ESSE 2220 – Lab 8 Report

Feature Matching in Satellite Images (BFMatcher & FLANN)

Course: ESSE 2220
Lab Title: Feature Matching in Satellite Images (BFMatcher & FLANN)
Date Performed: 2025-11-14
Date Submitted: 2025-11-14

Names & Group Info

Group Number: GROUP 5
Members: Yathharthha Kaushal · Owen Oliver

---

1. Dataset Pair

1.1 The downloaded satellite image pair used for this lab is pair 8 with the filenames:

- socal-fire_00000325_pre_disaster.png
- socal-fire_00000325_post_disaster.png

1.2 Original Images

Before Disaster:

After Disaster:

1.3 Scene Description

> The scene shows a semi-urban area with a mix of residential buildings to the right, and hilly terrain with vegetation to the left, the road seperating the two areas is surrounded by dirt, with some dirt trails branching off.

---

2. Summary Table of All Methods

Matcher	Detector	Raw Matches	Distance Filter	Ratio Test	Symmetry	Observation
BF	SIFT	10664	1701	662	3471	Dense but noisy until filtered
BF	ORB	500	500	21	193	Few matches yet mostly crisp around structures
FLANN	SIFT	10664	1612	720	--	Slightly noisier because ANN search returns near-duplicates
FLANN	ORB	489	489	37	--	Sparse but consistent thanks to binary descriptors

---

3. Questions About Your Table

3.1 Which combination produced the largest number of raw matches? Why?

> BF + SIFT (and equivalently FLANN + SIFT) produced the most raw matches at 10 664 because SIFT detects many high-quality scale-invariant keypoints in this textured fire scene, giving the matcher far more descriptors to pair than ORB.

3.2 Which filtering method removed the most mismatches?

> The Lowe ratio test on the SIFT descriptors removed the most mismatches (dropping BF matches from 10 664 to 662 and FLANN matches to 720) because comparing the top two candidates suppresses ambiguous pairings that share similar distances.

3.3 Which method (BF or FLANN) gave more stable results between your two satellite images?

> BFMatcher was more stable: both the raw and filtered BF counts were consistent between detectors, and the symmetry cross-check ensured each match was mutual, so small illumination or viewpoint shifts between the pre/post images did not change the retained matches much.

3.4 Which detector (SIFT or ORB) worked better for your pair?

> SIFT worked better—thousands of its matches survived the filters while ORB retained only a few dozen after quality checks, indicating that scale and rotation invariance from SIFT was necessary for this scene's fine structural changes.

---

4. Feature Extraction (SIFT and ORB) – Timing + Descriptors

4.1 Extraction Time for Each Image and Detector

SIFT

Image	Keypoints	Descriptor Shape	Dtype	Extraction Time (s)
PRE	10,664	(10664, 128)	float32	0.3962
POST	7,727	(7727, 128)	float32	0.5271

ORB

Image	Keypoints	Descriptor Shape	Dtype	Extraction Time (s)
PRE	500	(500, 32)	uint8	0.0425
POST	500	(500, 32)	uint8	0.0379

4.2 Why ORB is Typically Faster than SIFT

> ORB relies on FAST keypoint detection plus BRIEF-style binary descriptors, so it mostly performs intensity comparisons and bit operations at a single scale pyramid. SIFT, in contrast, builds multi-scale Difference-of-Gaussian pyramids, fits sub-pixel extrema, and forms 128-D histogram descriptors with floating-point gradients. The additional octave construction and floating-point math make SIFT far heavier per keypoint, which is why my timings show ORB finishing in ~0.04 s while SIFT takes roughly ten times longer on the same images.

---

5. Filtering Methods – BFMatcher

5.1 Distance Filter

> **Distance Threshold:** Removes obviously bad matches where the descriptor distance is large (e.g., above 225 for ORB), so it keeps only the closest-looking correspondences in descriptor space.
> **Lowe's Ratio Test:** For each keypoint it compares the best and second-best neighbours; if the best distance is not at least ~30% smaller, that match is likely ambiguous and gets discarded.
> **Symmetry / Cross-Check:** Forces mutual agreement by keeping only pairs where feature A's best match is feature B and vice versa, which drops one-sided or repetitive texture matches that tend to be false positives.

---

6. Filtering Methods – FLANN

6.1 Why FLANN Might Produce Slightly Different Results Each Run

> FLANN performs approximate nearest-neighbour searches that use randomized trees (KD-trees or LSH tables) and sample only a subset of leaves per query. The random seeding and the probabilistic search order mean two runs can explore different branches, so the exact neighbour returned—and therefore the match counts—can drift slightly between executions.

6.2 What Type of Mismatches Each Filter Removes

> **Distance Threshold:** Applies the same hard cutoff to FLANN's approximate matches, trimming very weak pairings that still slipped through the ANN search.
> **Lowe's Ratio Test:** Works the same as with BF but guards against approximate neighbours by ensuring the top candidate is significantly better than the runner-up, which rejects many near-duplicate keypoints in vegetation.

Appendix

A1. All 14 Required Output Images

View Output Images

Or if this button isn't working, then just type this url in your browser: https://github.com/YK12321/ESSE2220-Labs/tree/main/8/code/output

A2. Complete Python Code

```
"""
Lab 8: Feature Matching (BFMatcher + FLANN + Filtering)
------------------------------------------------------------
Instructions:
1. Choose ONE detector at a time (SIFT or ORB).
2. Extract features and measure extraction time.
3. Run BFMatcher (normal and crossCheck).
4. Run FLANN with
5. Apply THREE filters:
       - Distance Threshold
       - Lowe's Ratio Test
       - Symmetry Check (BFMatcher crossCheck=True)
6. Visualize all results
7. Save all required images.
8. Repeat with the second detector.
"""

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

# ------------------------------------------------------------
# Step 1: Load images
# ------------------------------------------------------------
img1 = cv2.imread("socal-fire_00000325_pre_disaster.png", cv2.IMREAD_GRAYSCALE)
img2 = cv2.imread("socal-fire_00000325_post_disaster.png", cv2.IMREAD_GRAYSCALE)


# ------------------------------------------------------------
# Step 2: Choose ONE detector (uncomment one)
# ------------------------------------------------------------
# ---- SIFT ----
# detector = cv2.SIFT_create()

# ---- ORB ----
detector = cv2.ORB_create()


# ------------------------------------------------------------
# Step 3: Detect + compute (with timing)
# ------------------------------------------------------------
t0 = time.time()
kp1, des1 = detector.detectAndCompute(img1, None)
t1 = time.time()
kp2, des2 = detector.detectAndCompute(img2, None)
t2 = time.time()

print("Keypoints img1:", len(kp1))
print("Keypoints img2:", len(kp2))

print("Descriptor shapes:", des1.shape, des2.shape)

print("Extraction time (img1):", t1 - t0)
print("Extraction time (img2):", t2 - t1)


# ------------------------------------------------------------
# Step 4: Create matchers (BF + crossCheck + FLANN)
```

```python
# ------------------------------------------------------------

# BF normal matcher
# (For SIFT → NORM_L2, For ORB → NORM_HAMMING)
bf = cv2.BFMatcher(cv2.NORM_HAMMING)

# BF symmetry check matcher
bf_cross = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

# FLANN matcher
# algorithm = type of search structure (1 = KDTree, 6 = LSH)
# trees = number of trees (more trees → slower but more accurate)
# checks = how many leaf nodes to search (higher = better accuracy)
flann = cv2.FlannBasedMatcher(
    indexParams = dict(algorithm=6, trees=5),
    searchParams = dict(checks=50)
)


# ------------------------------------------------------------
# Step 5: Perform matching (raw outputs)
# ------------------------------------------------------------

# Best match for each descriptor
matches_bf = bf.match(des1, des2)

# Two best matches (m,n) for each descriptor
knn_bf = bf.knnMatch(des1, des2, k=2)

# FLANN single best match (approximate)
matches_flann = flann.match(des1, des2)

# FLANN two best matches (approximate)
knn_flann = flann.knnMatch(des1, des2, k=2)


# ------------------------------------------------------------
# Step 6: Filtering methods
# ------------------------------------------------------------

# Distance Threshold
# Typical range:
#   SIFT: 150–300
#   ORB:  20–60
def distance_filter(matches, threshold):
    return [m for m in matches if m.distance < threshold]


# Lowe’s Ratio Test (m.distance < ratio * n.distance)
# Typical range: 0.6–0.8
def ratio_test(knn_matches, ratio=0.75):
    good = []
    # knn_matches is a list-of-lists: each element contains up to k matches.
    # E.g., when k=2, knnMatch normally returns pairs like [m, n].
    # But sometimes FLANN/BFMatcher will return fewer than k matches for a
    # descriptor (for example when a second-neighbour couldn't be found).
    #
    # The old version did `for m, n in knn_matches:` which assumes every list
    # element has at least two values; that causes a ValueError when a
    # singleton list appears. To prevent that, we check the length and skip
    # entries that don't have two matches.
    for m_n in knn_matches:
        # make sure we have at least two neighbours; skip if not
        if not m_n or len(m_n) < 2:
            # Skipping entries with only one or zero matches avoids unpacking
            # errors and is consistent with common implementations of
            # Lowe's ratio test (which compares the best and second-best).
            continue
        # Unpack first and second best matches and apply Lowe's ratio test
        m, n = m_n[0], m_n[1]
        if m.distance < ratio * n.distance:
            good.append(m)
    return good


# Symmetry Check: already done using BFMatcher crossCheck=True
matches_sym = bf_cross.match(des1, des2)


# ------------------------------------------------------------
# Step 7: Apply ONE method at a time
# ------------------------------------------------------------
# Students should comment/uncomment ONE line:

# chosen_matches = matches_bf                   # Raw BF
# chosen_matches = matches_flann                # Raw FLANN
# chosen_matches = distance_filter(matches_bf, threshold=225)
# chosen_matches = ratio_test(knn_bf, ratio=0.7)
# chosen_matches = matches_sym                  # Symmetry check
# chosen_matches = distance_filter(matches_flann, threshold=225)
chosen_matches = ratio_test(knn_flann, ratio=0.7)


# ------------------------------------------------------------
# Step 8: Visualization (ONE LINE ONLY)
# ------------------------------------------------------------

img_out = cv2.drawMatches(img1, kp1, img2, kp2, chosen_matches, None)
cv2.imwrite("flann_ratio.jpg", img_out)
print("Matches: ", len(chosen_matches))


# ------------------------------------------------------------
# REQUIRED OUTPUT IMAGES
# ------------------------------------------------------------
"""
For EACH detector (SIFT, ORB):

1. raw_bf.jpg               (matches_bf)
2. raw_flann.jpg            (matches_flann)
3. bf_distance.jpg          (distance filter on BF)
4. bf_ratio.jpg             (ratio test on BF)
5. bf_symmetry.jpg          (crossCheck output)
6. flann_distance.jpg       (distance filter on FLANN)
7. flann_ratio.jpg          (ratio test on FLANN)
```

```
Total: 7 images per detector → 14 images in the lab.
"""
```

Performance analysis helper

```
"""Lab 8 – Section 4 helper script.

This streamlined version only gathers the numbers needed for
Section 4 of the report (feature extraction time + descriptor info
for both SIFT and ORB). No matchers or images are produced here.
"""

from pathlib import Path
import time

import cv2


IMAGE_NAMES = {
    "pre": "socal-fire_00000325_pre_disaster.png",
    "post": "socal-fire_00000325_post_disaster.png",
}


def load_images():
    """Load the grayscale satellite pair and fail loudly if missing."""
    base_dir = Path(__file__).resolve().parent
    images = {}
    for label, name in IMAGE_NAMES.items():
        path = base_dir / name
        img = cv2.imread(str(path), cv2.IMREAD_GRAYSCALE)
        if img is None:
            raise FileNotFoundError(f"Could not open {path}")
        images[label] = img
    return images


def time_extraction(detector, image, warmup=True):
    """Detect keypoints/descriptors and report duration."""
    if warmup:
        # First call primes OpenCV's internal buffers so the timed run
        # measures steady-state performance instead of one-time setup cost.
        detector.detectAndCompute(image, None)
    start = time.perf_counter()
    keypoints, descriptors = detector.detectAndCompute(image, None)
    duration = time.perf_counter() - start
    return keypoints, descriptors, duration


def describe_descriptors(descriptors):
    if descriptors is None:
        return "None", "None"
    return str(descriptors.shape), str(descriptors.dtype)


def main():
    images = load_images()
    detector_factories = {
        "SIFT": cv2.SIFT_create,
        "ORB": cv2.ORB_create,
    }

    for detector_name, create_detector in detector_factories.items():
        print(f"=== {detector_name} ===")
        for label, image in images.items():
            detector = create_detector()
            keypoints, descriptors, duration = time_extraction(detector, image)
            shape, dtype = describe_descriptors(descriptors)
            print(
                f"{label.upper()} | keypoints: {len(keypoints):5d} | "
                f"descriptor shape: {shape} | dtype: {dtype}"
            )
            print(f"Extraction time: {duration:.4f} s")
        print()


if __name__ == "__main__":
    main()
```