

PRACTICE BOOK

Chapter 7

Visual Odometry and Visual SLAM

Week 7 visual-localisation learning pipeline



Figure 1. The chapter's learning story.

Lei Wang & Xuefei Yin

Griffith University

How to Use This Chapter

Purpose

Use this chapter while doing the Week 7 workshop. It explains the ideas, coding patterns, checkpoints and debugging strategies, but deliberately leaves important implementation decisions for you to complete. Run the experiments, inspect the outputs and write your own observations.

Workshop focus

The supplied Week 7 material moves from a calibrated monocular image sequence to ORB correspondences, the Essential Matrix and relative pose, a qualitative visual-odometry trajectory, and keyframe-based loop-closure verification. Optional relocalisation extends the same ideas.

Learning Outcomes

- Explain why reliable feature correspondences are needed for camera-motion estimation.
- Detect ORB features and descriptors and match binary descriptors with Hamming distance.
- Use a ratio test and RANSAC to reduce ambiguous and geometrically inconsistent matches.
- Explain the role of the Essential Matrix and interpret recovered rotation and translation direction.
- Accumulate relative motions into a qualitative monocular visual-odometry trajectory.
- Explain trajectory drift and unknown monocular scale.
- Use keyframes to propose and geometrically verify a loop-closure candidate.
- Distinguish visual odometry, loop closure and relocalisation.

Part 1 — Inspect the Image Sequence and Calibration

Before doing geometry, verify the data. A VO system depends on correct frame ordering, readable images and the correct camera intrinsics.

1. Check the environment

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

print("OpenCV:", cv2.__version__)
print("NumPy:", np.__version__)
```

2. Find the ordered images

```
from pathlib import Path

DATA_DIR = Path("week7_vo_sequence")
IMAGE_DIR = DATA_DIR / "images"
```

```
paths = sorted(IMAGE_DIR.glob("*.png"))

print("Number of frames:", len(paths))
print("First:", paths[0])
print("Last:", paths[-1])
```

Checkpoint

If paths is empty, stop. Check the working directory, folder name and file extension before debugging any vision algorithm.

3. Read camera parameters

```
import json

with open(DATA_DIR / "camera_parameters.json") as f:
    camera = json.load(f)

fx, fy = camera["fx_px"], camera["fy_px"]
cx, cy = camera["cx_px"], camera["cy_px"]

K = np.array([
    [fx, 0, cx],
    [0, fy, cy],
    [0, 0, 1]
], dtype=np.float64)

print(K)
```

Key idea

K contains the camera's intrinsic parameters. The Essential Matrix stage assumes calibrated image measurements, so use the supplied calibration consistently.

4. Inspect three frames

```
indices = [0, len(paths)//2, len(paths)-1]

fig, axes = plt.subplots(1, 3, figsize=(12, 4))
for ax, i in zip(axes, indices):
    image = cv2.cvtColor(cv2.imread(str(paths[i])),
                        cv2.COLOR_BGR2RGB)
    ax.imshow(image)
    ax.set_title(f"Frame {i}")
    ax.axis("off")
plt.tight_layout()
plt.show()
```

- Identify textured regions, corners and stable scene structures.
- Identify areas likely to be poor for feature matching, such as blank surfaces, reflections or moving objects.
- Choose two consecutive frame pairs for your later experiments: one relatively easy and one more challenging.

Part 2 — ORB Feature Matching

The goal is to establish candidate correspondences: image locations that may represent the same physical scene structure in two frames.

The four-stage pattern

1. Detect local features and compute descriptors.
2. Match descriptors using an appropriate distance metric.
3. Reject ambiguous matches using a ratio test.
4. Convert retained matches into paired 2D coordinates.

1. Detect ORB features

```
orb = cv2.ORB_create(nfeatures=2000)

kp1, des1 = orb.detectAndCompute(gray1, None)
kp2, des2 = orb.detectAndCompute(gray2, None)

print("Keypoints:", len(kp1), len(kp2))
```

Remember

A keypoint is a distinctive image location. Its descriptor is a compact representation used to compare local appearance. ORB descriptors are binary.

2. Match ORB descriptors

```
matcher = cv2.BFMatcher(cv2.NORM_HAMMING)
pairs = matcher.knnMatch(des1, des2, k=2)

print("Candidate pairs:", len(pairs))
```

Hamming distance is appropriate for binary ORB descriptors. The two-neighbour match gives the best and second-best candidates needed by the ratio test.

3. Apply the ratio test

```
RATIO = 0.75

good = []
for pair in pairs:
    if len(pair) < 2:
        continue
    m, n = pair
    if m.distance < RATIO * n.distance:
        good.append(m)

print("Retained:", len(good))
```

Experiment

Try a slightly stricter ratio. Compare both the number and visual quality of matches. A smaller count is not automatically worse if the remaining correspondences are more reliable.

4. Build the point arrays

```
pts1 = np.float32([kp1[m.queryIdx].pt for m in good])
pts2 = np.float32([kp2[m.trainIdx].pt for m in good])

print(pts1.shape, pts2.shape)
```

Shape check

For N matches, both arrays should contain N two-dimensional image coordinates. If the sizes do not agree, do not continue to the geometry stage.

5. Visualise the matches

```
image = cv2.drawMatches(
    gray1, kp1, gray2, kp2, good, None,
    flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS
)

plt.figure(figsize=(12, 5))
plt.imshow(image, cmap="gray")
plt.axis("off")
plt.tight_layout()
plt.show()
```

Discussion prompts

- Why are correct feature correspondences essential for estimating camera motion?
- Why can descriptor matching produce incorrect correspondences?
- Why are static scene structures generally more useful than independently moving objects?

Hint: answer these after inspecting your own match visualisation, not from the number of matches alone.

Part 3 — Essential Matrix and Relative Pose

Now turn the 2D correspondences into a relative camera-motion estimate. This is where appearance information becomes geometric information.

1. Check the precondition

```
if len(good) < 8:
    raise RuntimeError(
        "Not enough correspondences."
    )
```

Engineering habit

Make assumptions explicit. Geometric estimation needs enough observations; clear checks make failures understandable.

2. Estimate the Essential Matrix with RANSAC

```
E, ransac_mask = cv2.findEssentialMat(
    pts1, pts2, K,
    method=cv2.RANSAC,
    prob=0.999,
    threshold=1.0
)

if E is None:
    raise RuntimeError("Essential Matrix estimation failed.")
```

What RANSAC contributes

Some descriptor matches are still outliers. RANSAC seeks a geometric model supported by a consistent subset and returns a mask indicating candidate inliers.

3. Recover relative pose

```
_, R, t, pose_mask = cv2.recoverPose(
    E, pts1, pts2, K,
    mask=ransac_mask.copy()
)

print("R =\n", R)
print("t =\n", t)
```

Critical interpretation

R is a 3×3 rotation matrix. t is a translation-direction estimate. With only a monocular pair, the absolute travelled

distance is not determined by these images alone.

4. Measure the inlier ratio

```
flags = ransac_mask.ravel().astype(bool)

inlier_matches = [
    m for m, keep in zip(good, flags) if keep
]

n_inliers = len(inlier_matches)
ratio = 100.0 * n_inliers / len(good)

print(f"Inliers: {n_inliers}/{len(good)} ({ratio:.1f}%")
```

5. Visualise the geometric inliers

```
image = cv2.drawMatches(
    gray1, kp1, gray2, kp2,
    inlier_matches, None,
    flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS
)

plt.figure(figsize=(12, 5))
plt.imshow(image, cmap="gray")
plt.title("RANSAC inlier correspondences")
plt.axis("off")
plt.tight_layout()
plt.show()
```

Discussion prompts

- What geometric relationship does the Essential Matrix represent?
- Why is RANSAC useful before recovering camera motion?
- Why does monocular `recoverPose()` provide translation direction rather than absolute translation distance?

Part 4 — Build a Qualitative Visual-Odometry Trajectory

Repeat the relative-pose calculation for consecutive frames and accumulate the results into a camera trajectory.

1. Represent R and t together

```
def make_transform(R, t):
    T = np.eye(4)
    T[:3, :3] = R
    T[:3, 3] = t.ravel()
    return T
```

Coordinate-frame warning

Before composing transforms, write down what T_{rel} means and what coordinate frame T_{world} represents. A wrong inversion or multiplication order can produce a reversed, mirrored or implausible trajectory.

2. Build the trajectory

```
T_world = np.eye(4)
positions = [T_world[:3, 3].copy()]

for i in range(len(paths) - 1):
    gray1 = load_frame(paths[i], gray=True)
    gray2 = load_frame(paths[i + 1], gray=True)

    R, t, *_ = estimate_motion(gray1, gray2, K)
    T_rel = make_transform(R, t)

    # TODO:
    # Update T_world using one consistent transform convention.

    positions.append(T_world[:3, 3].copy())

positions = np.asarray(positions)
```

3. Plot X-Z

```
plt.figure(figsize=(7, 6))
plt.plot(positions[:, 0], positions[:, 2],
         "-o", markersize=2)

plt.scatter(positions[0, 0], positions[0, 2],
            marker="s", s=80, label="Start")
plt.scatter(positions[-1, 0], positions[-1, 2],
            marker="x", s=80, label="Final")

plt.xlabel("X (arbitrary units)")
plt.ylabel("Z (arbitrary units)")
plt.title("Monocular Visual-Odometry Trajectory")
plt.axis("equal")
```

```
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.savefig("week7_vo_trajectory.png", dpi=150)
plt.show()
```

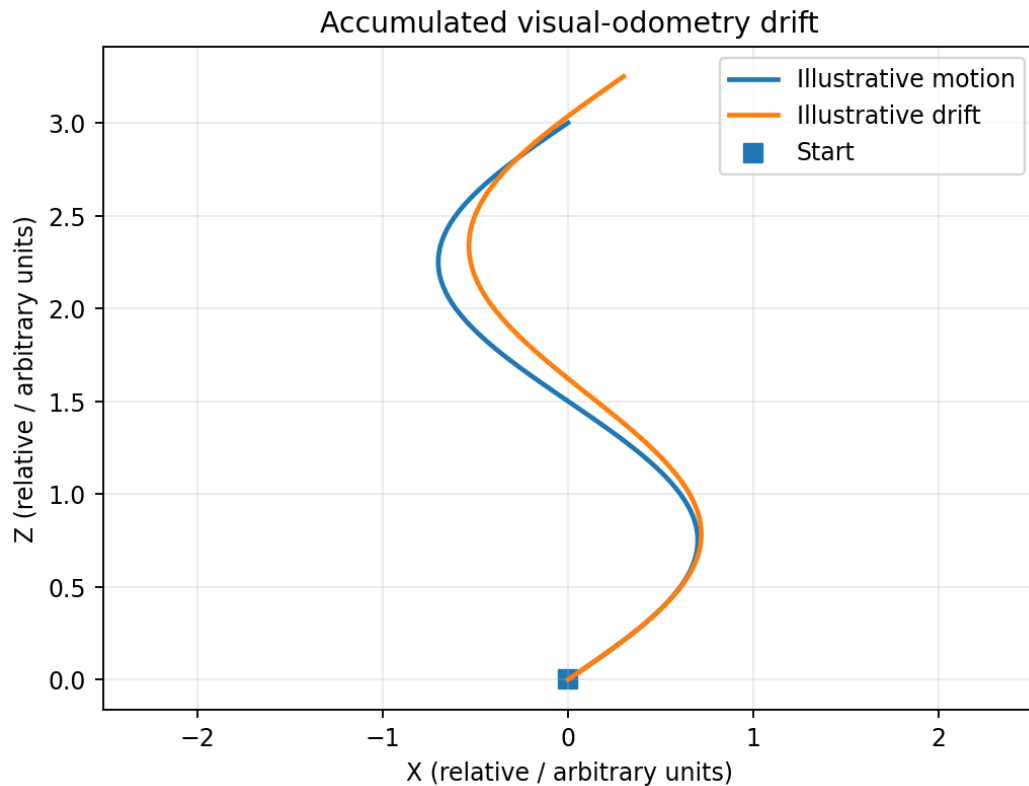


Figure 2. Conceptual example of accumulated trajectory drift; this is an illustration, not a supplied-sequence result.

Scale warning

The trajectory is qualitative. Treat the recovered translations as arbitrary relative units. Do not describe the plotted distance as metres.

4. Diagnose the trajectory

- Look for sudden jumps and compare them with the corresponding frame pair.
- Check the number and percentage of geometric inliers for suspicious pairs.
- Ask whether a difficult image pair could explain the behaviour before changing the code.
- Remember that repeated composition of imperfect local estimates naturally produces drift.

Discussion prompts

- Why does visual odometry accumulate error over time?
- Why is the absolute scale of a monocular trajectory unknown?
- What additional sensors or information could provide metric scale?

Part 5 — Keyframes and Simple Loop Closure

Local visual odometry mainly connects nearby frames. Keyframes introduce persistent visual references that can be revisited later.

Loop closure: recognise, then verify

descriptor similarity → geometric verification → constraint

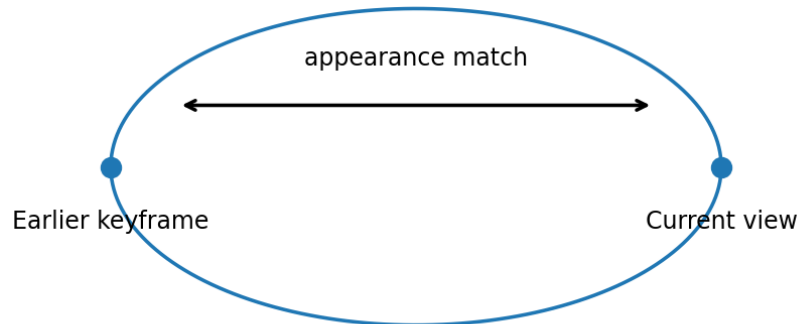


Figure 3. Loop closure concept: propose a previous view using appearance, then verify it geometrically.

1. Select keyframes

```
def keyframe_indices(n_frames, interval=10):  
    return list(range(0, n_frames - 1, interval))  
  
keyframes = keyframe_indices(  
    len(paths), interval=10  
)  
  
print(keyframes)
```

Experiment

Try a different interval. More keyframes increase the number of possible references and comparisons; fewer keyframes reduce computation but may miss useful views.

2. Propose a candidate by descriptor matches

```
candidates = []  
  
for idx in keyframes:  
    key = load_frame(paths[idx], gray=True)  
  
    _, _, matches, _, _ = match_frames(  
        key, current_key, descriptor_extractor, descriptor_extractor, descriptor_extractor)
```

```

        current, key
    )

    candidates.append((len(matches), idx))

candidates.sort(reverse=True)

best_idx = candidates[0][1]
print("Candidate ranking:", candidates)
print("Proposed keyframe:", best_idx)

```

Important

This is a proposal, not yet a confirmed loop closure. Appearance similarity can be misleading.

3. Geometrically verify the candidate

```

kp1, kp2, matches, pts1, pts2 = match_frames(
    current, best
)

if len(matches) >= 8:
    F, mask = cv2.findFundamentalMat(
        pts1, pts2,
        cv2.FM_RANSAC,
        1.0, 0.99
    )

    # TODO:
    # Turn mask into boolean flags.
    # Keep only inlier matches.
    # Count the inliers.
else:
    print("Not enough matches for verification.")

```

Core idea

A useful loop-closure test combines appearance and geometry: descriptor matching proposes a place; RANSAC checks whether the correspondences are geometrically consistent.

4. Visualise the verified matches

```

image = cv2.drawMatches(
    current, kp1, best, kp2,
    inlier_matches, None,
    flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS
)

plt.figure(figsize=(12, 5))

```

```
plt.imshow(image, cmap="gray")
plt.title("Verified loop-closure candidate")
plt.axis("off")
plt.tight_layout()
plt.show()
```

Discussion prompts

- Why does a SLAM system use keyframes instead of treating every frame equally?
- Why is feature similarity alone insufficient to confirm a loop closure?
- How can loop closure help reduce accumulated visual-odometry drift?

Optional Part 6 — Relocalisation After Tracking Loss

Relocalisation is a recovery idea: when local frame-to-frame tracking is no longer trusted, use persistent keyframes as visual references.

1. Choose a frame where you will assume local tracking has failed.
2. Compare it against the stored keyframes.
3. Use descriptor matching to generate candidates.
4. Use geometric verification to score candidates.
5. Select the strongest verified candidate and visualise the result.

```
best_keyframe = None
best_inliers = -1

for idx in keyframes:
    key = load_frame(paths[idx], gray=True)

    result = verify_candidate(
        current, key
    )

    n_inliers = result[-1]

    if n_inliers > best_inliers:
        best_inliers = n_inliers
        best_keyframe = idx

print("Best keyframe:", best_keyframe)
print("Verified inliers:", best_inliers)
```

Terminology check

Visual odometry estimates local motion from nearby frames. Loop closure recognises a previously visited place. Relocalisation uses persistent visual references to recover location after local tracking has been lost.

Reflection: Explain, in your own words, why matching the current frame with a previous keyframe is different from simply matching it with the immediately previous frame.

Supplementary Material

	Fundamental Matrix $\mathbf{F}$	Essential Matrix $\mathbf{E}$
Represents	Epipolar geometry	Calibrated epipolar geometry
Requires camera calibration?	No	Yes
Uses camera intrinsics $\mathbf{K}$?	No	Yes
Can help reject bad matches?	✓	✓
Used for camera pose estimation?	Indirectly	Yes
Typical function	<code>findFundamentalMat()</code>	<code>findEssentialMat()</code>

Use this section during the workshop if students encounter failures. Encourage them to diagnose the stage rather than immediately changing random parameters.

Symptom	Likely cause	First thing to inspect
Very few keypoints	Low texture / unsuitable image	Visualise the frame and inspect textured regions
Very few retained matches	Large motion, blur, weak features, ambiguous descriptors	Compare keypoint counts and match image
Enough matches but few RANSAC inliers	Many descriptor outliers or poor geometry	Inspect ratio test and inlier visualisation
Essential Matrix is None	Insufficient or unusable correspondences	Check number and quality of matches
Trajectory jumps	Unstable pairwise pose estimate	Inspect inlier count and problematic frame pair
Trajectory drifts	Accumulated local estimation errors	Explain composition of frame-to-frame errors
False loop candidate	Repeated visual appearance	Require geometric verification
Student reports metres	Misunderstanding monocular scale	Revisit <code>recoverPose</code> translation scale limitation

Putting the Pipeline Together

1. Inspect the sequence and calibration.
2. Detect ORB features.
3. Match binary descriptors.
4. Reject ambiguous matches with the ratio test.
5. Use RANSAC-based geometry to reject outliers.
6. Recover relative rotation and translation direction.
7. Accumulate local motions into a qualitative trajectory.
8. Store selected keyframes.
9. Propose a loop candidate from visual similarity.
10. Geometrically verify the candidate.

Best debugging strategy

If the final trajectory is wrong, move backwards through the evidence: trajectory → relative pose → inliers → matches → keypoints. Do not immediately rewrite the last stage.

Suggested file structure

```
week7_visual_localisation/  
|  
|-- week7_vo_sequence/  
|   |-- images/  
|   |-- camera_parameters.json  
|  
|-- part1_sequence.py  
|-- part2_feature_matching.py  
|-- part3_relative_pose.py  
|-- part4_visual_odometry.py  
|-- part5_loop_closure.py  
|-- part6_relocalisation.py    # optional  
|  
|-- week7_vo_trajectory.png  
|-- your_loop_closure_result.png
```

Minimum self-check

- I can explain the difference between a keypoint and a descriptor.
- I can explain why Hamming distance is used for ORB.
- I can explain what the ratio test rejects.
- I can explain what a RANSAC inlier represents.
- I can interpret R and t from recoverPose().
- I can explain unknown monocular scale.
- I can explain trajectory drift.
- I can explain why keyframes are useful.
- I can explain why loop closure requires geometric verification.
- I can distinguish VO, loop closure and relocalisation.

Key Code Parts You Should Be Able to Explain

Concept	You should be able to explain
ORB creation	What nfeatures controls and why more features do not automatically mean better tracking.
BFMatcher + Hamming	Why the descriptor type determines the useful distance measure.
Ratio test	Why comparing the best and second-best candidate helps reject ambiguity.
findEssentialMat	How calibrated two-view geometry and RANSAC are used together.
recoverPose	What rotation and translation direction mean, and what monocular scale does not provide.
4×4 transform	How rotation and translation are represented and composed.
Trajectory accumulation	Why local estimation errors can become global drift.
Keyframes	Why selected persistent views support place recognition.
Fundamental-matrix RANSAC	Why appearance similarity must be checked geometrically.
Relocalisation	Why persistent keyframes can recover from loss of local tracking.

Final Takeaway

Good visual localisation is a chain of evidence.

Features → correspondences → geometric verification → motion → trajectory → persistent references

Remember

A program that runs is not automatically a reliable localisation system. Check intermediate results, state your assumptions, understand coordinate conventions, and know the limitations of monocular geometry.