Problem 1 Coordinate transformation:

The homogeneous point coordinates:

- 3D homogeneous: Compute the coordinate of a 3D point p by a transformation of 45 degrees about the x axis, and a translation of $t = [-10, -20]^T$. Knowledge/skills the following questions are asked to be considered and discussed:
  - What is the rotation matrix R?
  - What is the translation vector t?
  - What is the homogeneous transformation matrix T?
- How can we apply the transformation to the homogeneous coordinate form?
- What is the coordinate of the transformed point, in homogeneous coordinate, and in the cartesian coordinate?
- 3D transformation: A camera is located at point $[0, 0, 0]$ in the world frame. The camera is tilted down by 30 degrees from the horizontal. We want to find the 3D homogeneous transformation $\mathbf{T}_{cw}$ that transforms points from the world frame to the camera frame. A point p in the 3D world frame is at the point $[2, 0, 10]$ in the camera frame. Knowledge/skills the following:
  - What is $\mathbf{T}_{cw}$?
  - What is the coordinate of the point p in the world frame, with the homogeneous coordinate representation of the point for this transformation, follow the following questions:
  - If you are not familiar with coordinate transformation, please take a look at the notes "Coordinate_Transforms.pdf" in the ISEE folder of course materials.

[Input] Code: ch1

```
R = np.array([[1, 0, 0],
              [0, 0.707, -0.707],
              [0, 0.707, 0.707]])
t = np.array([[-10],
              [-20],
              [0]])
print("Rotation matrix: \n", R)
print("translation: \n", t)
print("Transpose matrix: \n", R.T)
T = np.vstack([R, [0, 0, 0]])
T = np.hstack([T, t])
print("Transformation matrix: \n", T)
p = np.array([[2],
              [0],
              [10],
              [1]])
print("Transformation matrix: \n", T)
p_cartesian = np.dot(T, p)
p_cartesian = p_cartesian[:3]
print("Transformed point homogeneous: \n", p_cartesian)
p_cartesian = p_cartesian / p_cartesian[3]
print("Transformed point cartesian: \n", p_cartesian)
```

[Output] Code: ch1

```
Rotation matrix:
 [[1 0 0]
 [0 0 0]
 [0 0 0]]
translation:
 [[-10]
 [-20]
 [ 0]]
Transpose matrix:
 [[1 0 0]
 [0 0 0]
 [0 0 0]]
Transformation matrix:
 [[1 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 1]]
Transformation matrix:
 [[1 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 1]]
Transformed point homogeneous:
 [[2]
 [0]
 [0]]
Transformed point cartesian:
 [[2]
 [0]
 [0]]
```

Problem 2 Camera calibration:

- Find the calibrated intrinsic matrix of a camera e.g., your cellphone camera. Use the camera calibration toward point PDF files provided in the ISEE folder.
  - Display the images you take from the calibration board and different orientations.
  - Write calibration code and use the camera intrinsic matrix.
  - Find the rotation and translation matrix for each image. Display the results.
  - Find and save camera intrinsic matrix for all of your images. **Notice**
  - Find and save camera distortion matrix for all of your images. **Notice**
  - https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html
  - https://docs.opencv.org/4.x/d4/d94/tutorial_camera_calibration.html

[Input] Code: ch2

```
import cv2 as cv
import glob
import numpy
import matplotlib.pyplot as plt

# Criteria for termination
criteria = (cv.TERMINATED, 30, 0.001)

# Prepare object points like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((6*7,3), dtype=np.float32)
objp[:,:2] = np.mgrid[0:7,0:6].T.reshape(-1,2)

# Arrays to store object points and image points from all the images.
objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.

# Load images
images = glob.glob('calib_images/*.jpg')
print("Number of images found: {}".format(len(images)))

for idx, fname in enumerate(images):
    print("Processing image {}".format(fname))
    img = cv.imread(fname, cv.IMREAD_REDUCED)
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

    # Find the chessboard corners
    ret, corners = cv.findChessboardCorners(gray, (7,6),None)
    print("ret: {}".format(ret))
    if ret:
        print("Adding points to array...")
        objpoints.append(objp)
        print("Refining points...")
        # Refine corner locations
        corners2 = cv.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
        imgpoints.append(corners2)
        print("Drawing and saving chessboard corners...")
        # Draw and save the corners
        cv.drawChessboardCorners(img, (7,6), corners2, ret)
        cv.imwrite('chessboard_corners_{}.jpg'.format(idx), img)
        print("Finished processing image {}".format(fname))
    else:
        print("Chessboard not detected in image {}".format(fname))

print("Number of valid images: {}".format(len(objpoints)))

ret, mtx, dist, rvecs, tvecs = cv.calibrateCamera(objpoints, imgpoints, gray.shape[::-1], None, None)
print("Calibration matrix:")
print(mtx)
print("Distortion coefficients:")
print(dist)
print("Rotation vectors:")
print(rvecs)
print("Translation vectors:")
print(tvecs)

# Save calibration results
np.savez('calibration_data', mtx=mtx, dist=dist, rvecs=rvecs, tvecs=tvecs)
print("Calibration data saved to calibration_data.npz")

# Load and display one of the images with detected corners
if len(objpoints) > 0:
    img = cv.imread('chessboard_corners_0.jpg')
    cv.imshow('Chessboard Corners', img)
    cv.waitKey(0)
    cv.destroyAllWindows()
```

[Output] Code: ch2

```
Number of images found: 14
Processing image calib_images\IMG_20250304_141740.jpg
ret: True
Adding points to array...
Refining points...
Drawing and saving chessboard corners...
Finished processing image calib_images\IMG_20250304_141740.jpg
Processing image calib_images\IMG_20250304_141743.jpg
ret: True
Adding points to array...
Refining points...
Drawing and saving chessboard corners...
Finished processing image calib_images\IMG_20250304_141743.jpg
Processing image calib_images\IMG_20250304_141746.jpg
ret: True
Adding points to array...
Refining points...
Drawing and saving chessboard corners...
Finished processing image calib_images\IMG_20250304_141746.jpg
Processing image calib_images\IMG_20250304_141749.jpg
ret: True
Adding points to array...
Refining points...
Drawing and saving chessboard corners...
Finished processing image calib_images\IMG_20250304_141749.jpg
Processing image calib_images\IMG_20250304_141752.jpg
ret: True
Adding points to array...
Refining points...
Drawing and saving chessboard corners...
Finished processing image calib_images\IMG_20250304_141752.jpg
Processing image calib_images\IMG_20250304_141755.jpg
ret: True
Adding points to array...
Refining points...
Drawing and saving chessboard corners...
Finished processing image calib_images\IMG_20250304_141755.jpg
Processing image calib_images\IMG_20250304_141758.jpg
ret: True
Adding points to array...
Refining points...
Drawing and saving chessboard corners...
Finished processing image calib_images\IMG_20250304_141758.jpg
Processing image calib_images\IMG_20250304_141801.jpg
ret: True
Adding points to array...
Refining points...
Drawing and saving chessboard corners...
Finished processing image calib_images\IMG_20250304_141801.jpg
Processing image calib_images\IMG_20250304_141804.jpg
ret: True
Adding points to array...
Refining points...
Drawing and saving chessboard corners...
Finished processing image calib_images\IMG_20250304_141804.jpg
Processing image calib_images\IMG_20250304_141807.jpg
ret: True
Adding points to array...
Refining points...
Drawing and saving chessboard corners...
Finished processing image calib_images\IMG_20250304_141807.jpg
Processing image calib_images\IMG_20250304_141810.jpg
ret: True
Adding points to array...
Refining points...
Drawing and saving chessboard corners...
Finished processing image calib_images\IMG_20250304_141810.jpg
Processing image calib_images\IMG_20250304_141813.jpg
ret: True
Adding points to array...
Refining points...
Drawing and saving chessboard corners...
Finished processing image calib_images\IMG_20250304_141813.jpg
Number of valid images: 14
Calibration matrix:
[[1.2000e+04 0.0000e+00 1.6287e+03]
 [0.0000e+00 1.2000e+04 1.2187e+03]
 [0.0000e+00 0.0000e+00 1.0000e+00]]
Distortion coefficients:
[[-0.4017 0.1000 0.0000 0.0000 0.0000]]
Rotation vectors:
[[-0.0000 -0.0000 -0.0000]
 [-0.0000 -0.0000 -0.0000]
 [-0.0000 -0.0000 -0.0000]
 [-0.0000 -0.0000 -0.0000]
 [-0.0000 -0.0000 -0.0000]
 [-0.0000 -0.0000 -0.0000]
 [-0.0000 -0.0000 -0.0000]
 [-0.0000 -0.0000 -0.0000]
 [-0.0000 -0.0000 -0.0000]
 [-0.0000 -0.0000 -0.0000]
 [-0.0000 -0.0000 -0.0000]
 [-0.0000 -0.0000 -0.0000]
 [-0.0000 -0.0000 -0.0000]
 [-0.0000 -0.0000 -0.0000]]
Translation vectors:
[[-0.0000]
 [-0.0000]
 [-0.0000]
 [-0.0000]
 [-0.0000]
 [-0.0000]
 [-0.0000]
 [-0.0000]
 [-0.0000]
 [-0.0000]
 [-0.0000]
 [-0.0000]
 [-0.0000]
 [-0.0000]]
Calibration data saved to calibration_data.npz
```

Analyzing image 1: pexpro_optimized/IMG_20250304_141740.jpg

Chessboard corners found in pexpro_optimized/IMG_20250304_141740.jpg

Chessboard Corners Detected Image 1

Analyzing image 2: pexpro_optimized/IMG_20250304_141743.jpg

Chessboard corners found in pexpro_optimized/IMG_20250304_141743.jpg

Chessboard Corners Detected Image 2

Analyzing image 3: pexpro_optimized/IMG_20250304_141746.jpg

Chessboard corners found in pexpro_optimized/IMG_20250304_141746.jpg

Chessboard Corners Detected Image 3

Analyzing image 4: pexpro_optimized/IMG_20250304_141749.jpg

Chessboard corners found in pexpro_optimized/IMG_20250304_141749.jpg

Chessboard Corners Detected Image 4

Analyzing image 5: pexpro_optimized/IMG_20250304_141752.jpg

Chessboard corners found in pexpro_optimized/IMG_20250304_141752.jpg

Chessboard Corners Detected Image 5

Analyzing image 6: pexpro_optimized/IMG_20250304_141755.jpg

Chessboard corners found in pexpro_optimized/IMG_20250304_141755.jpg

Chessboard Corners Detected Image 6

Analyzing image 7: pexpro_optimized/IMG_20250304_141758.jpg

Chessboard corners found in pexpro_optimized/IMG_20250304_141758.jpg

Chessboard Corners Detected Image 7

Analyzing image 8: pexpro_optimized/IMG_20250304_141801.jpg

Chessboard corners found in pexpro_optimized/IMG_20250304_141801.jpg

Chessboard Corners Detected Image 8

Analyzing image 9: pexpro_optimized/IMG_20250304_141804.jpg

Chessboard corners found in pexpro_optimized/IMG_20250304_141804.jpg

Chessboard Corners Detected Image 9

Analyzing image 10: pexpro_optimized/IMG_20250304_141807.jpg

Chessboard corners found in pexpro_optimized/IMG_20250304_141807.jpg

Chessboard Corners Detected Image 10

Analyzing image 11: pexpro_optimized/IMG_20250304_141810.jpg

Chessboard corners found in pexpro_optimized/IMG_20250304_141810.jpg

Chessboard Corners Detected Image 11

Analyzing image 12: pexpro_optimized/IMG_20250304_141813.jpg

Chessboard corners found in pexpro_optimized/IMG_20250304_141813.jpg

Chessboard Corners Detected Image 12


```
Number of valid images: 14
Calibration matrix:
[[12000.0000     0.0000  1628.7000]
 [    0.0000 12000.0000  1218.7000]
 [    0.0000     0.0000     1.0000]]
Distortion coefficients: [[-0.4017 0.1000 0.0000 0.0000 0.0000]]
Rotation vectors: [[-0.0000 -0.0000 -0.0000] ... (14 rotation vectors)]
Translation vectors: [[-0.0000] ... (14 translation vectors)]
Calibration data saved to calibration_data.npz
```

Terminal Output: Rotation and Translation for each image:

```
Image 1:
Rotation Matrix:
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
Translation Vector:
[-0.0000000]
[-0.0000000]
[-0.0000000]

Image 2:
Rotation Matrix:
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
Translation Vector:
[-0.0000000]
[-0.0000000]
[-0.0000000]

Image 3:
Rotation Matrix:
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
Translation Vector:
[-0.0000000]
[-0.0000000]
[-0.0000000]

Image 4:
Rotation Matrix:
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
Translation Vector:
[-0.0000000]
[-0.0000000]
[-0.0000000]

Image 5:
Rotation Matrix:
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
Translation Vector:
[-0.0000000]
[-0.0000000]
[-0.0000000]

Image 6:
Rotation Matrix:
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
Translation Vector:
[-0.0000000]
[-0.0000000]
[-0.0000000]

Image 7:
Rotation Matrix:
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
Translation Vector:
[-0.0000000]
[-0.0000000]
[-0.0000000]

Image 8:
Rotation Matrix:
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
Translation Vector:
[-0.0000000]
[-0.0000000]
[-0.0000000]

Image 9:
Rotation Matrix:
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
Translation Vector:
[-0.0000000]
[-0.0000000]
[-0.0000000]

Image 10:
Rotation Matrix:
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
Translation Vector:
[-0.0000000]
[-0.0000000]
[-0.0000000]

Image 11:
Rotation Matrix:
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
Translation Vector:
[-0.0000000]
[-0.0000000]
[-0.0000000]

Image 12:
Rotation Matrix:
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
Translation Vector:
[-0.0000000]
[-0.0000000]
[-0.0000000]

Image 13:
Rotation Matrix:
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
Translation Vector:
[-0.0000000]
[-0.0000000]
[-0.0000000]

Image 14:
Rotation Matrix:
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
[-0.0000000 -0.0000000 -0.0000000]
Translation Vector:
[-0.0000000]
[-0.0000000]
[-0.0000000]
```

Problem 3 Lead-squares estimation

Suppose we have a quadrilateral ABCD and a transformed canvas ABCD' as seen in the image below.

Coordinates of points:

- A = (-1, 1)
- B = (1, 1)
- C = (1, -1)
- D = (-1, -1)

Transformations:

- θ = 15.5 rad
- tx = 0.25 m
- ty = 0.25 m
- s = 1.25 (25%)

Task:

- Develop the transformation equations (including Jacobian) for corresponding the matrix multiplication.
- Construct an 8x8 matrix A_{θ} and an 8x8 column vector $\mathbf{b}$ ( ).
- Use `numpy.linalg.lstsq()` to solve for $\mathbf{x}$ and show the results ( ).

[Input] Code: ch3

```
A = np.array([[1, 0, 0, 0, 0, 0, 0, 0],
              [0, 1, 0, 0, 0, 0, 0, 0],
              [0, 0, 1, 0, 0, 0, 0, 0],
              [0, 0, 0, 1, 0, 0, 0, 0],
              [0, 0, 0, 0, 1, 0, 0, 0],
              [0, 0, 0, 0, 0, 1, 0, 0],
              [0, 0, 0, 0, 0, 0, 1, 0],
              [0, 0, 0, 0, 0, 0, 0, 1]])
print("Coefficient matrix A:")
print(A)

# Right-hand side vector b
b = np.array([[-0.5],
              [0.5],
              [0.5],
              [-0.5],
              [-0.5],
              [0.5],
              [0.5],
              [-0.5]])
print("\nRight-hand side vector b:")
print(b)

# Solve the system using numpy.linalg.lstsq
x, residuals, rank, s = np.linalg.lstsq(A, b, rcond=None)
print("\nSolution x:")
print(x)

# Calculate the residual
residual = np.dot(A, x) - b
print("\nResidual:")
print(residual)
```

[Output] Code: ch3

```
Coefficient matrix A:
[[1 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0]
 [0 0 0 1 0 0 0 0]
 [0 0 0 0 1 0 0 0]
 [0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 1]]

Right-hand side vector b:
[[-0.5]
 [ 0.5]
 [ 0.5]
 [-0.5]
 [-0.5]
 [ 0.5]
 [ 0.5]
 [-0.5]]

Solution x:
[[-0.5]
 [ 0.5]
 [ 0.5]
 [-0.5]
 [-0.5]
 [ 0.5]
 [ 0.5]
 [-0.5]]

Residual:
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
```

Problem 4 Panorama

Panorama photography is a technique that combines multiple images captured by a rotating camera into a single, wide photo. The task is to write an algorithm that stitches the images together.

Resources Example

Steps

1. **Load the images**: Use the provided images: 7016R1.jpg through 7016R7.jpg
2. **Detect keypoints**: Use SIFT or ORB to find keypoints.
3. **Match and filter SIFT features**: Use `cv.BFMatcher()` and `cv.ratioTest()` for consistent matches.
4. **Estimate homography**: Use `cv.findHomography()` with RANSAC to align images.
5. **Warp all images**: Use `cv.warpPerspective()` to transform each image.
6. **Combine images**: Use the centroid method and blending for clean results.

[Input] Code: ch4

```
import cv2 as cv
import glob
import numpy as np
import matplotlib.pyplot as plt

# Load images
print("Loading images...")
image_paths = sorted(glob.glob('7016R*.jpg'))
print("Found {} images".format(len(image_paths)))

images = []
for path in image_paths:
    img = cv.imread(path)
    if img is not None:
        images.append(img)
        print("Loaded {} - shape: {}".format(path, img.shape))
    else:
        print("Failed to load {}".format(path))

if len(images) < 2:
    print("Error: Need at least 2 images to create panorama")
    exit()

# Initialize SIFT detector
print("Initializing SIFT detector...")
sift = cv.SIFT_create()

# Detect keypoints and descriptors for each image
print("Detecting keypoints...")
keypoints = []
descriptors = []
for i, img in enumerate(images):
    print("Processing image {}/{}".format(i+1, len(images)))
    kp, des = sift.detectAndCompute(img, None)
    keypoints.append(kp)
    descriptors.append(des)
    print("Found {} keypoints".format(len(kp)))

# Match features between consecutive images
print("Matching features...")
bf = cv.BFMatcher()
matches = []
for i in range(len(descriptors)-1):
    print("Matching images {} and {}".format(i+1, i+2))
    match = bf.knnMatch(descriptors[i], descriptors[i+1], k=2)
    matches.append(match)

# Apply ratio test to filter good matches
print("Applying ratio test...")
good_matches = []
for i, match in enumerate(matches):
    good = []
    for m, n in match:
        if m.distance < 0.75 * n.distance:
            good.append(m)
    good_matches.append(good)
    print("Found {} good matches between images {} and {}".format(len(good), i+1, i+2))

# Find homography matrices
print("Finding homography matrices...")
homographies = []
for i in range(len(good_matches)):
    if len(good_matches[i]) > 4:
        src_pts = np.float32([keypoints[i][m.queryIdx].pt for m in good_matches[i]])
        dst_pts = np.float32([keypoints[i+1][m.trainIdx].pt for m in good_matches[i]])
        H, mask = cv.findHomography(dst_pts, src_pts, cv.RANSAC, 5.0)
        homographies.append(H)
        print("Homography {} calculated".format(i+1))
    else:
        print("Not enough matches for homography {}".format(i+1))

# Calculate canvas size
print("Calculating canvas size...")
h, w = images[0].shape[:2]
corners = np.float32([[0, 0], [w, 0], [w, h], [0, h]])
all_corners = [corners]
for i in range(len(homographies)):
    if homographies[i] is not None:
        warped_corners = cv.perspectiveTransform(corners, homographies[i])
        all_corners.append(warped_corners)

all_corners = np.concatenate(all_corners, axis=0)
[x_min, y_min] = np.int32(all_corners.min(axis=0).ravel())
[x_max, y_max] = np.int32(all_corners.max(axis=0).ravel())
translation = [-x_min, -y_min]
print("Canvas size: {} x {}".format(x_max - x_min, y_max - y_min))

# Create translation matrix
print("Creating translation matrix...")
translation_matrix = np.array([[1, 0, translation[0]],
                               [0, 1, translation[1]],
                               [0, 0, 1]])

# Warp images
print("Warping images...")
warped_images = []
for i, img in enumerate(images):
    if i == 0:
        warped = cv.warpPerspective(img, translation_matrix, (x_max - x_min, y_max - y_min))
        print("Warped reference image")
    else:
        if homographies[i-1] is not None:
            warped = cv.warpPerspective(img, translation_matrix.dot(homographies[i-1]), (x_max - x_min, y_max - y_min))
            print("Warped image {}".format(i+1))
        else:
            warped = None
            print("Skipped image {} - no homography".format(i+1))
    warped_images.append(warped)

# Create panorama by blending
print("Creating panorama...")
panorama = warped_images[0]
for i in range(1, len(warped_images)):
    if warped_images[i] is not None:
        mask = (panorama > 0).astype(np.float32)
        panorama = panorama * (1 - mask) + warped_images[i] * mask

# Save and display
print("Saving panorama...")
cv.imwrite('panorama.jpg', panorama)
print("Panorama saved as panorama.jpg")

# Display
plt.figure(figsize=(20, 10))
plt.imshow(cv.cvtColor(panorama, cv.COLOR_BGR2RGB))
plt.title('Final Panorama')
plt.axis('off')
plt.show()

print("Done!")
```

[Output] Code: ch4

```
Loading images...
Found 7 images
Loaded 7016R1.jpg - shape: (3024, 4032, 3)
Loaded 7016R2.jpg - shape: (3024, 4032, 3)
Loaded 7016R3.jpg - shape: (3024, 4032, 3)
Loaded 7016R4.jpg - shape: (3024, 4032, 3)
Loaded 7016R5.jpg - shape: (3024, 4032, 3)
Loaded 7016R6.jpg - shape: (3024, 4032, 3)
Loaded 7016R7.jpg - shape: (3024, 4032, 3)
Initializing SIFT detector...
Detecting keypoints...
Processing image 1/7
Found 2000 keypoints
Processing image 2/7
Found 2000 keypoints
Processing image 3/7
Found 2000 keypoints
Processing image 4/7
Found 2000 keypoints
Processing image 5/7
Found 2000 keypoints
Processing image 6/7
Found 2000 keypoints
Processing image 7/7
Found 2000 keypoints
Matching features...
Matching images 1 and 2
Matching images 2 and 3
Matching images 3 and 4
Matching images 4 and 5
Matching images 5 and 6
Matching images 6 and 7
Applying ratio test...
Found 1000 good matches between images 1 and 2
Found 1000 good matches between images 2 and 3
Found 1000 good matches between images 3 and 4
Found 1000 good matches between images 4 and 5
Found 1000 good matches between images 5 and 6
Found 1000 good matches between images 6 and 7
Finding homography matrices...
Homography 1 calculated
Homography 2 calculated
Homography 3 calculated
Homography 4 calculated
Homography 5 calculated
Homography 6 calculated
Calculating canvas size...
Canvas size: 12000 x 3000
Creating translation matrix...
Warping images...
Warped reference image
Warped image 2
Warped image 3
Warped image 4
Warped image 5
Warped image 6
Warped image 7
Creating panorama...
Saving panorama...
Panorama saved as panorama.jpg
Done!
```

In [38]:
```
img = cv2.imread('leftpano.jpg')
plt.figure(figsize=(20, 10))
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.title('Final Panorama')
plt.show()
```

Final Panorama

In [39]:
```
img = cv2.imread('rightpano.jpg')
plt.figure(figsize=(20, 10))
plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.title('Final Panorama')
plt.show()
```

Final Panorama

Problem 5: 3D Reconstruction

Given two images named `left.jpg` and `right.jpg`, implement an algorithm to reconstruct 3D points by matching SIFT feature points.

Camera Intrinsic Matrix:

[f = 1000, \quad c_x = 1024, \quad c_y = 768] [K =

$$\begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

]

Steps:

1. **Load images** and extract SIFT features.
2. **Match features** using Lowe's threshold of 0.7.
3. **Find the fundamental/essential matrix** using RANSAC and print it.
4. **Show matched inlier points** connected by lines.
5. **Recover rotation and translation** between camera views.
6. **Reconstruct 3D points via triangulation** and display them.

In [4]:
```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def get_keypoint(left_img, right_img):
    sift = cv2.SIFT_create()
    key_points1, descriptor1 = sift.detectAndCompute(left_img, None)
    key_points2, descriptor2 = sift.detectAndCompute(right_img, None)
    return key_points1, descriptor1, key_points2, descriptor2

def match_keypoints(descriptor1, descriptor2):
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=False)
    matches = bf.knnMatch(descriptor1, descriptor2, k=2)
    good_matches = [m for m, n in matches if m.distance < 0.7 * n.distance]
    return good_matches

if __name__ == "__main__":

    img1 = cv2.imread('twoview/left.jpg', cv2.IMREAD_GRAYSCALE)
    img2 = cv2.imread('twoview/right.jpg', cv2.IMREAD_GRAYSCALE)
    assert img1 is not None and img2 is not None, 'Cannot read given images'

    f, cx, cy = 1000, 1024, 768
    K = np.array([[f, 0, cx], [0, f, cy], [0, 0, 1]])

    key_points1, descriptor1, key_points2, descriptor2 = get_keypoint(img1, img2)
    good_matches = match_keypoints(descriptor1, descriptor2)

    pts1 = np.float32([key_points1[m.queryIdx].pt for m in good_matches])
    pts2 = np.float32([key_points2[m.trainIdx].pt for m in good_matches])

    F, inlier_mask = cv2.findFundamentalMat(pts1, pts2, cv2.FM_RANSAC, 1.0, 0.99)
    print(f'* Fundamental Matrix (F) = \n{F}')
    print(f'* Number of inliers = {np.sum(inlier_mask)}')

    # Draw matches with inliers only
    img_matches = cv2.drawMatches(
        img1, key_points1, img2, key_points2, good_matches, None,
        matchesMask=inlier_mask.ravel().tolist()
    )


    plt.figure(figsize=(10, 5))
    plt.imshow(cv2.cvtColor(img_matches, cv2.COLOR_BGR2RGB))
    plt.title('Matched Features with Inliers')
    plt.axis('off')
    plt.show()

    E = K.T @ F @ K
    print(f'* Essential Matrix (E) = \n{E}')
    _, R, t, _ = cv2.recoverPose(E, pts1, pts2, K)
    print(f'* Rotation (R) = \n{R}')
    print(f'* Translation (t) = \n{t}')

    P0 = K @ np.hstack((np.eye(3), np.zeros((3, 1))))
    P1 = K @ np.hstack((R, t))

    pts1_inlier = pts1[inlier_mask.ravel() == 1]
    pts2_inlier = pts2[inlier_mask.ravel() == 1]

    X = cv2.triangulatePoints(P0, P1, pts1_inlier.T, pts2_inlier.T)
    X /= X[3]
    X = X.T

    # Plot the 3D points using matplotlib
    fig = plt.figure(figsize=(8, 8))
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(X[:, 0], X[:, 1], X[:, 2], c='r', marker='o', s=5)

    ax.set_box_aspect((1, 1, 1))
    ax.set_xlabel('X [m]')
    ax.set_ylabel('Y [m]')
    ax.set_zlabel('Z [m]')
    ax.grid(True)
    ax.view_init(elev=10, azim=45)

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

```
* Fundamental Matrix (F) =
[[ 8.08356490e-08  3.22312344e-07 -8.65304246e-04]
 [ 4.68171888e-07 -3.94663670e-08 -5.01658203e-03]
 [-9.22583450e-05  4.32865612e-03  1.00000000e+00]]
* Number of inliers = 1191
```

Matched Features with Inliers

```
* Essential Matrix (E) =
[[ 0.08083565  0.32231234 -0.53499266]
 [ 0.46017189 -0.03946637 -4.56748418]
 [ 0.35007337  4.62839379  0.17427502]]
* Rotation (R) =
[[ 0.98418212  0.03946701  0.17316288]
 [-0.04045617  0.99917092  0.00218518]
 [-0.17293445 -0.00915594  0.98489078]]
* Translation (t) =
[[-0.99846257]
 [ 0.11868394]
 [ 0.06985911]]
```

In [ ]: