

**GRIFFITH UNIVERSITY**

**3006ICT Robotics and Computer Vision**

# **PRACTICE BOOK**

## **Chapter 3**

### **Tracking Visual Features Across Video Frames**

*From code that runs to perception you can trust*

Dr Lei Wang

Griffith University

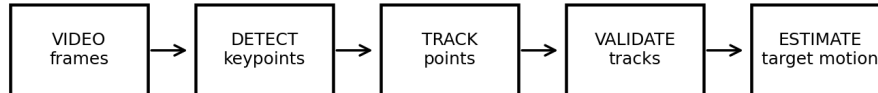
#### **How to use this chapter**

Use this chapter beside the Week 3 workshop. Read the concept first, copy the short code block into your notebook, run it, inspect the result, and then complete the mini-experiment. The goal is not to memorise OpenCV calls. The goal is to learn how to build, inspect, validate and stress-test a visual perception system.

## What This Chapter Is About

A single image tells us what is visible at one instant. A video gives us an ordered sequence of observations, which allows us to reason about how visual structure changes over time. In this chapter you will build a classical visual-tracking pipeline using OpenCV and sparse Lucas-Kanade optical flow.

### Workshop 3 perception pipeline



### The central engineering question

Do not stop at: “Does the tracker run?” Ask: “What is the tracker actually measuring, what assumptions does it make, how can I test those assumptions, and when should I stop trusting its output?”

### Learning outcomes

- Inspect a video and understand why frame order and frame rate matter.
- Select a task-relevant region of interest (ROI) and detect distinctive keypoints.
- Track keypoints between consecutive frames using sparse Lucas-Kanade optical flow.
- Interpret an optical-flow vector as apparent image-plane displacement.
- Aggregate many point motions into a robust target-motion estimate using the median.
- Validate tracks using status and tracking error, and recover by re-detecting features.
- Diagnose common failure modes instead of treating a plausible visual result as automatically correct.
- Use a repeatable perception-engineering workflow: RUN -> OBSERVE -> EXPLAIN -> STRESS-TEST -> DIAGNOSE -> IMPROVE.

### How Week 3 differs from Weeks 1 and 2

| Week   | Main engineering question                                                         |
|--------|-----------------------------------------------------------------------------------|
| Week 1 | How do I build a solution carefully from the problem, algorithm and tests?        |
| Week 2 | How can AI accelerate implementation while I retain engineering judgement?        |
| Week 3 | When can I trust a perception result, and what happens when its assumptions fail? |

## Setup and Working Style

The official workshop uses the Week 1 Python environment and the supplied video files. The core implementation uses OpenCV and NumPy. The examples below are written so that you can copy, paste and modify small pieces rather than trying to understand a large program all at once.

### Recommended files

| File                             | Purpose                                                                |
|----------------------------------|------------------------------------------------------------------------|
| 01_textured_target.mp4           | Main tracking example used in Parts 1-5.                               |
| part1_load_video.py              | Load the video and inspect the first frame.                            |
| part2_detect_keypoints.py        | Select an ROI and detect target features.                              |
| part3_sparse_optical_flow.py     | Track features frame-to-frame.                                         |
| part4_motion_and_trajectories.py | Visualise trajectories and estimate motion.                            |
| part5_validate_and_redetect.py   | Reject unreliable tracks and recover lost features.                    |
| Part 6 failure videos (optional) | Stress-test low texture, repeated patterns, occlusion and fast motion. |

### A useful rule for every experiment

#### **Predict -> Change one thing -> Run -> Observe -> Explain**

If you change five parameters at once, you cannot tell which change caused the result. Small controlled experiments teach you much more about an algorithm.

## Part 1 - Load and Inspect a Video

Before applying a tracking algorithm, inspect the input. This is simple defensive engineering: if the video cannot be read, there is no reason to debug optical flow.

### Key code

```
import cv2

VIDEO = "01_textured_target.mp4"

cap = cv2.VideoCapture(VIDEO)
ok, frame = cap.read()

if not ok:
    cap.release()
    raise FileNotFoundError(f"Cannot read {VIDEO}")

fps = cap.get(cv2.CAP_PROP_FPS)
height, width = frame.shape[:2]

print("Frame size:", width, "x", height)
print("FPS:", fps)

cv2.imshow("First frame", frame)
cv2.waitKey(0)
cv2.destroyAllWindows()

cap.release()
```

### What the important lines mean

| Line / function       | Why it matters                                                         |
|-----------------------|------------------------------------------------------------------------|
| cv2.VideoCapture(...) | Opens the video source and gives access to its frames.                 |
| cap.read()            | Returns both a success flag and the next frame. Always check the flag. |
| frame.shape[:2]       | Gives image height and width; useful for later ROI and boundary logic. |
| CAP_PROP_FPS          | Tells you how densely the scene is sampled in time.                    |
| cap.release()         | Releases the video resource when finished.                             |

#### Insight: video adds time

A frame is one observation. Tracking uses pairs of observations: previous frame -> current frame. The order matters because motion is defined between times.

### Practice

1. Print the number of channels in the frame. What do you expect for a normal colour OpenCV frame?
2. Print the FPS. If the video has 25 FPS, approximately how much time passes between consecutive frames?
3. Try to read a deliberately incorrect filename. What does the success flag tell you?

### Discussion question

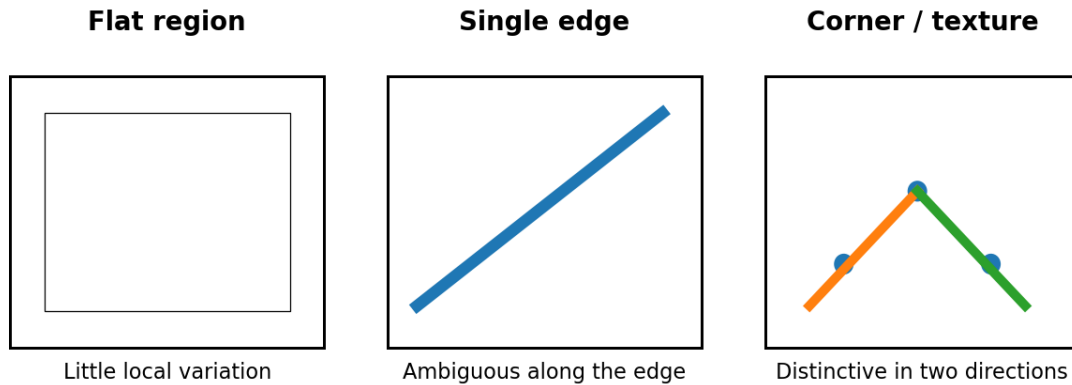
**What additional information does a video provide compared with a single image?**

**Answer.** A video provides temporal information: an ordered sequence of frames. This lets us estimate how image features move or change between observations.

**Engineering insight:** Temporal information does not automatically give 3D motion. It gives observations from which motion can be estimated.

## Part 2 - Select the Target and Detect Keypoints

Tracking every pixel is unnecessary. Instead, we detect a small set of distinctive image locations that can be recognised again in the next frame. The workshop uses Shi-Tomasi features through OpenCV's `goodFeaturesToTrack` function.



### Why corners?

A flat region contains little local information. A single edge is ambiguous because a point can slide along the edge. A corner or textured point changes strongly in two directions, making it more distinctive.

### Key code

```
def detect_keypoints(frame, roi):
    x, y, w, h = map(int, roi)

    gray = cv2.cvtColor(
        frame, cv2.COLOR_BGR2GRAY
    )

    mask = np.zeros_like(gray)
    mask[y:y + h, x:x + w] = 255

    points = cv2.goodFeaturesToTrack(
        gray,
        mask=mask,
        maxCorners=100,
        qualityLevel=0.3,
        minDistance=7
    )

    return points
```

### Understand the ROI mask

The mask is a simple but important piece of task knowledge. A value of 255 means feature detection is allowed; a value of 0 means the region is ignored. Without the mask, a visually strong corner in the background could be selected even though the robot cares about the target.

```
mask = np.zeros_like(gray)
mask[y:y + h, x:x + w] = 255
```

### Key parameters to experiment with

| Parameter | Meaning | Try this |
|-----------|---------|----------|
|-----------|---------|----------|

|              |                                          |                      |
|--------------|------------------------------------------|----------------------|
| maxCorners   | Maximum number of returned points.       | 100 -> 20            |
| qualityLevel | Minimum relative corner strength.        | 0.30 -> 0.10 or 0.50 |
| minDistance  | Minimum spacing between returned points. | 7 -> 20 -> 30        |

### Important: more points are not automatically better

A large number of weak, clustered or ambiguous points can be worse than a smaller set of distinctive, well-distributed points.

## Mini experiment: minDistance

```
# Run A
minDistance=7

# Run B
minDistance=30
```

Before running, predict what should happen. Then compare the number and spatial distribution of keypoints.

## Visualisation detail: floating-point coordinates

```
for u, v in points.reshape(-1, 2):
    cv2.circle(
        result,
        (round(u), round(v)),
        4,
        (0, 0, 255),
        -1
    )
```

Keep the feature coordinates as floating-point values for computation. Round only when a drawing function requires integer pixel coordinates. This preserves numerical information for later tracking and motion estimation.

## Discussion questions and answers

### Why are corners usually more suitable for tracking than flat regions or single edges?

**Answer.** Corners provide distinctive intensity changes in two image directions. A flat region has little local information, while a point on a single edge can be ambiguous along the edge.

### What happens if minDistance is increased?

**Answer.** Fewer points are usually returned and the selected points are more widely separated because nearby candidate corners are suppressed.

### Why restrict feature detection to the target ROI?

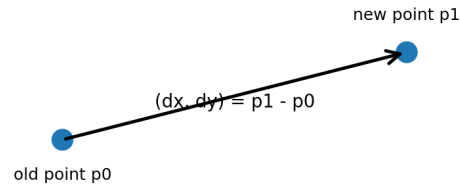
**Answer.** It makes the detector task-relevant and reduces the chance of selecting strong but irrelevant background features.

**Engineering insight:** A perception algorithm does not automatically know what the robot cares about. The ROI injects task knowledge.

## Part 3 - Track Keypoints with Sparse Optical Flow

Part 2 answered: 'Where are useful features?' Part 3 asks: 'Where did those same features go in the next frame?' The workshop uses pyramidal Lucas-Kanade sparse optical flow.

**One optical-flow vector: previous position -> current position**



This is image-plane displacement, not automatically 3D/world motion.

### Core code

```
new_gray = cv2.cvtColor(
    frame, cv2.COLOR_BGR2GRAY
)

p1, status, error = cv2.calcOpticalFlowPyrLK(
    old_gray,
    new_gray,
    p0,
    None
)

valid = status.ravel() == 1

good_new = p1[valid]
good_old = p0[valid]
```

### Read the variables as a story

| Variable | Interpretation                                               |
|----------|--------------------------------------------------------------|
| old_gray | The previous image at time t.                                |
| new_gray | The current image at time t+1.                               |
| p0       | Known feature locations in the previous image.               |
| p1       | Estimated locations of those features in the current image.  |
| status   | OpenCV's success/failure indicator for each track.           |
| error    | Tracking error measure returned by the optical-flow routine. |

### Optical flow is image-plane motion

A vector  $p1 - p0$  describes apparent pixel displacement between frames. It is not automatically the physical displacement of the object in metres or the motion of the object in the world.

### Visualise one vector

```
for new, old in zip(
    good_new.reshape(-1, 2),
    good_old.reshape(-1, 2)):
    # Plot vector from old to new
```

```

a, b = np.round(new).astype(int)
c, d = np.round(old).astype(int)

cv2.line(
    frame, (c, d), (a, b),
    (0, 255, 0), 2
)

cv2.circle(
    frame, (a, b),
    3, (0, 0, 255), -1
)

```

## The two most important temporal-update lines

```

old_gray = new_gray
p0 = good_new.reshape(-1, 1, 2)

```

### Memory rule: 'today becomes yesterday'

After processing the current frame, it becomes the reference frame for the next iteration. Likewise, the newly tracked points become the starting points for the next step.

## A useful deliberate bug

Temporarily comment out the two state-update lines above and run the tracker. The program may still execute, but the temporal reasoning is now wrong because you keep comparing against stale state. Restore the lines after the experiment.

## Discussion questions and answers

### What does an optical-flow vector represent?

**Answer.** The apparent pixel displacement of an image feature between two consecutive frames.

### Why is optical flow not always equal to the true physical motion of the target?

**Answer.** Image motion can include camera motion, perspective effects, depth changes, local deformation and tracking error. Therefore pixel motion is not automatically world-space motion.

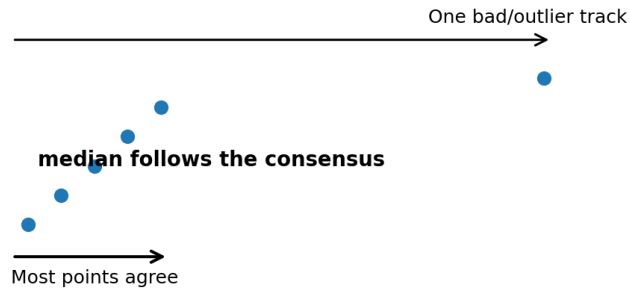
### What does status = 0 indicate?

**Answer.** OpenCV could not reliably track that feature into the new frame. The point should be excluded from later motion estimation.

## Part 4 - Visualise Trajectories and Estimate Target Motion

We now move from individual measurements to a compact estimate. A robot controller often does not need every feature location; it may need a stable estimate such as 'the target moved right by approximately 3 pixels this frame'.

### Why use the median across many tracks?



### Robust aggregation

```
displacements = good_new - good_old

dx, dy = np.median(
    displacements,
    axis=0
)

cx, cy = np.median(
    good_new,
    axis=0
)
```

| Quantity            | Interpretation                                                       |
|---------------------|----------------------------------------------------------------------|
| good_new - good_old | One displacement vector per surviving feature.                       |
| median(..., axis=0) | Robustly combines many vectors into one representative displacement. |
| cx, cy              | Approximate target centre based on the current feature cloud.        |

### Why median rather than one point?

One feature can be wrong. A median is relatively resistant to a small number of outliers, so it is a sensible baseline estimator for a collection of noisy point motions.

### Motion labels and a deadband

```
def motion_label(dx, threshold=2):
    if dx < -threshold:
        return "MOVING LEFT"
    if dx > threshold:
        return "MOVING RIGHT"
    return "APPROXIMATELY STATIONARY"
```

The threshold creates a small deadband around zero. Without it, tiny numerical or visual fluctuations could cause the system to alternate between LEFT and RIGHT even when the target is effectively stationary.

### Persistent trajectories

```
# Create once, before the tracking loop.
```

```

trails = np.zeros_like(old_frame)

# Inside the loop:
for new, old in zip(
    good_new.reshape(-1, 2),
    good_old.reshape(-1, 2)):

    a, b = np.round(new).astype(int)
    c, d = np.round(old).astype(int)

    cv2.line(
        trails, (c, d), (a, b),
        (0, 255, 0), 2
    )

# Overlay the persistent trails.
frame = cv2.add(frame, trails)

```

### Perception abstraction

The pipeline is progressively transforming information: pixels -> keypoints -> correspondences -> flow vectors -> target-motion estimate. This pattern appears throughout robotics perception.

## Discussion questions and answers

### Why is the median displacement often more robust than using one point?

**Answer.** A single point can be a bad correspondence. The median uses many measurements and is less sensitive to a small number of outliers.

### Why should the result be called image-plane motion rather than world-space motion?

**Answer.** The estimate is expressed in pixels. Without camera geometry, depth and appropriate motion assumptions, pixel displacement cannot be interpreted as metric 3D displacement.

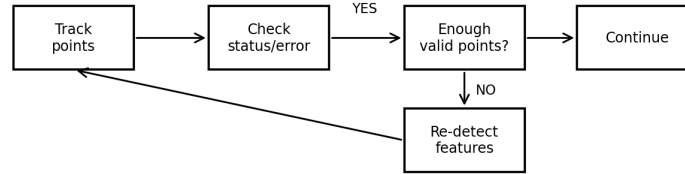
### Why can different points on the same target have slightly different flow vectors?

**Answer.** Perspective, depth differences, local deformation, lighting changes, noise and tracking errors can make the apparent motion differ across the target.

## Part 5 - Validate Tracks and Re-detect Lost Features

This is the most important engineering upgrade in the workshop. A tracker can produce a numerical answer even when the answer is no longer trustworthy. We therefore add explicit validation and a recovery mechanism.

### Reliable perception needs a recovery mechanism



### Two-stage validation

```

valid = (
    (status.ravel() == 1) &
    (error.ravel() < MAX_ERROR)
)

good_new = p1[valid].reshape(-1, 2)
good_old = p0[valid].reshape(-1, 2)
  
```

| Check                             | Meaning                                                                         |
|-----------------------------------|---------------------------------------------------------------------------------|
| <code>status == 1</code>          | OpenCV reports that the feature was successfully tracked.                       |
| <code>error &lt; MAX_ERROR</code> | The tracking result is sufficiently consistent with the chosen error criterion. |
| Both checks                       | Only the points that pass both reliability gates are used downstream.           |

### Track health and recovery

```

MIN_POINTS = 15
MAX_ERROR = 20

if len(good_new):
    dx, dy = np.median(
        good_new - good_old,
        axis=0
    )
    roi = move_roi(
        roi, dx, dy, frame.shape
    )

if len(good_new) < MIN_POINTS:
    p0 = detect(new_gray, roi)
else:
    p0 = good_new.reshape(-1, 1, 2)
  
```

### Why move the ROI?

The initial ROI describes where the target was at the beginning. If the target moves, keeping the ROI fixed means that re-detection will eventually search the wrong place. The robust motion estimate is therefore used to shift the ROI.

### Why re-detect?

Tracking propagates an existing representation. Re-detection refreshes that representation using the current image. This is a simple recovery mechanism: when evidence becomes too weak, do not keep propagating it indefinitely.

**Trade-off: recovery introduces a new risk**

Re-detection can recover useful features, but it may also select background points or features from another object. An engineering improvement can introduce a new failure mode, so recovery itself must be validated.

**Discussion questions and answers****Why can tracked points disappear even when the target is still visible?**

**Answer.** Occlusion, blur, illumination changes, movement outside the frame, weak texture or tracking uncertainty can make individual correspondences unavailable.

**Why is re-detection useful, and what risk does it introduce?**

**Answer.** It refreshes the feature set when too few reliable points remain. The risk is that newly detected points may belong to the background or a different object.

**How does validation improve temporal consistency?**

**Answer.** It prevents unreliable point estimates from being propagated into later frames, reducing the chance that one bad correspondence contaminates subsequent motion estimates.

## The Reusable Tracking Skeleton

The following skeleton is deliberately compact. It is a reference pattern you can reuse in later robotics projects. The important structure is more valuable than memorising every OpenCV argument.

```
# 1. Open video
cap = cv2.VideoCapture(VIDEO)
ok, frame = cap.read()
if not ok:
    raise FileNotFoundError(VIDEO)

# 2. Initialise target + features
roi = cv2.selectROI("Select target", frame, False, False)
old_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
p0 = detect(old_gray, roi)

# 3. Process one frame at a time
while True:
    ok, frame = cap.read()
    if not ok:
        break

    new_gray = cv2.cvtColor(
        frame, cv2.COLOR_BGR2GRAY
    )

    # 4. Estimate correspondences
    p1, status, error = cv2.calcOpticalFlowPyrLK(
        old_gray, new_gray, p0, None
    )

    # 5. Validate
    valid = (
        (status.ravel() == 1) &
        (error.ravel() < MAX_ERROR)
    )

    good_new = p1[valid]
    good_old = p0[valid]

    # 6. Estimate motion
    if len(good_new) > 0:
        dx, dy = np.median(
            good_new - good_old,
            axis=0
        )

    # 7. Recover if evidence is weak
    if len(good_new) < MIN_POINTS:
        p0 = detect(new_gray, roi)
    else:
        p0 = good_new.reshape(-1, 1, 2)

    # 8. Advance temporal state
    old_gray = new_gray

cap.release()
```

### Reusable architecture

INITIALISE -> ESTIMATE -> VALIDATE -> AGGREGATE -> DECIDE/RECOVER -> UPDATE STATE. This structure is useful far beyond optical flow.

## Where this pattern appears later

- Object tracking: detect an object, track it, validate the estimate, and re-detect when confidence drops.
- Visual odometry: estimate motion, reject bad correspondences, aggregate many measurements, and monitor quality.
- Robot localisation: estimate position, check residuals, reject inconsistent observations, and recover from loss.
- Sensor fusion: combine measurements, quantify reliability, and avoid letting one faulty sensor dominate the estimate.

## Debugging Guide - What Should I Check First?

| Symptom                             | First things to check                                              |
|-------------------------------------|--------------------------------------------------------------------|
| No video / first frame is empty     | File path, cap.read() success flag, video codec.                   |
| No keypoints                        | ROI, grayscale conversion, texture, qualityLevel, minDistance.     |
| Keypoints mostly on background      | ROI mask is wrong or too large.                                    |
| Tracks disappear quickly            | Texture, blur, illumination, motion size, status and error.        |
| Trajectories jump                   | Bad correspondences, insufficient validation, repeated patterns.   |
| Motion direction is noisy           | Too few points, outliers, threshold too small, target deformation. |
| ROI stays behind target             | Motion estimate not used to update ROI.                            |
| Re-detection locks onto background  | ROI is too large or recovery needs stronger validation.            |
| Output looks plausible but is wrong | Inspect raw tracks, count valid points, test controlled cases.     |

### A practical debugging sequence

1. Check the input: can the video be opened and read?
2. Check the first frame: is the target actually visible?
3. Check the ROI: is it around the target rather than the background?
4. Check initial features: are there enough distinctive points?
5. Check one frame transition: do the vectors make visual sense?
6. Check track count: is evidence gradually disappearing?
7. Check validation: are bad points being rejected?
8. Check recovery: does re-detection occur when expected?
9. Only then tune parameters.

#### Do not debug by changing everything

If the tracker fails, do not immediately change ROI, qualityLevel, minDistance, LK settings and thresholds together. Isolate the failure first.

## Optional Extension - Failure Analysis

Part 6 of the official workshop is optional and not marking. It is nevertheless an excellent engineering exercise because it deliberately challenges the assumptions of the baseline tracker.

| Condition        | Typical observation                            | Underlying issue                         | Possible direction                                               |
|------------------|------------------------------------------------|------------------------------------------|------------------------------------------------------------------|
| Low texture      | Few or unstable points                         | Not enough distinctive local information | Cautious detector tuning or another sensor.                      |
| Repeated pattern | Tracks can attach to similar-looking locations | Correspondence ambiguity                 | Appearance or geometric validation.                              |
| Occlusion        | Tracks disappear                               | Target is temporarily unobservable       | Re-detect after visibility returns; use other sensing if needed. |
| Fast motion      | Abrupt track loss, blur                        | Frame-to-frame displacement too large    | Higher frame rate or more suitable tracking settings.            |

### Failure-analysis method

1. OBSERVE: What exactly do the tracks do?
2. DIAGNOSE: Which assumption or source of information is failing?
3. IMPROVE: What parameter, validation rule, recovery method or sensor change could help?

#### Key principle

No parameter can recover visual information that is not currently visible. Good engineering includes knowing when an algorithm's assumptions have been violated.

## Consolidated Discussion Questions - Revision Answers

### Part 1: What does a video add beyond a single image?

**Answer.** Temporal ordering. It provides consecutive observations from which apparent motion can be estimated.

### Part 2: Why are corners useful for tracking?

**Answer.** They are locally distinctive in more than one image direction, making correspondence less ambiguous.

### Part 2: What does minDistance do?

**Answer.** It controls the spatial separation between selected features. Increasing it usually reduces clustering and the number of returned points.

### Part 2: Why use an ROI?

**Answer.** To make perception task-relevant and reduce irrelevant background features.

### Part 3: What is an optical-flow vector?

**Answer.** The apparent pixel displacement of a tracked feature between two frames.

### Part 3: Why is optical flow not automatically physical motion?

**Answer.** Camera motion, perspective, depth, deformation and errors can all affect image motion.

### Part 3: What does status = 0 mean?

**Answer.** That feature should not be trusted for the current correspondence.

### Part 4: Why median displacement?

**Answer.** It combines many measurements while being relatively robust to a small number of outliers.

### Part 4: Why image-plane motion?

**Answer.** The estimate is in pixel coordinates; metric world motion requires additional geometry and assumptions.

### Part 5: Why validate with both status and error?

**Answer.** The status flag is a first gate, while the error threshold adds a second check against unreliable tracks.

### Part 5: Why re-detect?

**Answer.** To refresh the feature representation when the surviving evidence becomes too weak.

### Part 5: What is the risk of re-detection?

**Answer.** New features may belong to the background or another object, especially if the ROI is inaccurate.

## Hands-on Mini Experiments

### Experiment A - Feature density

Keep the video and ROI fixed. Change only minDistance.

| Setting | Prediction: number of points | Prediction: spacing | Observed |
|---------|------------------------------|---------------------|----------|
| 7       | _____                        | _____               | _____    |
| 20      | _____                        | _____               | _____    |
| 30      | _____                        | _____               | _____    |

### Experiment B - Feature quality

Keep minDistance fixed. Compare qualityLevel values such as 0.10, 0.30 and 0.50.

| qualityLevel | Number of points | Visual quality | Comments |
|--------------|------------------|----------------|----------|
| 0.10         | _____            | _____          | _____    |
| 0.30         | _____            | _____          | _____    |
| 0.50         | _____            | _____          | _____    |

### Experiment C - Track health

Run the validated tracker and record how many valid points survive over time.

| Frame | Valid tracks | Median dx | Median dy | Trust? Why? |
|-------|--------------|-----------|-----------|-------------|
| 0     | _____        | _____     | _____     | _____       |
| 10    | _____        | _____     | _____     | _____       |
| 20    | _____        | _____     | _____     | _____       |
| 30    | _____        | _____     | _____     | _____       |
| 40    | _____        | _____     | _____     | _____       |

### Experiment D - Stress one assumption

Choose one: crop the target so it has less texture; increase the target speed; temporarily cover part of it; or choose a repeated pattern. Describe what changed.

- Assumption I changed: \_\_\_\_\_
- What I predicted: \_\_\_\_\_
- What I observed: \_\_\_\_\_
- Likely reason: \_\_\_\_\_
- One improvement: \_\_\_\_\_

## Using AI as an Engineering Assistant

AI can be useful for explaining OpenCV functions, generating small code blocks and suggesting tests. It should not be used as a substitute for understanding the tracking pipeline.

### Useful prompts for this chapter

Explain `cv2.goodFeaturesToTrack` in plain English and explain what `maxCorners`, `qualityLevel` and `minDistance` change.

Review my optical-flow code and identify logical errors without rewriting the whole program.

Explain what `p0`, `p1`, `status` and `error` represent in `cv2.calcOpticalFlowPyrLK`.

Suggest three normal tests, three boundary tests and three failure tests for my feature tracker.

My tracks jump to the background. Give me a debugging checklist before suggesting code changes.

Explain why median displacement is more robust than using one tracked point.

Review my re-detection strategy and identify situations where it could select the wrong object.

#### Good AI habit

Ask AI to explain, review, test and diagnose your code before asking it to replace your code. Your responsibility remains: understand the assumptions and verify the result.

## Chapter 3 Checklist

| Skill                            | I can explain / do this                                                                                                   |
|----------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| Video input                      | I can open a video, check <code>cap.read()</code> , inspect frame size and FPS.                                           |
| Temporal reasoning               | I understand previous frame vs current frame.                                                                             |
| ROI                              | I can explain why task-relevant regions matter.                                                                           |
| Keypoints                        | I can explain why distinctive corners are useful.                                                                         |
| <code>goodFeaturesToTrack</code> | I know the roles of <code>maxCorners</code> , <code>qualityLevel</code> and <code>minDistance</code> .                    |
| Optical flow                     | I can explain <code>p0</code> -> <code>p1</code> and status.                                                              |
| Image-plane motion               | I do not confuse pixels with 3D/world motion.                                                                             |
| Temporal state                   | I understand why <code>old_gray</code> = <code>new_gray</code> and <code>p0</code> = <code>good_new</code> are essential. |
| Aggregation                      | I can explain median displacement.                                                                                        |
| Validation                       | I can explain status and error filtering.                                                                                 |
| Recovery                         | I can explain when and why re-detection occurs.                                                                           |
| ROI update                       | I understand why the ROI should follow the target.                                                                        |
| Failure analysis                 | I can use OBSERVE -> DIAGNOSE -> IMPROVE.                                                                                 |
| Engineering judgement            | I know when a result should no longer be trusted.                                                                         |

## Key functions to remember

| Function / operation                                          | Purpose                                           |
|---------------------------------------------------------------|---------------------------------------------------|
| <code>cv2.VideoCapture</code>                                 | Open a video source.                              |
| <code>cap.read()</code>                                       | Read the next frame and success flag.             |
| <code>cv2.cvtColor</code>                                     | Convert BGR image to grayscale.                   |
| <code>cv2.selectROI</code>                                    | Select a rectangular target region.               |
| <code>np.zeros_like</code>                                    | Create an image-sized mask or drawing canvas.     |
| <code>cv2.goodFeaturesToTrack</code>                          | Detect distinctive trackable points.              |
| <code>cv2.calcOpticalFlowPyrLK</code>                         | Estimate point correspondences between frames.    |
| <code>np.median</code>                                        | Robustly aggregate point locations/displacements. |
| <code>cv2.line</code> / <code>cv2.circle</code>               | Visualise motion and feature locations.           |
| <code>cv2.rectangle</code>                                    | Visualise the target ROI.                         |
| <code>cv2.waitKey</code>                                      | Control a simple OpenCV display loop.             |
| <code>cap.release</code> / <code>cv2.destroyAllWindows</code> | Release resources.                                |

## One-Page Reference: The Whole Idea

### Workshop 3 perception pipeline



### Think in layers

| Layer             | Question                                                    |
|-------------------|-------------------------------------------------------------|
| Input             | What did the camera/video actually provide?                 |
| Feature detection | Where are distinctive, task-relevant image locations?       |
| Correspondence    | Where did those locations move in the next frame?           |
| Validation        | Which correspondences should I trust?                       |
| Aggregation       | How can many noisy measurements become one useful estimate? |
| Recovery          | What should I do when the evidence becomes too weak?        |
| Decision          | Is the estimate reliable enough for the next stage?         |

#### The most important Week 3 lesson

A perception system is not reliable simply because it produces an output. Reliability comes from understanding what is measured, checking the evidence, recognising assumptions, and having a response when those assumptions fail.

### Engineering workflow to remember

**RUN -> OBSERVE -> EXPLAIN -> STRESS-TEST -> DIAGNOSE -> IMPROVE**

Carry this workflow into later robotics projects. The algorithm may change, the sensor may change, and the model may become more sophisticated, but the engineering responsibility remains the same: understand the evidence before trusting it.

### End of Chapter 3