7CCEMSAP

Sensing and Perception

Homography

Planar Transformation in ROS2

Prepared by: Dr. S. Morris

2025-10-30

1 Learning Objectives

This tutorial introduces **Homography** - a fundamental concept in computer vision and robotics for understanding planar transformations between different viewpoints. Homography is essential for applications such as robot navigation, object recognition, and 3D scene understanding.

Prerequisites: - Working ROS2 environment - Basic familiarity with computer vision concepts - Understanding of geometric transformations

Learning Goals: - **Homography matrix** computation and properties - **Planar transformation** between different viewpoints - **Feature matching** for homography estimation - **Image warping** and perspective correction - **ROS2 integration** for real-time homography processing

2 What is Homography?

Homography is a transformation that maps points from one plane to another plane. It is particularly useful for:

- **Planar object recognition** from different viewpoints - **Perspective correction** and image rectification - **Camera pose estimation** from planar scenes - **Image stitching** and panorama creation - **Visual servoing** in robotics

Homography is widely used in robotics applications including: - Autonomous navigation and mapping - Object recognition and tracking - Augmented reality and visual servoing - Image registration and alignment

3 Step-by-Step Tutorial

3.1 Step 1: Setup

Add the Homography package to your existing ROS2 workspace:

```
# Navigate to your existing ROS2 workspace
cd ~/ros2_ws/src

# Clone the Homography repository
git clone https://github.kcl.ac.uk/7CCEMSAP/Homography.git homography

# Build the new package
cd ~/ros2_ws
colcon build --packages-select homography
source install/setup.bash
```

3.2 Step 2: Understanding the Code

Code Location: The complete working code has been provided for you. You can find the main Homography implementation in:

```
~/ros2_ws/src/homography/homography/homography_node.py
```

This file contains the complete solution that you can examine, run, and modify.

The Homography node does the following:

1. **Loads image pairs** from datasets 2. **Detects and matches features** between images 3. **Computes homography matrix** using RANSAC 4. **Warps images** using the homography transformation 5. **Visualizes results** and publishes to ROS2

Key parameters you can modify: - ‘max$_{f}eatures$‘ : $Maximumnumberoffeaturestodetect-$ ‘$match_{r}atio$‘ : $Lowe'sratiotestthreshold-$‘$ransac_{t}hreshold$‘ : $RANSACthresholdforhomographyestimation-$ ‘$warp_{m}ode$‘ : $Typeofimagewarpingtoperform$

3.3 Step 3: Code Breakdown

Tip: Open the code file in your editor to follow along:

```
nano ~/ros2_ws/src/homography/homography/homography_node.py
# or
vim ~/ros2_ws/src/homography/homography/homography_node.py
```

Let’s examine the Python code in detail:

3.3.1 **Homography Pipeline Overview**

The Homography node implements a complete planar transformation pipeline:

1. **Feature Detection** - Extract keypoints using SIFT detector 2. **Feature Matching** - Match features between image pairs 3. **Homography Estimation** - Compute transformation matrix using RANSAC 4. **Image Warping** - Apply homography to warp images 5. **Visualization** - Display results and transformations

3.3.2 **Homography Computation (Lines 45-65)**

```
def compute_homography(self, kp1, kp2, matches):
    # Extract matched points
    src_pts = np.float32([kp1[m.queryIdx].pt for m in matches]).reshape
        (-1, 1, 2)
    dst_pts = np.float32([kp2[m.trainIdx].pt for m in matches]).reshape
        (-1, 1, 2)

    # Compute homography using RANSAC
    H, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC, 5.0)

    return H, mask
```

- **Point Extraction**: Converts matched keypoints to coordinate arrays - **RANSAC**: Robust estimation to handle outliers in feature matches - **Homography Matrix**: 3x3 transformation matrix for planar mapping - **Inlier Mask**: Identifies which matches are geometrically consistent

3.3.3 **Image Warping (Lines 67-85)**

```
def warp_image(self, img, H, mode='perspective'):
    h, w = img.shape[:2]

    if mode == 'perspective':
        # Apply homography transformation
        warped = cv2.warpPerspective(img, H, (w, h))
    elif mode == 'affine':
        # Convert to affine transformation
```

```
        H_affine = H[:2, :]
        warped = cv2.warpAffine(img, H_affine, (w, h))

    return warped
```

- **Perspective Warping**: Full homography transformation - **Affine Warping**: Simplified transformation for parallel planes - **Image Dimensions**: Maintains original image size - **Transformation Types**: Different warping modes for different applications

3.4 Step 4: Run the Homography Pipeline

Basic usage:

```
cd ~/ros2_ws
source install/setup.bash

# Run the Homography node
ros2 run homography homography_node
```

Advanced usage with custom parameters:

```
# Run with custom parameters
ros2 run homography homography_node --ros-args \
  -p max_features:=500 \
  -p match_ratio:=0.75 \
  -p ransac_threshold:=5.0 \
  -p warp_mode:=perspective
```

The node will: - Load image pairs from the dataset folder - Compute homography between images - Warp images using the transformation - Display results and transformations (matches view and original vs warped) - **Automatically save results** to the output directory - **Gracefully terminate** after processing is complete

Saved outputs (per image pair, filename-based).

- homography_vis_<img1>_to_<img2>.png — side-by-side with matches
- warped_<img1>_to_<img2>.png — warped second image (into <img2>frame)
- original_vs_warped_<img1>_to_<img2>.png — original vs warped
- H_<img1>_to_<img2>.json — pair homography matrix (JSON)
- H_<img1>_to_<img2>.txt — homography matrix (human-readable)
- H_<img1>_to_<img2>.csv — homography matrix (CSV)

Legacy-style warp filenames.

- <img1>-<img2>.png — image 2 warped into image 1's frame (2→1)
- <img2>-<img1>.png — image 1 warped into image 2's frame (1→2)

Summary outputs.

- homography_summary_<timestamp>.json — list of processed pairs, parameters, and a brief human-readable summary (pairs processed, total inliers, average inlier ratio, output folder)

3.5 Step 5: Visualize the Results

The Homography node will automatically display: - **Original images** side by side - **Matched features** with correspondence lines - **Warped images** showing perspective correction - **Transformation visualization** with grid overlays

Interactive visualization:

- Windows: "Homography: original vs matches (left/right)" and "Original (left) vs Warped (right)"
- Press any key while a window is focused to advance/close (if configured)
- Check terminal output for transformation statistics

3.6 Step 6: Advanced Exercises

3.6.1 **Multi-View Homography**

- **Goal**: Compute homography for multiple image pairs - **Implementation**: Chain homography transformations across image sequences - **Learning**: Understand how to build panoramic views from multiple images

3.6.2 **Homography Decomposition**

- **Goal**: Extract camera motion from homography matrix - **Implementation**: Decompose homography into rotation and translation - **Learning**: Understand the relationship between homography and camera pose

3.6.3 **Robust Homography Estimation**

- **Goal**: Improve homography estimation with better feature matching - **Implementation**: Use advanced feature descriptors and matching strategies - **Learning**: Understand how to handle challenging image pairs

3.6.4 **Real-Time Homography**

- **Goal**: Optimize for real-time camera input - **Implementation**: Use camera streams and optimize computation - **Learning**: Understand performance optimization for live video processing

4 Summary

This tutorial has covered **Homography** implementation:

- **Homography Matrix**: Understanding planar transformations - **Feature Matching**: Finding correspondences between images - **RANSAC Estimation**: Robust homography computation - **Image Warping**: Applying transformations for perspective correction - **ROS2 Integration**: Real-time homography processing

What the node achieves. It loads image pairs, detects and matches SIFT features, estimates a homography using RANSAC, and demonstrates the mapping by warping one image into the other's frame. For each pair, it saves: match visualization, original-versus-warped view, canonical warped images (2→1 and 1→2), and the homography matrix in JSON, TXT, and CSV formats for easy inspection.

5 Next Steps

Continue Learning: - Implement multi-view homography and panorama stitching - Explore homography decomposition for camera pose estimation - Add robust feature matching and outlier rejection - Experiment with different transformation types

Advanced Projects: - Develop visual servoing systems using homography - Implement augmented reality applications - Create panoramic image stitching systems - Build SLAM systems with planar scene understanding