is radial distance **`d = sqrt(((x - W/2)/(W/2))^2 + ((y - H/2)/(H/2))^2)`.**
    c. Create the mask based on distance, often using a Gaussian falloff or a simpler polynomial. Example: `mask = exp(-d**2 / (2 * sigma**2))`, where `sigma` controls the vignette size (e.g., `sigma = 0.4`). Ensure mask values are clamped to [0, 1].
    d. If processing a color video, the mask should be applied to all color channels, so you might need to replicate the 2D mask into a 3D mask (H x W x 3).
3. Loop through each frame of the input video.
4. For each frame:
    a. Convert the frame's data type to float (if it isn't already) to allow multiplication by the mask values.
    b. Multiply the frame pixel-wise by the pre-computed vignette mask. **`output_frame = frame * mask`.**
    c. Convert the resulting frame back to the original data type (e.g., 8-bit unsigned integer), ensuring values are clipped correctly to [0, 255].
5. Write the **`output_frame`** to the output video file.
6. Release resources.

Key Concepts: Video I/O, Frame Iteration, Image Masking, Generating Synthetic Masks (Gaussian, Radial), Pixel-wise Operations, Data Type Conversion.

Tools/Libraries: `cv2.VideoCapture`, `cv2.VideoWriter`, NumPy for mask generation and element-wise multiplication.

Output: A video where the edges and corners are gradually darkened, drawing focus towards the center.

Challenge: Make the vignette intensity pulsate slightly over time (i.e., modify the `sigma` or the overall strength of the mask multiplication in the frame loop).

Answer the following Questions:

1. **Mask Profile Impact:** Compare the visual aesthetics produced by different vignette mask profiles: Gaussian, linear ramp from edge to center, cosine-squared, or a sharp circular cutoff. How does the steepness and shape of the falloff curve influence the perceived "naturalness" versus "stylization" of the effect?

2. **Color and Vignetting:** Applying the same luminance mask to all color channels (BGR) can sometimes cause perceived color shifts near the vignette edge, as channels might be affected differently relative to their original values. Analyze why this might happen. How could you design a vignette that aims for a purely luminance reduction with minimal color shift, potentially by operating in a different color space (like HSL or Lab)?

3. **Interaction with Content:** How does a static vignette interact with dynamic video content, particularly camera pans or zooms? Could a static vignette inadvertently obscure important details that move towards the edge, or create distracting brightness changes on static objects during a camera move?

4. **Quantization and Banding:** Multiplying by a smooth mask (float) and converting back to 8-bit integer can lead to banding artifacts in the smooth gradient areas of the vignette. Explain why this occurs. How could techniques like adding a small amount of noise (dithering) *before* quantization help break up these bands? What are the trade-offs of adding such noise?

5. **Dynamic Adaptation:** While the exercise specifies a *static* vignette, how could you make it subtly dynamic based on frame content? For instance, how might you automatically adjust the vignette's sigma (size) or intensity based on the overall brightness or complexity of the central region of the frame to enhance focus adaptively? What metrics could drive this adaptation?