Experiment 1.4

Aim: Write a program to evaluate the efficacy of human-guided control point selection for image alignment.

Software Required: Any Python IDE (e.g., PyCharm, Jupyter Notebook, GoogleColab)

Description: Human-guided control point selection is a technique used in image alignment tasks where a human operator manually selects a set of control points to guide the alignment process. These control points are typically landmarks or distinctive features in the images that can be easily identified by a human observer. The overview of the process:

- Image Pair Selection: Choose a pair of images that need to be aligned. These images can be overlapping or have a known transformation between them, such as images of the same scene taken from different viewpoints.
- Control Point Selection: Display the two images side by side and allow the human operator to interactively select control points in both images. The operator can use a mouse or other input device to click on corresponding points in the two images. Ideally, the control points should be distinctive features that can be easily identified in both images.
- Control Point Matching: After the operator selects control points in both images, the corresponding control points need to be identified automatically. This is done by matching the selected control points in one image to their corresponding points in the other image. Various matching techniques can be employed, such as feature-based matching algorithms like SIFT or SURF, or geometric matching methods like RANSAC.
- Transformation Estimation: Once the control point correspondences are established, a transformation model can be estimated to align the images. Common transformation models include translation, rotation, scaling, and affine transformations. More complex transformations, such as projective transformations or non-linear deformations, can also be considered depending on the alignment requirements.
- Alignment and Image Warping: Using the estimated transformation model, the images can be aligned by warping one image to match the

other. The transformation can be applied to individual pixels or regions of the image using techniques like interpolation or resampling.

- Evaluation and Refinement: It is important to evaluate the quality of the alignment and visually inspect the results. If necessary, the human operator can adjust or add more control points to refine the alignment. Iterative refinement can be performed until the desired alignment accuracy is achieved.

Human-guided control point selection allows the human operator to leverage their visual perception and domain knowledge to guide the alignment process. It is particularly useful when dealing with challenging alignment scenarios or when accurate alignment is critical. However, it can be time-consuming and subjective, as the quality of the alignment heavily relies on the operator's expertise. Therefore, automated techniques such as feature-based matching or deep learning-based approaches can be employed to reduce human involvement or assist in the control point selection process

Steps:

- Load the source and target images.
- Display the source and target images to the user.
- Prompt the user to select corresponding control points on both images.
- Store the selected control points for alignment.
- Perform image alignment using the control points.
- Display the aligned image to the user.
- Evaluate the alignment quality using metrics such as mean square error or structural similarity index.
- Calculate the efficacy of human-guided control point selection by comparing the alignment results with and without user intervention.
- Output the evaluation results and efficacy metrics.

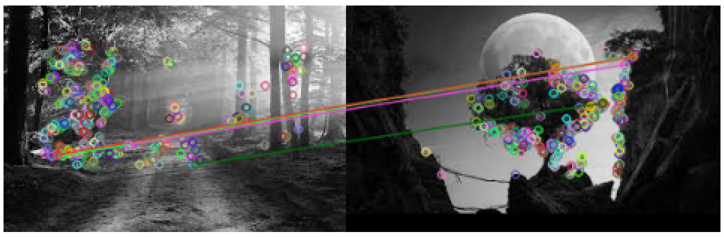
Code:

```
pip install matplotlib

# Import necessary libraries
import cv2
import numpy as np
from skimage.metrics import mean_squared_error, structural_similarity
from google.colab.patches import cv2_imshow  # Import cv2_imshow for image display in Colab
# Function to load an image
def load_image(image_path):
    return cv2.imread(image_path)
# Function to display an image
def display_image(image, title="Image"):
    cv2_imshow(image)  # Use cv2_imshow to display images in Colab
# Load the source and target images
source_image = load_image("/content/imag.png")
target_image = load_image("/content/images.png")
# Display the source and target images to the user
display_image(source_image, "Source Image")
display_image(target_image, "Target Image")

import cv2
import numpy as np
from google.colab.patches import cv2_imshow
# Function to load an image
def load_image(image_path):
    return cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)  # Load the image in grayscale
# Load the source and target images
source_image = load_image("/content/imag.png")
target_image = load_image("/content/images.png")
# Initialize the ORB (Oriented FAST and Rotated BRIEF) detector
orb = cv2.ORB_create()
# Find key points and descriptors in the images
source_keypoints, source_descriptors = orb.detectAndCompute(source_image, None)
target_keypoints, target_descriptors = orb.detectAndCompute(target_image, None)
# Initialize a Brute-Force Matcher
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
# Match the descriptors
matches = bf.match(source_descriptors, target_descriptors)
# Sort the matches by distance (smaller distances are better)
matches = sorted(matches, key=lambda x: x.distance)
# Take the top N matches (you can adjust this threshold)
top_matches = matches[:4]
# Get the corresponding points in both images
source_points = np.float32([source_keypoints[match.queryIdx].pt for match in
top_matches]).reshape(-1, 1, 2)
target_points = np.float32([target_keypoints[match.trainIdx].pt for match in top_matches]).reshape(-
1, 1, 2)
# Draw the matches
match_image = cv2.drawMatches(source_image, source_keypoints, target_image, target_keypoints,
top_matches, None)
# Display the matched points and matches
cv2_imshow(match_image)
cv2.waitKey(0)
cv2.destroyAllWindows()

# Function to perform image alignment using control points
def align_images(source_image, target_image, source_points, target_points):
    transformation_matrix, _ = cv2.estimateAffinePartial2D(
        source_points, target_points)
    aligned_image = cv2.warpAffine(source_image, transformation_matrix,
                                  (target_image.shape[1], target_image.shape[0]))
    return aligned_image
# Perform image alignment using the control points
aligned_image = align_images(source_image, target_image, source_points, target_points)
# Display the aligned image to the user
display_image(aligned_image, "Aligned Image")

# Function to evaluate alignment quality using metrics
def evaluate_alignment(aligned_image, target_image):
    mse = mean_squared_error(target_image, aligned_image)
    ssim = structural_similarity(target_image, aligned_image, channel_axis=None)  # Use
channel_axis=None
    return {"Mean Squared Error": mse, "Structural Similarity Index": ssim}
# Evaluate alignment quality
alignment_quality = evaluate_alignment(aligned_image, target_image)
print("Alignment Quality:")
print(alignment_quality)

# Resize source_image to match the dimensions of target_image
source_image = cv2.resize(source_image, (target_image.shape[1], target_image.shape[0]))

# Function to calculate the efficacy of human-guided control point selection
def calculate_efficacy(alignment_quality_no_selection, alignment_quality_with_selection):
    mse_no_selection = alignment_quality_no_selection["Mean Squared Error"]
    mse_with_selection = alignment_quality_with_selection["Mean Squared Error"]
    efficacy = 1 - (mse_with_selection / mse_no_selection)
    return efficacy
# Calculate the efficacy of human-guided control point selection
alignment_quality_no_selection = evaluate_alignment(target_image, source_image)
efficacy = calculate_efficacy(
    alignment_quality_no_selection, alignment_quality)
# Output the evaluation results and efficacy metrics
print("Efficacy of Human-Guided Control Point Selection:")
print(f"Efficacy: {efficacy * 100:.2f}%")
```

Implementation:

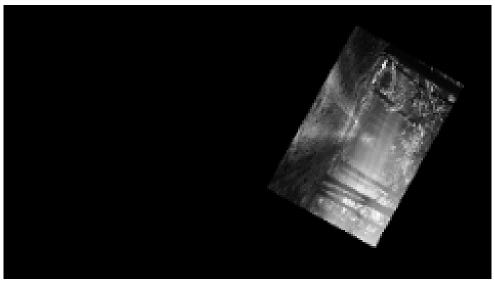
```python
# Function to load an image
def load_image(image_path):
    return cv2.imread(image_path)

# Function to display an image
def display_image(image, title="Image"):
    cv2_imshow(image)  # Use cv2_imshow to display images in Colab

# Load the source and target images
source_image = load_image("/content/imag.png")
target_image = load_image("/content/images.png")

# Display the source and target images to the user
display_image(source_image, "Source Image")
display_image(target_image, "Target Image")
```

```python
# Match the descriptors
matches = bf.match(source_descriptors, target_descriptors)

# Sort the matches by distance (smaller distances are better)
matches = sorted(matches, key=lambda x: x.distance)

# Take the top N matches (you can adjust this threshold)
top_matches = matches[:4]

# Get the corresponding points in both images
source_points = np.float32([source_keypoints[match.queryIdx].pt for match in top_matches]).reshape(-1, 1, 2)
target_points = np.float32([target_keypoints[match.trainIdx].pt for match in top_matches]).reshape(-1, 1, 2)

# Draw the matches
match_image = cv2.drawMatches(source_image, source_keypoints, target_image, target_keypoints, top_matches, None)

# Display the matched points and matches
cv2_imshow(match_image)
cv2.waitKey(0)
cv2.destroyAllWindows()

# Now you can proceed with the rest of your alignment and evaluation code using the matched points.
```



```python
[75] # Function to perform image alignment using control points
     def align_images(source_image, target_image, source_points, target_points):
```

```python
# Perform image alignment using the control points
aligned_image = align_images(source_image, target_image, source_points, target_points)

# Display the aligned image to the user
display_image(aligned_image, "Aligned Image")
```



```python
[76] # Function to evaluate alignment quality using metrics
     def evaluate_alignment(aligned_image, target_image):
         mse = mean_squared_error(target_image, aligned_image)
         ssim = structural_similarity(target_image, aligned_image, channel_axis=None)  # Use channel_axis=None
         return {"Mean Squared Error": mse, "Structural Similarity Index": ssim}

     # Evaluate alignment quality
     alignment_quality = evaluate_alignment(aligned_image, target_image)
     print("Alignment Quality:")
     print(alignment_quality)
```

```
Alignment Quality:
{'Mean Squared Error': 7606.229404761904, 'Structural Similarity Index': 0.036097332663618815}
```

```python
[78] # Resize source_image to match the dimensions of target_image
     source_image = cv2.resize(source_image, (target_image.shape[1], target_image.shape[0]))
```

```python
# Function to calculate the efficacy of human-guided control point selection
def calculate_efficacy(alignment_quality_no_selection, alignment_quality_with_selection):
    mse_no_selection = alignment_quality_no_selection["Mean Squared Error"]
    mse_with_selection = alignment_quality_with_selection["Mean Squared Error"]
    efficacy = 1 - (mse_with_selection / mse_no_selection)
    return efficacy

# Calculate the efficacy of human-guided control point selection
alignment_quality_no_selection = evaluate_alignment(target_image, source_image)
efficacy = calculate_efficacy(
    alignment_quality_no_selection, alignment_quality)

# Output the evaluation results and efficacy metrics
print("Efficacy of Human-Guided Control Point Selection:")
print(f"Efficacy: {efficacy * 100:.2f}%")
```

```
Efficacy of Human-Guided Control Point Selection:
Efficacy: -16.30%
```