Verification and Validation Report: Image Feature Correspondences for Camera Calibration

Kiran Singh

April 16, 2025

1 Revision History

Date	**Version**	**Notes**
2025-04-16	Rev 1.0	Initial Release

2 Symbols, Abbreviations and Acronyms

symbol	**description**
CSV	comma separated value
FAST	Features from Accelerated Segment Test
IFCS	Image Feature Correspondences for Camera Calibration software
T	Test

Contents

1 Revision History **i**

2 Symbols, Abbreviations and Acronyms **ii**

3 Functional Requirements Evaluation **1**
- 3.1 Feature Detection 1
- 3.2 Feature Comparison 4

4 Nonfunctional Requirements Evaluation **6**
- 4.1 Reliability 6
- 4.2 Usability 6
- 4.3 User Experience 7
- 4.4 Maintainability 9
- 4.5 Performance 9

5 Comparison to Existing Implementation **9**

6 Unit Testing **9**

7 Changes Due to Testing **11**

8 Automated Testing **13**

9 Trace to Requirements **14**

10 Trace to Modules **15**

11 Code Coverage Metrics **16**
- 11.1 Control Module 16
- 11.2 Plot Image Module 16
- 11.3 Output Format Module 16

List of Tables

1 Traceability matrix between modules and identified unit tests 14

2 Traceability matrix mapping unit tests to associated software modules(M2–M11). 15

List of Figures

1 Outputs of the image smoothing procedure on an untransformed ArUco marker 2
2 Greyscale and noise-reduced images generated from the building dataset 3
3 Generated images of the keypoint detection test with mapped keypoints 4
4 Generated images of the keypoint detection test with mapped keypoints 5
5 Optimal candidate matches between ArUco_000 and ArUco_001 6
6 Outline of automated unit tests 10
7 Unit Test Summary.txt 11
8 Generated Unit Test Reports 12
9 Specification Parameters Unit Test Report 12
10 Summary of code coverage across all modules 17
11 Code coverage for Control Module 18
12 Code coverage for Plot Image Module 18
13 Code coverage for Output Format Module 19

This document outlines the results of the system and unit tests for the Image Feature Correspondences for Camera Calibration software. The detaileds of the associated tests are outlined in the **VnV Plan**.

3 Functional Requirements Evaluation

3.1 Feature Detection

Image Smoother

1. **STFR-IS-01**

   This test evaluates the capacity of the system to import generated ArUco marker imagery, convert the imagery to a greyscale format, perform Gaussian smoothing, and save both the greyscale and smoothed images using a kernel size of 5 and a standard deviation of 1.0. This test was manually initiated and executed via Pytest using the test_STFR-IS-01.py program. The test passed all checks and provides a summary.txt of the results under its timestamped STFR-IS-01 output folder.

   Requirements Addressed:

   - R1 (Update Image Noise)
   - R5 (Default Noise Suppression)
   - R9 (Perform Noise Reduction)

   Figure 1 shows an example of the generated greyscale and smoothed imagery of the ArUco markers. All generated output imagery can be reviewed in the STFR-IS-01 output folder.

   As the results from processing a binary ArUco marker do not produce a distinct change in colour to the human eye, a second test was executed on the test imagery provided in the testImages\building folder.

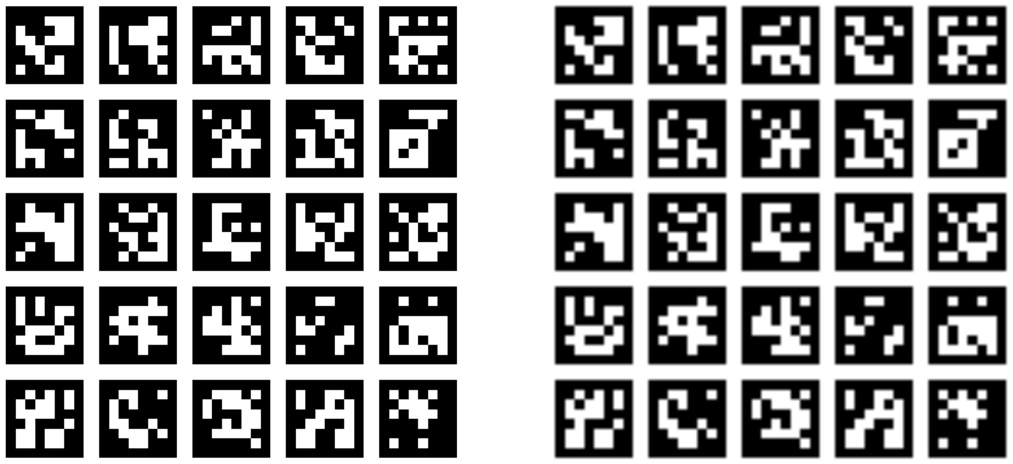

(a) Converted Greyscale Image (b) Gaussian-Smoothed Image

Figure 1: Outputs of the image smoothing procedure on an untransformed ArUco marker

Keypoint Detection

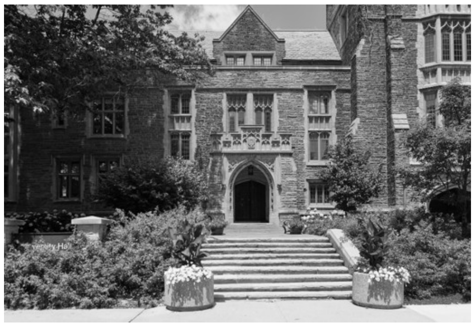
1. **STFR-KP-01**
   This test evaluates the capacity of the system to import smoothed ArUco marker imagery and identify keypoints through use of rotated-FAST methods. A pixel intensity threshold of 60 was set for this test. This test was manually initiated and executed via Pytest using the test_STFR-KP-01.py program. The test successfully passed all checks and provides a summary.txt of the results under its timestamped STFR-KP-01 output folder.

   Requirements Addressed:

   - R2 (Update Pixel Intensity Threshold)
   - R6 (Perform Corner Detection)
   - R10 (Identify Keypoints)

   Figure 3 shows an example of the generated greyscale and smoothed imagery of the ArUco markers. All generated output imagery and corresponding CSV files can be reviewed in the STFR-KP-01 output folder.

(a) Building - RGB Input Image


(b) Building - Greyscale Image

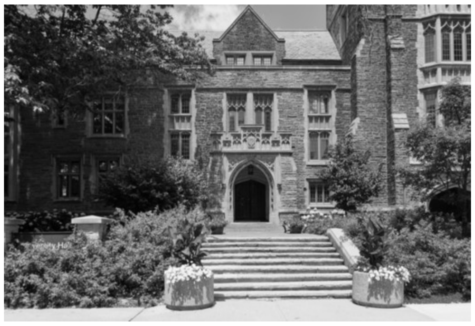

(c) Building - Smoothed Image

Figure 2: Greyscale and noise-reduced images generated from the building dataset

Feature Description

1. **STFR-FD-01**
   This test evaluates the capacity of the system to create oriented-BRIEF descriptors using identified keypoints from smoothed greyscale imagery. A target of 100 was set with a search patch size of 31. This test was manually initiated and executed via Pytest using the test_STFR-FD-01.py program. The test successfully passed all checks and provides a summary.txt of the results under its timestamped STFR-FD-01 output folder.

   Requirements Addressed:

   - R3 (Update Patch Size)

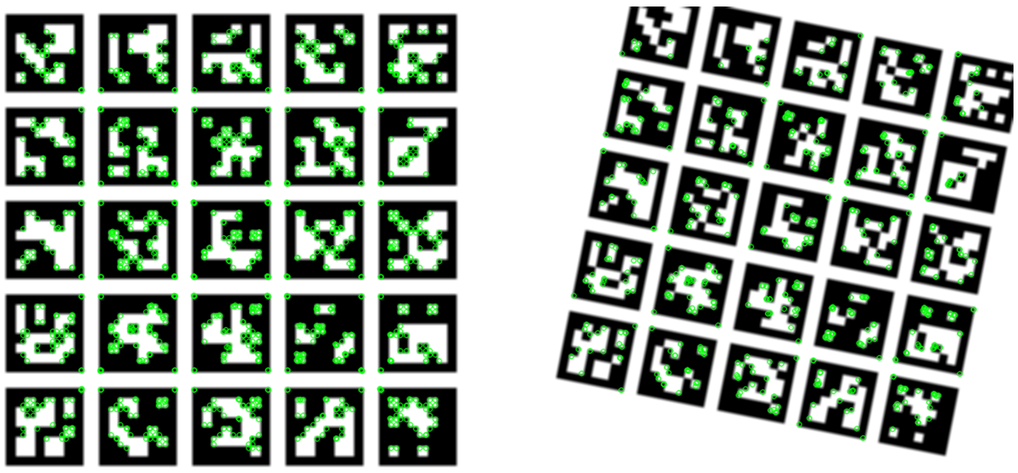

(a) ArUco_000 with keypoints (b) ArUco_001 with keypoints

Figure 3: Generated images of the keypoint detection test with mapped keypoints

- R4 (Update Descriptor Bin Size)
- R7 (Binary Descriptors)
- R11 (Define Descriptors)

Figure 4 shows an example of two ArUco markers with keypoints scaled per their feature descriptors. All generated output imagery and corresponding CSV files can be reviewed in the STFR-FD-01 output folder.

3.2 Feature Comparison

Descriptor Comparison

1. **STFR-FM-01**
   This test evaluates the capacity of the system to compare two sets of predefined feature descriptor between two similar images. Using brute-force matching, a maximum Hamming distance of 25 was set for displayed images. Additionally, a maximum of 30 match candidates were permitted to be displayed in the generated imagery. This test was

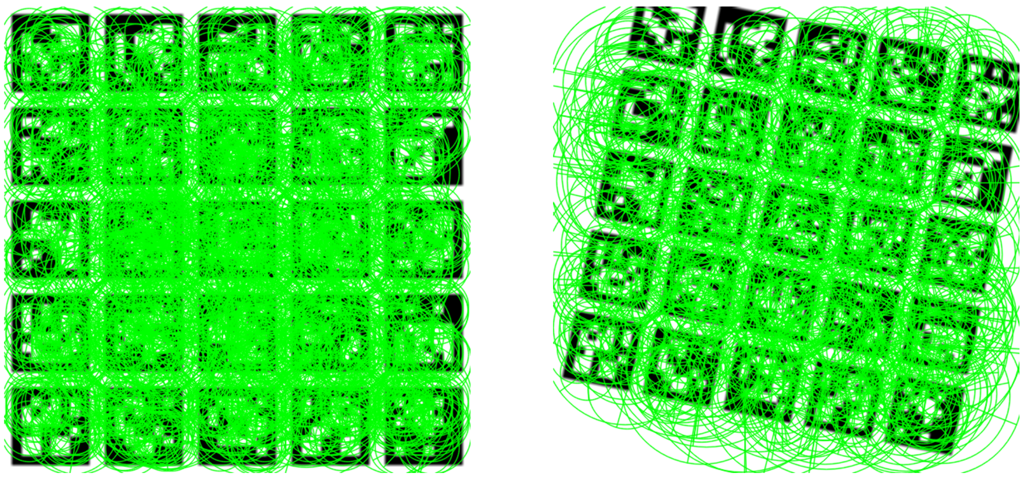

(a) ArUco_000 with scaled keypoints (b) ArUco_001 with scaled keypoints

Figure 4: Generated images of the keypoint detection test with mapped keypoints

manually initiated and executed via Pytest using the test_STFR-FM-01.py program. The test successfully passed all checks and provides a summary.txt of the results under its timestamped STFR-FM-01 output folder.

Requirements Addressed:

- R8 (Descriptor Comparison - Hamming Distance)
- R12 (Search for Matches Candidates)
- R13 (Verify Match Candidates)
- R14 (Confirm Pose Identifiers)
- R15 (Report Match Candidates)

Figure 5 shows an example of thirty candidate feature matches between markers ArUco_000 and ArUco_001. All generated output imagery and corresponding CSV files can be reviewed in the STFR-FM-01 output folder.

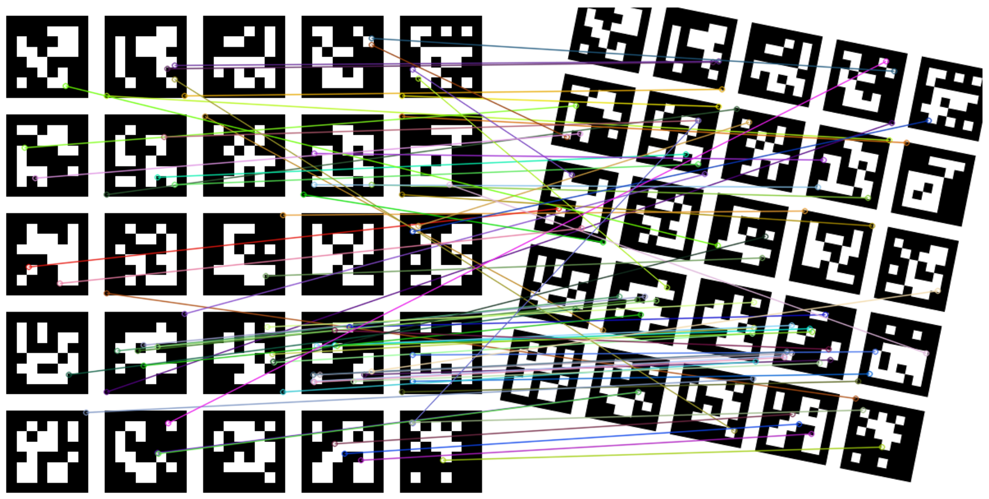


Figure 5: Optimal candidate matches between ArUco_000 and ArUco_001

4 Nonfunctional Requirements Evaluation

4.1 Reliability

- NFR1 (Invariance to the Order of Images)

A test script named **test_STNFR-RE-01.py** was prepared. In this script, two images from the **Lego** image set are renamed and reordered, as outlined in the **Lego_Swap** folder. Both of these datasets are processed to identify candidate matches. The resulting match candidates are compared between each data set to confirm that each descriptor and corresponding coordinates from the Lego dataset is matched to the equivalent descriptor in the transformed Lego_Swap dataset. This test passed with zero errors, and the results are summarized in the **STNFR-RE-01**.

4.2 Usability

Requirements addressed:

- NFR2 (Simple to Use)

4.3 User Experience

STNFR-UX-1

The Domain Expert was tasked with executing the complete end-to-end feature matching process using the IFC software. The user was provided with a dataset of 12 ArUco images from a single camera, each image labeled according to a consistent naming pattern. All images were extracted from the Raw Image Library offered within the IFCS program. Each image was processed using the default configuration.

The user was responsible for updating the IFC configuration file with the image path, the desired output directories, and appropriate camera metadata. Following this setup, the user executed the pipeline and reviewed the outputs in the designated folders.

The test was then repeated using the Lego2 image set from the library to affirm the results of the test. However, the user was asked to update the following parameters within the Parameter Specification Module and to describe their experience.

The following aspects of the experience were evaluated using a scale of 1 (Very Difficult/Unsatisfied) to 10 (Very Easy/Satisfied):

- **Ease of preparing input files:** 8 — The naming structure was intuitive.
- **Ease of assigning camera properties:** 7 — Manual but manageable.
- **Clarity of configuration file:** 7 — Clear but would benefit from examples.
- **Understandability of output files:** 8 — Outputs were logically organized and interpretable.
- **Overall satisfaction with the pipeline:** 9 — The experience was efficient and the results were accurate.
- **Comfort modifying processing parameters:** 6 — Could benefit from using a YAML file instead of direct source code edits.

- **Likelihood of using the software again:** 9 — User expressed strong confidence in the pipeline.

The test confirmed that the IFC software is usable by technical personnel with domain knowledge, and the outputs aligned with the user's expectations. Although system parameters were not modified in the test case with ArUco markers, the test with the Lego2 images set yielded valuable insight as the user noted that a YAML configuration interface would improve their perceived ease of use in future tests.

IFCS_DEMO-01

This assessment was introduced in an adhoc fashion to support the code walkthrough of the IFCS. Ten (10) audience members with varying degrees of experience in image processing, from novice to advanced, were given a demonstration of the IFCS software. This demonstration included a detailed walkthough of the download process, installation processes, and setup of the virtual environment. Audience members were shown how to add new images from the provided library to the inputs folder, adjust the methods and parameters of image processing, and initiate the IFCS pipeline. Once completed, audience members were shown the following outputs.

- generated greyscale imagery
- generated smoothed imagery
- generated imagery with keypoints
- generated imagery with descriptors
- generated imagery that compare images with candidate descriptor matches
- CSV files of keypoints, descriptors and candidate matches

Following the demonstration, a show-of-hands identified that seven (7) of ten audience members stated that they have a favourable opinion of the software and its simplicity of use. At a success rate of 70%, this fall below the target rate of 80%. However, this may be improved with the implementation of any one of a user manual, user walkthrough video, or a one-on-one training session with one of the IFCS developers.

4.4 Maintainability

Requirements Addressed:

- NFR3 (Allocation of Developer Resources for New Features)

Per the **VnV Plan**, this requirement has been identified as out of scope for the Rev 1.0 release.

4.5 Performance

Requirements Addressed:

- NFR4 (Timing Metrics)
- NFR5 (Memory Usage Metrics)

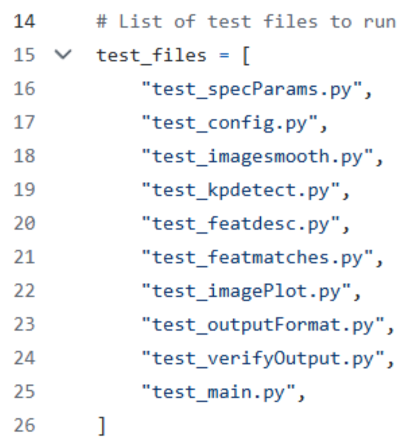
Timing metrics and memory usage were identified as test features of interest for the lifespan of the IFCS software. These tools would be appended to compare the relative performance of different methods such as FAST and Harris scores for keypoint detection. However, as the scope of the Winter 2025 development cycle narrowed to prioritize robust performance of ORB feature detection and brute-force matching, the implementation of timing metrics will be deferred to the development cycle of Summer 2025. These metric wil be assessed as part of the systems tests as follows through the Pytest Monitor plugin, which has the capacity to assess both timing and memory metrics and is suitable for integration with GitHub Actions.

5 Comparison to Existing Implementation

This section is **not applicable**.

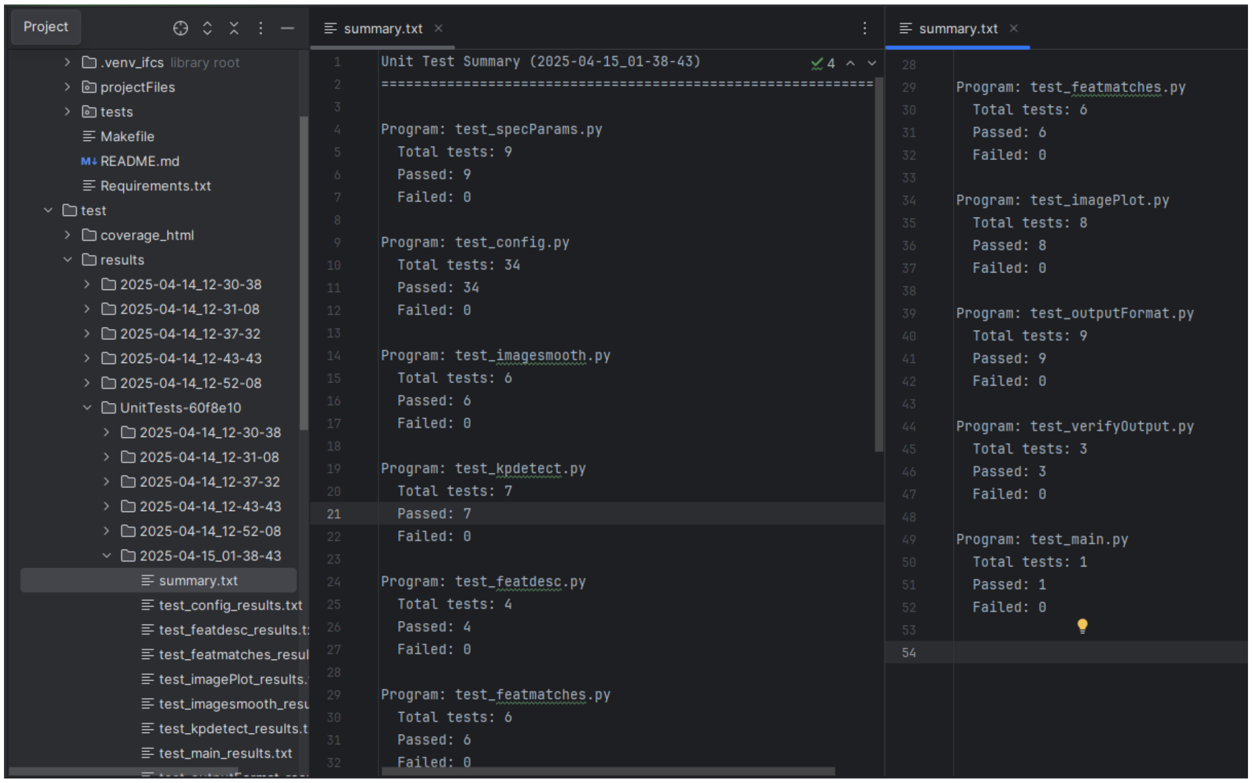
6 Unit Testing

All unit tests are automated to run via a pull request and Pytest Github Actions. These tests can be found in the **test** folder. Each test is initiated by the **run_unit_checks.py** program, as shown in Figure 6. Upon completion of the unit tests a **summary** file is generated that outlines the quantity of tests that have passed or failed for each module, as shown in Figure 7. In the same folder, a **detailed report of each unit test** is outlined for the corresponding module, as shown in Figure 8. A detailed example of the unit test report for the **Specification Parameters Module (M4)** is outlined in Figure 9. All unit tests were shown to have passed successfully for each module.



```
# List of test files to run
test_files = [
    "test_specParams.py",
    "test_config.py",
    "test_imagesmooth.py",
    "test_kpdetect.py",
    "test_featdesc.py",
    "test_featmatches.py",
    "test_imagePlot.py",
    "test_outputFormat.py",
    "test_verifyOutput.py",
    "test_main.py",
]
```

Figure 6: Outline of automated unit tests



```
Project
  > .venv_ifcs library root
  > projectFiles
  > tests
    Makefile
    README.md
    Requirements.txt
v test
  > coverage_html
  v results
    > 2025-04-14_12-30-38
    > 2025-04-14_12-31-08
    > 2025-04-14_12-37-32
    > 2025-04-14_12-43-43
    > 2025-04-14_12-52-08
    v UnitTests-60f8e10
      > 2025-04-14_12-30-38
      > 2025-04-14_12-31-08
      > 2025-04-14_12-37-32
      > 2025-04-14_12-43-43
      > 2025-04-14_12-52-08
      v 2025-04-15_01-38-43
          summary.txt
          test_config_results.txt
          test_featdesc_results.t
          test_featmatches_resul
          test_imagePlot_results.
          test_imagesmooth_resu
          test_kpdetect_results.t
          test_main_results.txt
```

```
summary.txt
Unit Test Summary (2025-04-15_01-38-43)
============================================================

Program: test_specParams.py
  Total tests: 9
  Passed: 9
  Failed: 0

Program: test_config.py
  Total tests: 34
  Passed: 34
  Failed: 0

Program: test_imagesmooth.py
  Total tests: 6
  Passed: 6
  Failed: 0

Program: test_kpdetect.py
  Total tests: 7
  Passed: 7
  Failed: 0

Program: test_featdesc.py
  Total tests: 4
  Passed: 4
  Failed: 0

Program: test_featmatches.py
  Total tests: 6
  Passed: 6
  Failed: 0
```

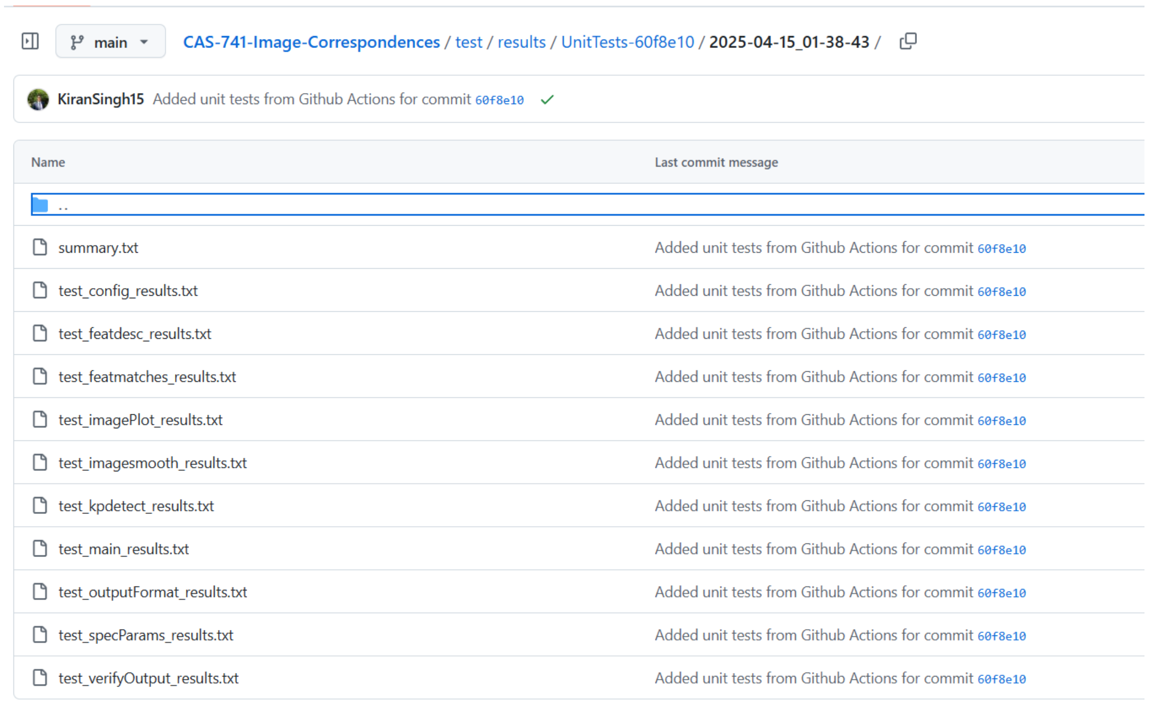
```
summary.txt

Program: test_featmatches.py
  Total tests: 6
  Passed: 6
  Failed: 0

Program: test_imagePlot.py
  Total tests: 8
  Passed: 8
  Failed: 0

Program: test_outputFormat.py
  Total tests: 9
  Passed: 9
  Failed: 0

Program: test_verifyOutput.py
  Total tests: 3
  Passed: 3
  Failed: 0

Program: test_main.py
  Total tests: 1
  Passed: 1
  Failed: 0
```

Figure 7: Unit Test Summary.txt

7 Changes Due to Testing

This section summarizes the most significant changes made to the modules and test infrastructure as a result of iterative testing, user feedback, and supervisor reviews, particularly following the Rev 0 demonstration. Each change was implemented to improve traceability, robustness, or test validation.

1. **Unification of Output Directory Structure**
   All outputs from functional tests (grayscale, smoothed images, keypoints, descriptors, and matches) are now saved under a single, timestamped path: `tests/Outputs/<timestamp>/`. This change was made in response to confusion around inconsistent file locations during test runs. It ensures better organization, test reproducibility, and avoids polluting production outputs.

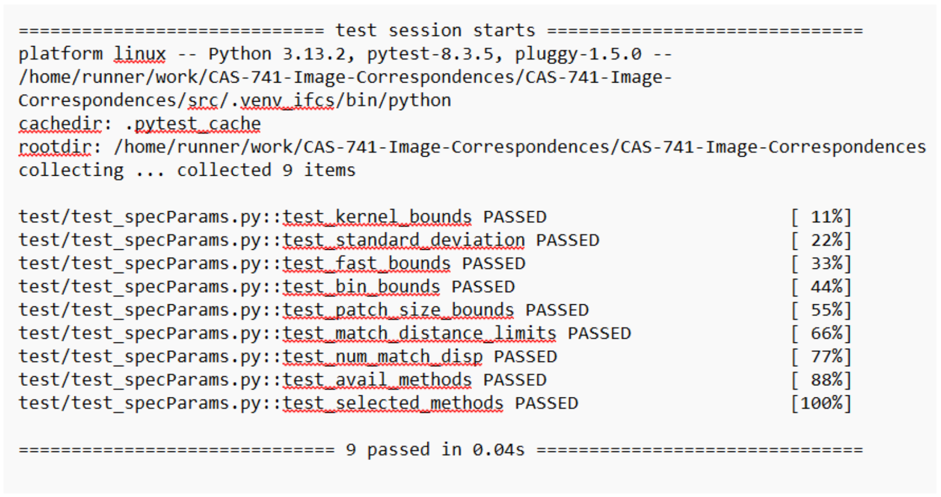


main   CAS-741-Image-Correspondences / test / results / UnitTests-60f8e10 / 2025-04-15_01-38-43 /

KiranSingh15 Added unit tests from Github Actions for commit 60f8e10 ✓

Name	Last commit message
..	
summary.txt	Added unit tests from Github Actions for commit 60f8e10
test_config_results.txt	Added unit tests from Github Actions for commit 60f8e10
test_featdesc_results.txt	Added unit tests from Github Actions for commit 60f8e10
test_featmatches_results.txt	Added unit tests from Github Actions for commit 60f8e10
test_imagePlot_results.txt	Added unit tests from Github Actions for commit 60f8e10
test_imagesmooth_results.txt	Added unit tests from Github Actions for commit 60f8e10
test_kpdetect_results.txt	Added unit tests from Github Actions for commit 60f8e10
test_main_results.txt	Added unit tests from Github Actions for commit 60f8e10
test_outputFormat_results.txt	Added unit tests from Github Actions for commit 60f8e10
test_specParams_results.txt	Added unit tests from Github Actions for commit 60f8e10
test_verifyOutput_results.txt	Added unit tests from Github Actions for commit 60f8e10

Figure 8: Generated Unit Test Reports



```
============================= test session starts ==============================
platform linux -- Python 3.13.2, pytest-8.3.5, pluggy-1.5.0 --
/home/runner/work/CAS-741-Image-Correspondences/CAS-741-Image-
Correspondences/src/.venv_ifcs/bin/python
cachedir: .pytest_cache
rootdir: /home/runner/work/CAS-741-Image-Correspondences/CAS-741-Image-Correspondences
collecting ... collected 9 items

test/test_specParams.py::test_kernel_bounds PASSED                       [ 11%]
test/test_specParams.py::test_standard_deviation PASSED                  [ 22%]
test/test_specParams.py::test_fast_bounds PASSED                         [ 33%]
test/test_specParams.py::test_bin_bounds PASSED                          [ 44%]
test/test_specParams.py::test_patch_size_bounds PASSED                   [ 55%]
test/test_specParams.py::test_match_distance_limits PASSED               [ 66%]
test/test_specParams.py::test_num_match_disp PASSED                      [ 77%]
test/test_specParams.py::test_avail_methods PASSED                       [ 88%]
test/test_specParams.py::test_selected_methods PASSED                    [100%]

============================== 9 passed in 0.04s ===============================
```

Figure 9: Specification Parameters Unit Test Report

2. **Validation of CSV Content and Structure**
   Functional tests now verify not only the existence of output files but also the correctness of their internal structure, including column names and row counts. For example, descriptor CSVs must contain binary string columns, and match files must include both descriptors and matching scores. This change prevents false positives where tests previously passed despite incomplete or invalid outputs.

3. **Enforcement of Parameter Bounds**
   Dedicated unit tests were introduced to ensure that all configurable parameters (e.g., Gaussian kernel size, standard deviation, patch size, bin count, match threshold) are within valid numerical and type constraints. These checks catch invalid user input early, reducing undefined behavior during runtime.

4. **Synthetic Imagery for CI Integration**
   A minimal testing pipeline using synthetic image pairs was implemented in `test_main.py`. This allows the pipeline to be tested in continuous integration (CI) environments without relying on external image datasets. It also ensures the processing pipeline can execute end-to-end with minimal dependencies.

5. **Inclusion of Feature Metadata in Match Results**
   Feature match outputs were extended to include 256-bit binary descriptors for both the query and train features, image IDs, and matching scores. This change enables deeper validation of match quality and allows visual or statistical analysis of correspondence accuracy in test reports.

6. **Standardized Test Summaries for Reporting**
   Each test script now generates a summary file (e.g., `test_kpdetect_results.txt`, `test_main_results.txt`) detailing test names, pass/fail status, and parameters. This facilitates automated result aggregation in CI pipelines and provides a consistent audit trail for testing history.

8 Automated Testing

All unit tests are automated to run via a pull request and Pytest via Github Actions. These tests can be found in the test folder. A review of the Github

Actions tasks outlines the list of artifacts for each push to the repository. Each test is initiated by the **run_unit_checks.py** program, as shown in Figure 6.

9 Trace to Requirements

The traceability of both functional and nonfunctional requirements to system tests can be found in Table 1. We note that the functional requirements are entirely covered by the outlined tests. We also note that NFR3 has been omitted from the scope of testing, and that NF4 and NFR5 has been deferred at the time of release for Rev 1.0 due to resource constraints within the development team. Testing for these requirements will be implemented as part of development during the Summer of 2025.

Requirement	**STFR-IS-01**	**STFR-KP-01**	**STFR-FD-01**	**STFR-FM-01**	**STNFR-RE-01**	**IFCS-DEMO-01**
R1	X					
R2		X				
R3			X			
R4			X			
R5	X					
R6		X				
R7			X			
R8				X		
R9	X					
R10		X				
R11			X			
R12				X		
R13				X		
R14				X		
R15				X		
NFR1					X	
NFR2						X
NFR3	-	-	-	-	-	-
NFR4*	-	-	-	-	-	-
NFR5*	-	-	-	-	-	-

Table 1: Traceability matrix between modules and identified unit tests

10 Trace to Modules

A full outline of the traceability between the unit tests and the modules as outlined in the MIS is provided in Table 2.

Test Name	**M1**	**M2**	**M3**	**M4**	**M5**	**M6**	**M7**	**M8**	**M9**	**M10**	**M11**
test_pipeline_synthetic_images	/	x	–	–	–	–	–	–	–	–	–
test_check_parameter_limits_valid	/	–	x	–	–	–	–	–	–	–	–
test_check_method_limits_valid	/	–	x	–	–	–	–	–	–	–	–
test_check_method_limits_invalid	/	–	x	–	–	–	–	–	–	–	–
test_kernel_bounds	/	–	–	x	–	–	–	–	–	–	–
test_standard_deviation	/	–	–	x	–	–	–	–	–	–	–
test_fast_bounds	/	–	–	x	–	–	–	–	–	–	–
test_bin_bounds	/	–	–	x	–	–	–	–	–	–	–
test_patch_size_bounds	/	–	–	x	–	–	–	–	–	–	–
test_avail_methods	/	–	–	x	–	–	–	–	–	–	–
test_selected_methods	/	–	–	x	–	–	–	–	–	–	–
test_output_keypoints_variable_size	/	–	–	–	x	–	–	–	–	–	–
test_output_descriptors_variable_size	/	–	–	–	x	–	–	–	–	–	–
test_output_matches_variable_size	/	–	–	–	x	–	–	–	–	–	–
test_check_match_uniqueness_valid	/	–	–	–	–	x	–	–	–	–	–
test_check_match_uniqueness_warns_for_same_ids	/	–	–	–	–	x	–	–	–	–	–
test_check_match_uniqueness_warns_with_matches	/	–	–	–	–	x	–	–	–	–	–
test_smooth_image_valid_gaussian	/	–	–	–	–	–	x	–	–	–	–
test_detect_keypoints_with_valid_orb	/	–	–	–	–	–	–	x	–	–	–
test_compute_descriptors_valid	/	–	–	–	–	–	–	–	x	–	–
test_match_features_no_loss	/	–	–	–	–	–	–	–	–	x	–
test_gen_kp_img_with_none_keypoints	/	–	–	–	–	–	–	–	–	–	x
test_gen_kp_img_no_flag	/	–	–	–	–	–	–	–	–	–	x
test_gen_kp_img_rich_keypoints	/	–	–	–	–	–	–	–	–	–	x
test_gen_kp_img_with_none_image	/	–	–	–	–	–	–	–	–	–	x
test_gen_matched_features_success	/	–	–	–	–	–	–	–	–	–	x

Table 2: Traceability matrix mapping unit tests to associated software modules(M2–M11).

11 Code Coverage Metrics

To evaluate the code coverage of the selected unit tests described in Section **6**, a test script was prepared: `run_coverage_report.py`. This script utilizes the `pytest-cov` plugin (version 6.1.1) to generate coverage statistics. Upon execution, it runs all designated unit tests and exports the coverage results to the `test/coverage_html` directory. To visualize the results, the user can download the `index.html` file and open it locally in a web browser.

Figure 10 provides a summary of the code coverage results. Most modules achieve 100% coverage, with the exception of the following:

- **Control Module** (90%)
- **Plot Image Module** (95%)
- **Output Format Module** (94%)

11.1 Control Module

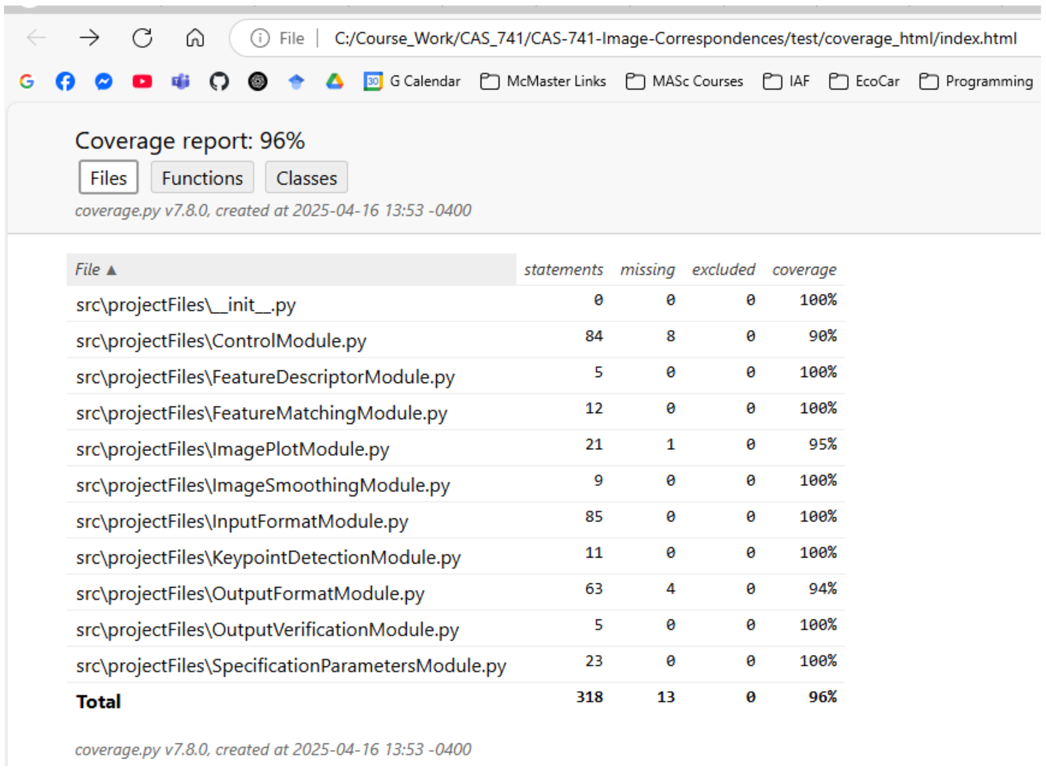
The Control Module achieved 90% coverage, as shown in Figure 11. The uncovered lines primarily correspond to fallback import statements located at the top of the module. These lines are included to facilitate relative package calls for unit testing and are not triggered during normal operation. Consequently, the reported coverage is considered sufficient.

11.2 Plot Image Module

The Plot Image Module reached 95% coverage. The uncovered line is a fallback statement used to support output image paths in alternative directory structures (e.g., `../Results/<timestamp>/Outputs` rather than `../projectFiles/Outputs`). As this line serves as a non-critical contingency, its exclusion from test coverage is considered acceptable.

11.3 Output Format Module

The Output Format Module achieved 94% coverage. The uncovered lines correspond to edge cases involving the absence of descriptors. As these cases


File
C:/Course_Work/CAS_741/CAS-741-Image-Correspondences/test/coverage_html/index.html
G Calendar
McMaster Links
MASc Courses
IAF
EcoCar
Programming
Files
Functions
Classes

Coverage report: 96%

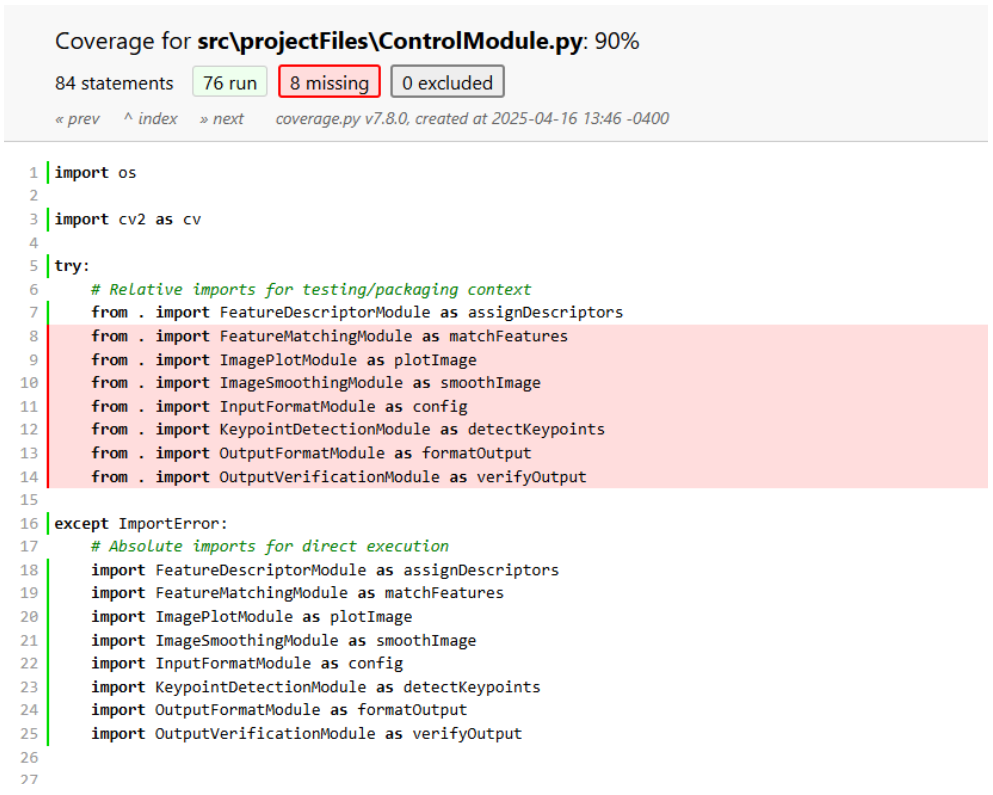
coverage.py v7.8.0, created at 2025-04-16 13:53 -0400

File ▲	statements	missing	excluded	coverage
src\projectFiles__init__.py	0	0	0	100%
src\projectFiles\ControlModule.py	84	8	0	90%
src\projectFiles\FeatureDescriptorModule.py	5	0	0	100%
src\projectFiles\FeatureMatchingModule.py	12	0	0	100%
src\projectFiles\ImagePlotModule.py	21	1	0	95%
src\projectFiles\ImageSmoothingModule.py	9	0	0	100%
src\projectFiles\InputFormatModule.py	85	0	0	100%
src\projectFiles\KeypointDetectionModule.py	11	0	0	100%
src\projectFiles\OutputFormatModule.py	63	4	0	94%
src\projectFiles\OutputVerificationModule.py	5	0	0	100%
src\projectFiles\SpecificationParametersModule.py	23	0	0	100%
Total	**318**	**13**	**0**	**96%**

coverage.py v7.8.0, created at 2025-04-16 13:53 -0400

Figure 10: Summary of code coverage across all modules

are expected to be handled internally by OpenCV—since descriptors are not computed unless explicitly initialized—the remaining coverage gap is not considered problematic.



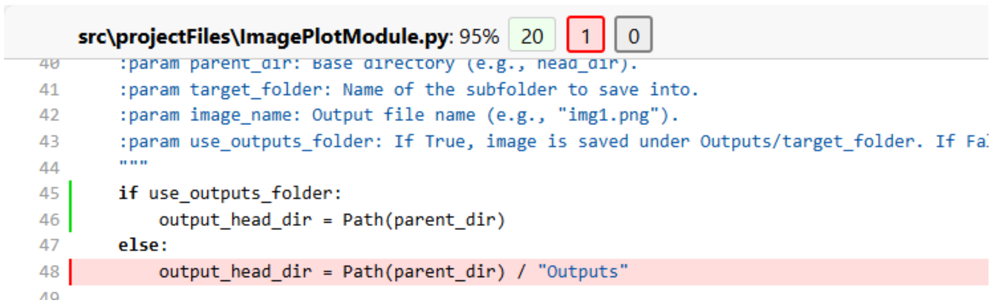
Coverage for **src\projectFiles\ControlModule.py**: 90%

84 statements 76 run 8 missing 0 excluded

« prev ^ index » next coverage.py v7.8.0, created at 2025-04-16 13:46 -0400

```python
import os

import cv2 as cv

try:
    # Relative imports for testing/packaging context
    from . import FeatureDescriptorModule as assignDescriptors
    from . import FeatureMatchingModule as matchFeatures
    from . import ImagePlotModule as plotImage
    from . import ImageSmoothingModule as smoothImage
    from . import InputFormatModule as config
    from . import KeypointDetectionModule as detectKeypoints
    from . import OutputFormatModule as formatOutput
    from . import OutputVerificationModule as verifyOutput

except ImportError:
    # Absolute imports for direct execution
    import FeatureDescriptorModule as assignDescriptors
    import FeatureMatchingModule as matchFeatures
    import ImagePlotModule as plotImage
    import ImageSmoothingModule as smoothImage
    import InputFormatModule as config
    import KeypointDetectionModule as detectKeypoints
    import OutputFormatModule as formatOutput
    import OutputVerificationModule as verifyOutput
```

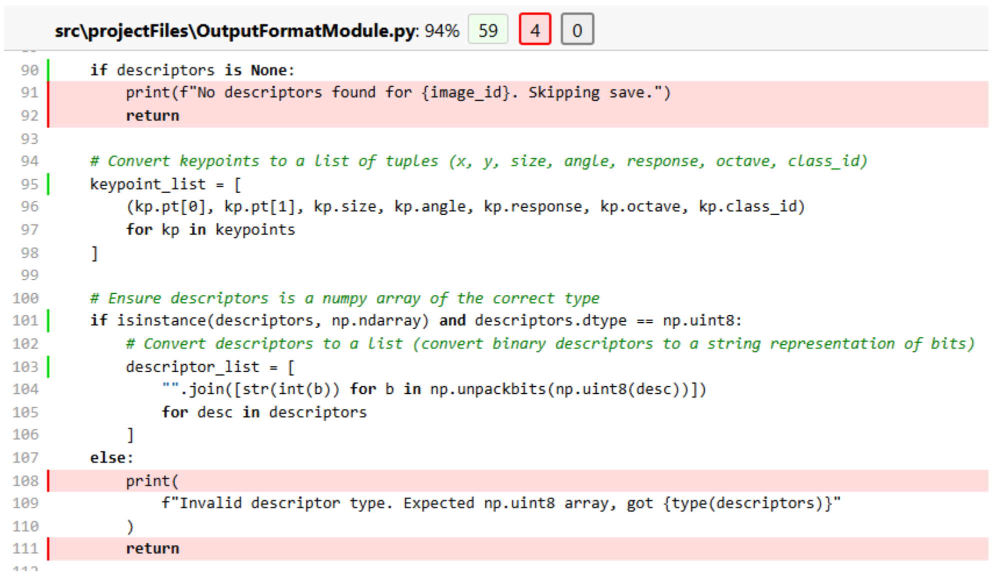
Figure 11: Code coverage for Control Module



src\projectFiles\ImagePlotModule.py: 95% 20 1 0

```python
    :param parent_dir: Base directory (e.g., head_dir).
    :param target_folder: Name of the subfolder to save into.
    :param image_name: Output file name (e.g., "img1.png").
    :param use_outputs_folder: If True, image is saved under Outputs/target_folder. If Fa
    """
    if use_outputs_folder:
        output_head_dir = Path(parent_dir)
    else:
        output_head_dir = Path(parent_dir) / "Outputs"
```

Figure 12: Code coverage for Plot Image Module



```python
    if descriptors is None:
        print(f"No descriptors found for {image_id}. Skipping save.")
        return

    # Convert keypoints to a list of tuples (x, y, size, angle, response, octave, class_id)
    keypoint_list = [
        (kp.pt[0], kp.pt[1], kp.size, kp.angle, kp.response, kp.octave, kp.class_id)
        for kp in keypoints
    ]

    # Ensure descriptors is a numpy array of the correct type
    if isinstance(descriptors, np.ndarray) and descriptors.dtype == np.uint8:
        # Convert descriptors to a list (convert binary descriptors to a string representation of bits)
        descriptor_list = [
            "".join([str(int(b)) for b in np.unpackbits(np.uint8(desc))])
            for desc in descriptors
        ]
    else:
        print(
            f"Invalid descriptor type. Expected np.uint8 array, got {type(descriptors)}"
        )
        return
```

Figure 13: Code coverage for Output Format Module

References