CS510 Computer Imaging PS5 Light Fields

March 12, 2025

[1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

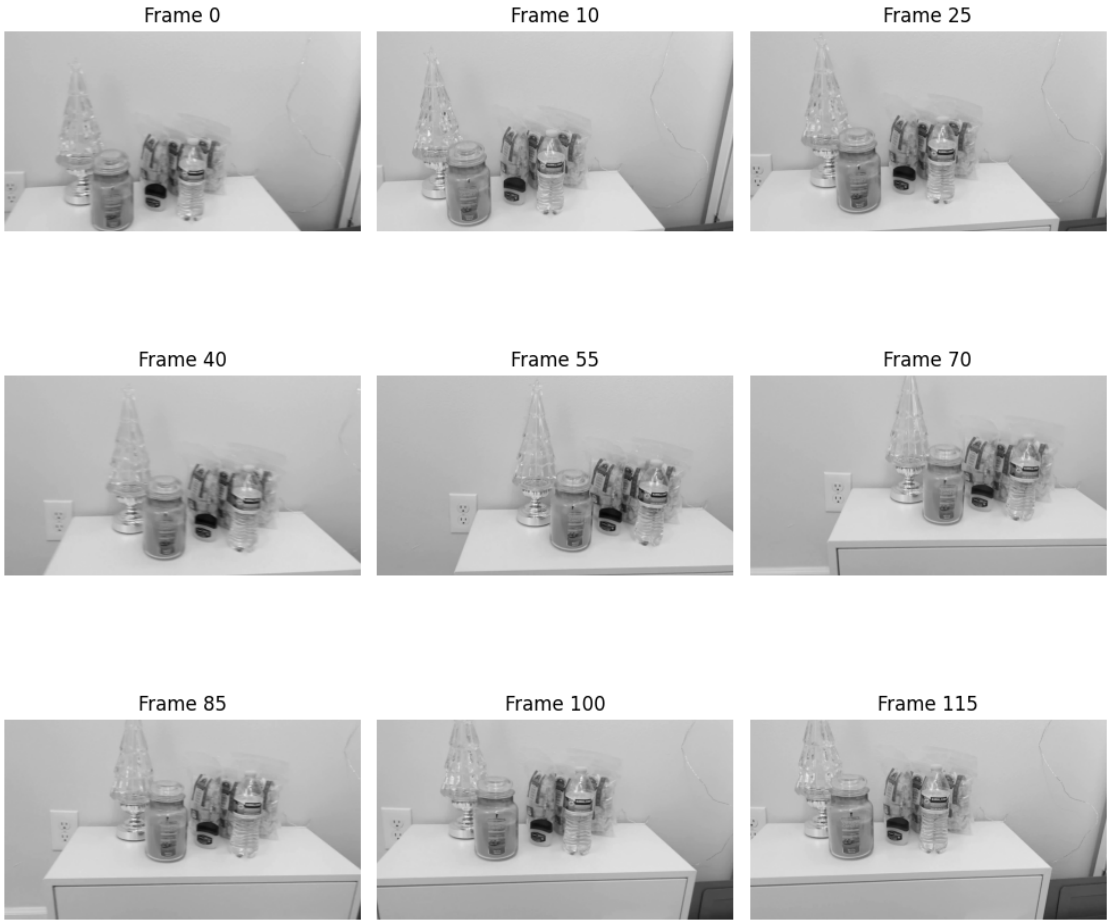
```
Mounted at /content/drive
```

[2]:
```
import os

# Set the working directory
working_directory = "/content/drive/My Drive/Colab Notebooks/ComputerImaging/
 ↪Problem Set 5"
os.chdir(working_directory)
```

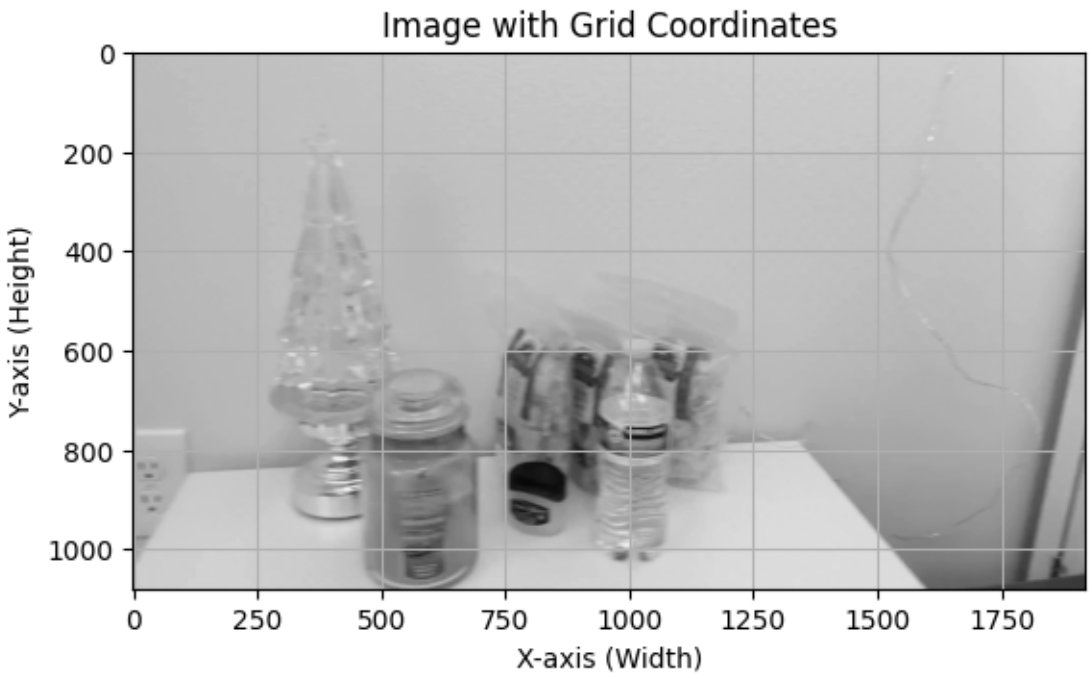
[3]:
```
# (b)
import cv2
import os
import numpy as np
import matplotlib.pyplot as plt

# Load the video
video_path = "video.mp4"  # Converted from MOV and MP4 and loaded
output_folder = "frames_output"

# Create folder to save frames
if not os.path.exists(output_folder):
    os.makedirs(output_folder)

# Extract frames
video = cv2.VideoCapture(video_path)
frame_counter = 0
frames = []  # Store frames for later visualization

while video.isOpened():
    flag, frame = video.read()
    if not flag:
        break  # Exit when no more frames

    # Convert to grayscale
    gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

```
    # Save frame
    frame_filename = os.path.join(output_folder, f"frame_{frame_counter:04d}.
 ↪png")
    cv2.imwrite(frame_filename, gray_frame)

    # Store every 5th frame for visualization
    if frame_counter % 5 == 0:
        frames.append(gray_frame)

    frame_counter += 1

video.release()
cv2.destroyAllWindows()

print(f"Extracted {frame_counter} frames and saved in {output_folder}")

# Display a 3x3 grid of selected frames
num_images = min(9, len(frames))
selected_frames = np.linspace(0, len(frames) - 1, num_images, dtype=int)

fig, axes = plt.subplots(3, 3, figsize=(10, 10))

for i, ax in enumerate(axes.ravel()):
    if i < num_images:
        ax.imshow(frames[selected_frames[i]], cmap='gray')
        ax.set_title(f"Frame {selected_frames[i]*5}")
        ax.axis("off")

plt.tight_layout()
plt.show()
```

```
Extracted 117 frames and saved in frames_output
```



[4]:
```
import matplotlib.pyplot as plt

# Display the image with axes
plt.imshow(frames[0], cmap='gray')
plt.xlabel("X-axis (Width)")
plt.ylabel("Y-axis (Height)")
plt.title("Image with Grid Coordinates")
plt.grid(True)  # Show grid for better visualization
plt.show()
```

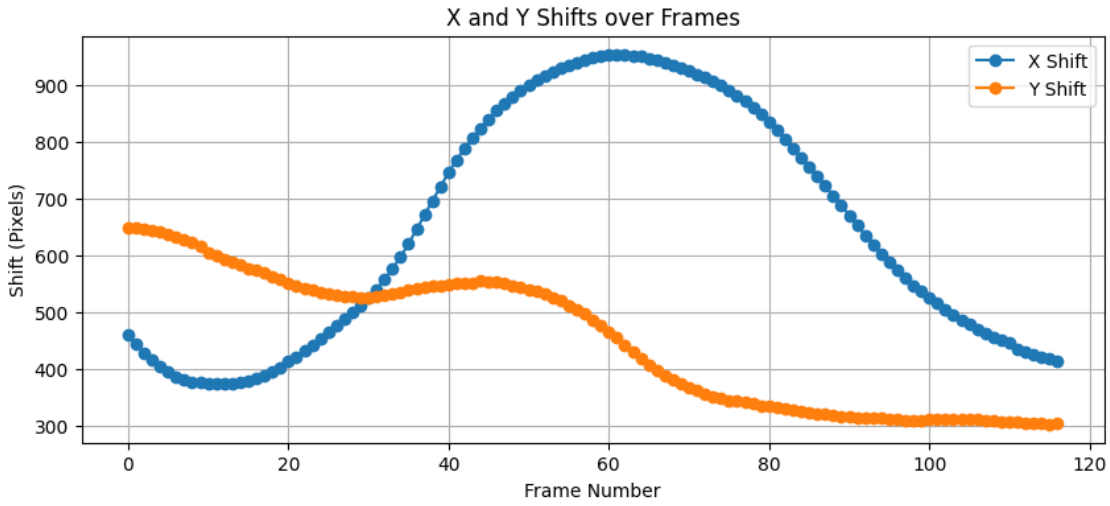


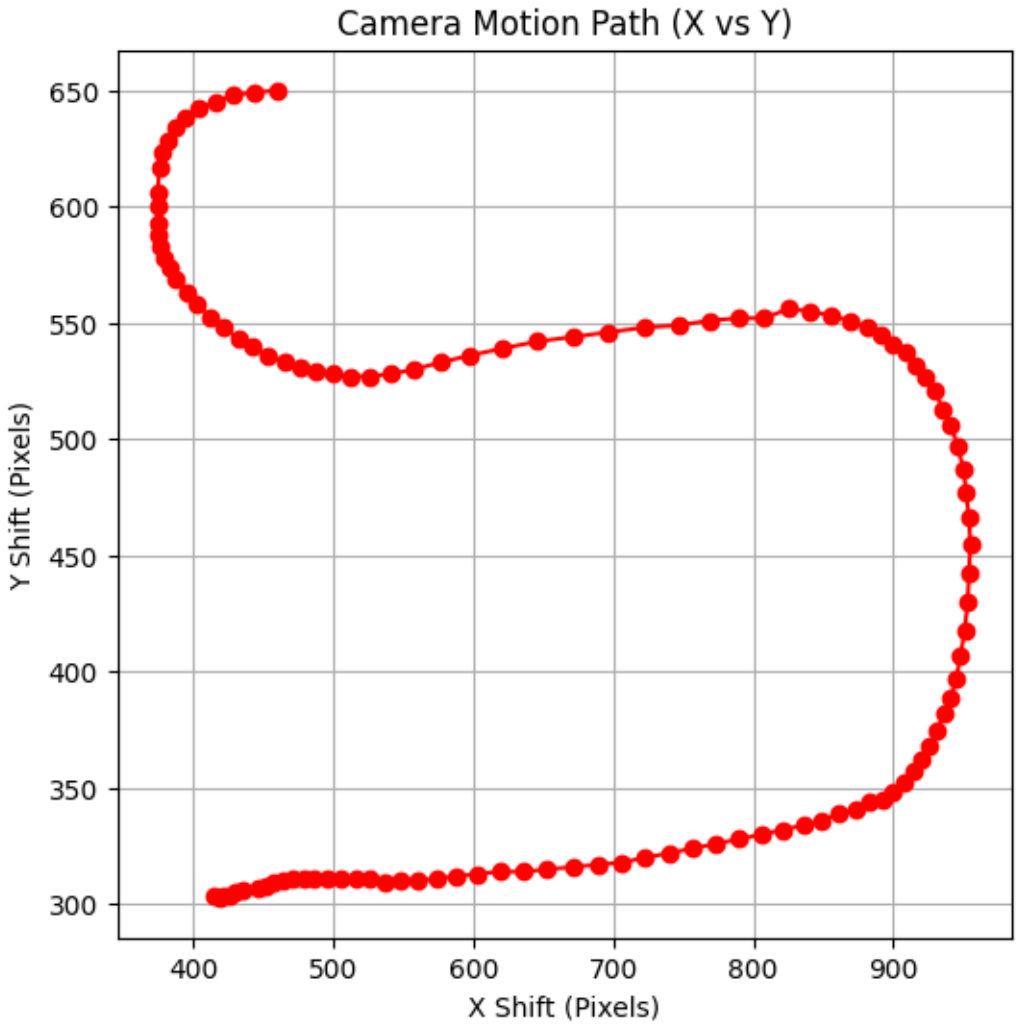
[5]:
```python
# (c)
import cv2
import matplotlib.pyplot as plt

first_frame = frames[0]

# Define ROI coordinates
x = 460
y = 650
w = 240
h = 1000

# Crop the selected region
cropped_template = first_frame[y:y+h, x:x+w]

# Display the cropped template
plt.figure(figsize=(4, 4))
plt.imshow(cv2.cvtColor(cropped_template, cv2.COLOR_BGR2RGB))
plt.title("Cropped Template")
plt.axis("off")
plt.show()

# Save the cropped template
cv2.imwrite("cropped_template.png", cropped_template)
print("Cropped template saved as 'cropped_template.png'")
```


```
Cropped template saved as 'cropped_template.png'
```

[6]:
```
# (d) template matching using normalized cross-correlation
import cv2
import numpy as np
import matplotlib.pyplot as plt
import os

# Load the saved cropped template
template = cv2.imread("cropped_template.png", cv2.IMREAD_GRAYSCALE)
template_h, template_w = template.shape  # Get template dimensions

# Define folder where frames are saved
output_folder = "frames_output"
frame_files = sorted([f for f in os.listdir(output_folder) if f.endswith(".
 ↪png")])  # Sorted frame list

# Store x, y shifts for all frames
shifts = []

# Perform template matching for each frame
for frame_file in frame_files:
    frame_path = os.path.join(output_folder, frame_file)
    frame = cv2.imread(frame_path, cv2.IMREAD_GRAYSCALE)
```

```
    # Apply template matching (Normalized Cross-Correlation)
    result = cv2.matchTemplate(frame, template, cv2.TM_CCOEFF_NORMED)

    # Find best match location
    min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(result)
    best_match_x, best_match_y = max_loc  # Top-left corner of best match

    # Store x and y shifts
    shifts.append([best_match_x, best_match_y])

# Convert to numpy array
shifts = np.array(shifts)

# Plot x and y shifts as a function of frame number
plt.figure(figsize=(10, 4))
plt.plot(shifts[:, 0], label="X Shift", marker="o")
plt.plot(shifts[:, 1], label="Y Shift", marker="o")
plt.xlabel("Frame Number")
plt.ylabel("Shift (Pixels)")
plt.title("X and Y Shifts over Frames")
plt.legend()
plt.grid(True)
plt.show()

# Plot X Shift vs Y Shift to visualize trajectory
plt.figure(figsize=(6, 6))
plt.plot(shifts[:, 0], shifts[:, 1], marker="o", linestyle="-", color="r")
plt.xlabel("X Shift (Pixels)")
plt.ylabel("Y Shift (Pixels)")
plt.title("Camera Motion Path (X vs Y)")
plt.grid(True)
plt.show()
```





[7]:
```
# (e)
import cv2
import numpy as np
import os
import matplotlib.pyplot as plt

# Load all frames
frame_files = sorted([f for f in os.listdir(output_folder) if f.endswith(".
 ↪png")])
frame_count = len(frame_files)

# Read the first frame to get dimensions
```

```
first_frame = cv2.imread(os.path.join(output_folder, frame_files[0]), cv2.
 ↪IMREAD_GRAYSCALE)
height, width = first_frame.shape

# Compute max shift values to determine expanded canvas size
max_shift_x = int(np.max(np.abs(shifts[0, :])))
max_shift_y = int(np.max(np.abs(shifts[1, :])))

# New canvas size
new_width = width + 2 * max_shift_x
new_height = height + 2 * max_shift_y

# Create an empty accumulator image for summing shifted frames
accumulated_image = np.zeros((new_height, new_width), dtype=np.float32)

# Offset shifts to fit within the new canvas
shift_offset_x = max_shift_x
shift_offset_y = max_shift_y

for i, frame_file in enumerate(frame_files):
    frame_path = os.path.join(output_folder, frame_file)
    frame = cv2.imread(frame_path, cv2.IMREAD_GRAYSCALE)

    # Get the corresponding shift (invert direction)
    shift_x, shift_y = -shifts[i, 0], -shifts[i, 1]

    # Adjust shifts to be within the expanded canvas
    shift_x += shift_offset_x
    shift_y += shift_offset_y

    # Create transformation matrix
    translation_matrix = np.float32([[1, 0, shift_x], [0, 1, shift_y]])

    # Apply translation using warpAffine to expanded canvas
    shifted_frame = cv2.warpAffine(frame, translation_matrix, (new_width,␣
 ↪new_height), borderMode=cv2.BORDER_CONSTANT, borderValue=0)

    # Accumulate the shifted frame
    accumulated_image += shifted_frame

# Compute a binary mask of nonzero pixels
mask = accumulated_image > 0  # True for valid pixels, False for black areas

# Find nonzero pixel locations
coords = np.argwhere(mask)

# Determine bounding box
```

```
if coords.size > 0:
    y_min, x_min = coords.min(axis=0)
    y_max, x_max = coords.max(axis=0)

    # Crop the accumulated image
    cropped_image = accumulated_image[y_min:y_max+1, x_min:x_max+1]
else:
    cropped_image = accumulated_image  # Fallback to original if no cropping␣
 ↪needed

# Normalize and convert back to 8-bit
cropped_image /= frame_count
cropped_image = np.clip(cropped_image, 0, 255)
final_image = np.uint8(cropped_image)

# Display the cropped final refocused image
plt.figure(figsize=(8, 8))
plt.imshow(final_image, cmap='gray')
plt.title("Refocused Image")
plt.axis("off")
plt.show()

# Save the cropped image
cv2.imwrite("refocused_image.png", final_image)
```

[7]: True

(f) [CS410=10 points | CS510=5 points] What is the runtime complexity of the algorithm in part (d)? (Express it using big-Oh notation in terms of the pixel resolution of the image and the number of frames.)

- Let N represent the number of frames.
- Let $H_f \times W_f$ reprresents the resolution of each image (height, weight)
- Let $H_t \times W_t$ represents the resolution of the template (height, weight)

We have 5 steps for (d).

1. Reading the template: $O(1)$ since we have the template stored in memory
2. Looping over frames: it runs N frames since the algorithm processes each frame one by one
3. Template matching (for each frame): I used the `cv2.matchTemplate` for this, and this function perfroms a sliding window operation over the image, where the template is compared to every possible location in the frame. The number of locations to check in the frame is size of the frame - size of the template. Which is: $(H_f - H_t + 1) \times (W_f - W_t + 1)$. Also, for each location, it computes a similarity measure between the template and the region of the frame, which is $O(H_t \times W_t)$ operation. So for step 3, the total time complexity is: $O(((H_f - H_t + 1) \times (W_f - W_t + 1)) \times (H_t \times W_t)) = O(H_f \times W_f \times H_t \times W_t)$ since H_f, W_f are much larger than H_t, W_t.
4. Finding the best matching using minMaxLoc: To find the maximum correlation value and its location, `cv2.minMaxLoc` used a linear scan through the result matrix, which is $(H_f - H_t + 1) \times (W_f - W_t + 1) = H_f \times W_f$. The complexity for this step is $O(H_f \times W_f)$.
5. Storing shift: $O(N)$. For each frame we store the results, so $O(N)$.

Thus, the overall complexity for processing all frame is: $O(N \times H_f \times W_f \times H_t \times W_t)$

[CS510=5 points] Propose some modifications to the algorithm to speed it up. (No need to implement them; only mention some speculative ideas in your write-up).

I found that there is other algorithms we can consider like SSD (sum of squared differences). Current method uses normalized cross-correlation (NCC), but by using SSD we can reduce the computational load by avoiding full normalization.

Another modification we can think of is: FFT-based Template Matching. Instead of performing the sliding window comparison pixel by pixel, we could use Fast Fourier Transform (FFT)-based techniques to speed up the convolution process. This method has a complexity of $O(N \log N)$ and can significantly speed up template matching, especially for larger images.

[ ]:
```
!pip install nbconvert >/dev/null
!apt-get install texlive texlive-xetex texlive-latex-extra pandoc >/dev/null
```

```
Extracting templates from packages: 100%
```

[ ]:
```
!jupyter nbconvert --to pdf --PDFExporter.verbose=True "/content/drive/MyDrive/
↪Colab Notebooks/ComputerImaging/Problem Set 5/CS510 Computer Imaging PS5␣
↪Light Fields.ipynb" >/dev/null
```