System Verification and Validation Plan for Image Feature Correspondences for Camera Calibration

Kiran Singh

April 15, 2025

Revision History

Date	Version	Notes
2025-02-27	1.0	Initial Release

Contents

List of Tables

List of Figures

1 Symbols, Abbreviations, and Acronyms

symbol	description
a	arbitrary positive integer
CI	Continuous Integration
CSV	Comma-Separated Values
DE	Domain Expert
d_{total}	bin size, or number of binary descriptions
FOV	field-of-view
I	2D array of pixel values for a greyscale image
I'	transformed 2D array of pixel values
IFC	Image Feature Correspondences
k	number of total keypoints
LD	Lead Developer
m	horizontal image dimension
MG	Module Guide
MIS	Module Interface Specification
n	vertical image dimension
ORB	Oriented FAST and Rotated BRIEF
p	patch size
PEP8	Python Enhancement Proposal-8
PNG	Portable Network Graphic
PS	Project Supervisor
RGB	Red-Green-Blue coloured imagery
SDI	Software Design Instructor
SLAM	Simultaneous Localization and Mapping
SRS	Software Requirements Specification
t	intensity threshold
VnV	Verification and Validation
σ	standard deviation

The intent of this document is to define the Verification and Validation (VnV) processes that will be used to assess Image Feature Correspondences for Camera Calibration, (IFC) software. Specifically, this document will be used to characterize the behaviour and performance of this software. The remaining sections of this document outline a detailed summary of the specific objectives of the VnV campaign. This includes considerations for the scope of the VnV efforts with respect to contraints that stem from the Winter 2025 development schedule, as well as anticipated activities for the Summer 2025 development cycle. Verification procedures are present for select deliverables and milestone, and include an overview of anticipated cases for both system tests unit tests.

2 General Information

2.1 Summary

The Image Feature Correspondences (IFC) software is a feature comparison algorithm that is intended to be used as part of a pipeline to perform extrinsic camera calibration for applications in mobile robotics. It accepts camera configuration parameters and greyscale imagery data at different poses to identify common features amongst collected images.

2.2 Objectives

The VnV process is intended to characterize how well the for the IFC software performs in its intended capacity to identify features amongst collected imagery. The performance of this system can vary significantly as it is influenced by factors such as overlap in camera fields-of-view (FOV), the observed contrast between objects in an image, and variance in scale, rotation, and ambient illumination conditions. Furthermore, as there is no common baseline to compare this software to as an oracle, the VnV campaign for the IFC software is intended to characterize the performance of the integrated image processing functions against a set of test datasets. Key objectives of this process are defined below.

- assess the reliability in the feature matching pipeline

- assess how users perceive their interactions with the IFC software

- characterize the performances of the IFC software with metrics such as processing time and memory usage

- cross-validate the performance of the IFC software with accepted bench-marked datasets

Verification of the individual functions within the OpenCV library themselves falls outside of the scope of this project, as we can assume that the library has been verified by its own implementation team.

2.3   Challenge Level and Extras

The proposed software constitutes a research project, as the IFC software serves as the front end of a robust optimization framework for extrinsic camera calibration. The IFC software must be capable of handling large volumes of high-resolution imagery while ensuring reliability through consistent and repeatable outputs. The software will be designed for robustness in data processing, accommodating a wide range of input conditions. Additionally, it will maintain a streamlined user experience, allowing users to override select default parameters while ensuring that the learning curve remains minimal for efficient adoption.

As the long-term goal for this software is to absorb it into a larger calibration pipeline for research in the domain of mobile robotic, this project is defined as an advanced level of challenge. This program will supply a set of installation instructions, source code, test scripts, test imagery, and verification report. No additional documentation, such as a user manual will be provided.

2.4   Relevant Documentation

Relevant documentation has been hyperlinked throughout the length of the document. This enables the reader to access each resource within the context of the section that the reference is invoked.

- **Software Requirements Specification (SRS)**
  Defines the functional and non-functional requirements of the IFC system. This document outlines the system context, theoretical models,

required inputs and outputs, and expected behavior under various operating conditions. It also establishes performance goals, usability expectations, and traceability with outlined models.

- **Module Guide (MG)**
  Presents the high-level architectural decomposition of the IFC software. The Module Guide describes the system's structure using a module hierarchy and outlines the responsibilities, relationships, and rationales behind each module, supporting design decisions with principles such as separation of concerns and information hiding.

- **Module Interface Specification (MIS)**
  Specifies the precise interface for each module defined in the Module Guide. This includes data types, function signatures, constants, and assumptions. The MIS supports correct and consistent implementation by detailing how modules interact and what services they provide.

Additional document includes the checklists as follows.

- **SRS Checklist**.

- **MG Checklist**

- **MIS Checklist**

- **VnV Checklist**

3 Plan

This section outlines how each aspect of the VnV effort will be performed. This may be handled by milestone or by general practices, such as continuous integration and linters.

3.1 Verification and Validation Team

The VnV team consists of four members, each of whom play a distinct role in the verification process. These roles and responsibilities are outlined in Table 1.

Name	Role	Description
Kiran Singh	Lead Developer and Test Designer (LD)	Responsibilities include the identification of critical business cases for integrated tests, assessment of schedule and scope considerations, design and implementation of unit tests, system tests, and documentation of test results.
Matthew Giamou	Project Supervisor (PS)	Lead consultant on integrated performance needs and decomposition for software modules. Responsibilities include review of proposed VnV scope, as proposed by Kiran S., and provision of feedback to the general scope of the VnV campaign, and approval of the individual test cases that are proposed by the LD.
Aliyah Jimoh	Domain Expert (DE)	Responsibilities include provision of feedback on proposed test cases for the scope of test cases for both the functional and non-functional requirements. They will also provide feedback on the code walkthrough. This statisfies the need for a reviewer that is removed from the design effort yet still holds sufficient domain knowledge to provide feedback on the overall design and associated VnV efforts.
Spencer Smith	Software Development Instructor (SDI)	Responsibilities include provision of feedback on proposed test cases for the scope of test cases for both the functional and non-functional requirements. They will also provide feedback via a code walkthrough. This reviewer provides an essential stream of feedback as they are fully removed from the application domain and may willing to question assumptions and address biases that may be implicit amongst other members of the VnV team.

Table 1: Roles and Responsibilities of the Verification and Validation Team

3.2 SRS Verification Plan

The **SRS** shall be reviewed by each member of the reviewer team to form consensus that the SRS has been correctly decomposed into sufficient requirements.

- the models are deemed to be comprehensible

- the models are deemed to be correct

- the associated requirements are traced correctly with respect to the models and project scope

- the requirements are decomposed in a manner that facilitates verification

Feedback on the **SRS** from the DE and SDI will be captured through the use of Github Issues. Specifically, both reviewers will use the **SRS Checklist**. The lead developer will respond in turn to each issue and if reserves the right to reject a proposed change as needed.
The Lead Developer will schedule a meeting with the Project Supervisor to walk through the first revision of the **SRS** document. In this meeting, the Project Supervisor will offer feedback and recommendations for candidate revisions to the outlined models and requirements. The Lead Developer will then create issues in Github to address each proposed revision.

3.3 Design Verification Plan

The DE and SDI will review the Module Guide **(MG)** and the Module Interface Specification **(MIS)** against the **MG Checklist** and **Checklist**. The objective of this review cycle is to ensure that the design of the system:

1. is unambiguous

2. adheres to best-practices of module design

3. aligns with the requirements as identified in the **SRS**

5

3.4 Verification and Validation Plan Verification Plan

The VnV Plan will be verified via inspection by the DE and the Software Development Instructor. The **VnV Plan Checklist** will be used as the assessment criteria for the inspection. Feedback will provided as Github issues and will be handled in the same manner as feedback for the SRS.

3.5 Implementation Verification Plan

The IFC software shall be verified against the test procedures outlined in Sections 4.1 and 4.2. Static verification of the IFC software will consist of a code walkthrough. This will take place on April 4th, 2025, during which the the DE and SDI will have the opportunity to observe the code and raise issues following the presentation via GitHub.

Dynamic verification of the IFC software will consist of system and unit tests via PyTest. System tests are outlined in Section 4. Unit tests will be outlined in Rev 2 of the VnV Plan in Section 5.

3.6 Automated Testing and Verification Tools

Several tools will be used to support automated testing and verification. They include:

- Continuous Integration (CI) will be facilitated via GitHub Actions. A pull request will be used to run automated tests.

- Pytest will be used to execute system tests, unit tests, and to assess code coverage.

- flake8 will be used as a linter to ensure adherence to PEP8 standards.

3.7 Software Validation Plan

Validation testing has been identified as out of scope for the IFCS software.

4 System Tests

This section outlines the general roadmap for the required integrated system tests.

4.1 Tests for Functional Requirements

This section outlines the system tests that verify the requirements outlined in Section 5 of the **SRS**.
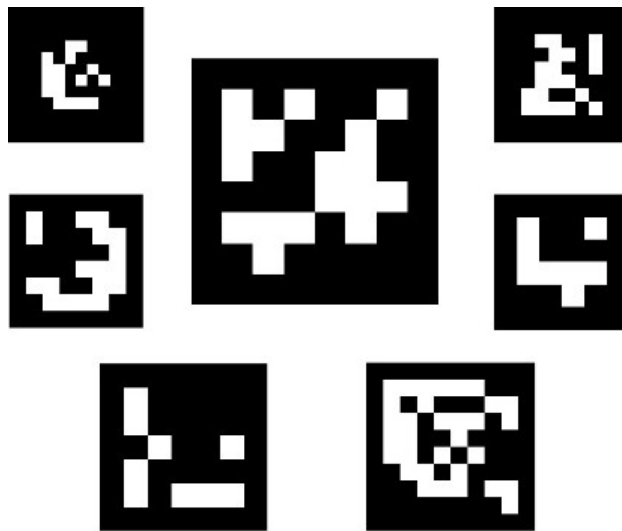

Figure 1: An example of a generated ArUco pattern. Image taken from **?**

4.1.1 Feature Detection

This test section is intended to assess that the system can accept new parameters from users and covers requirements R1 through R7 and R9 through R11, in Section 5.1 of the **SRS**.

Image Smoother

1. **STFR-IS-01**

   Requirements Addressed: R1, R5, R9
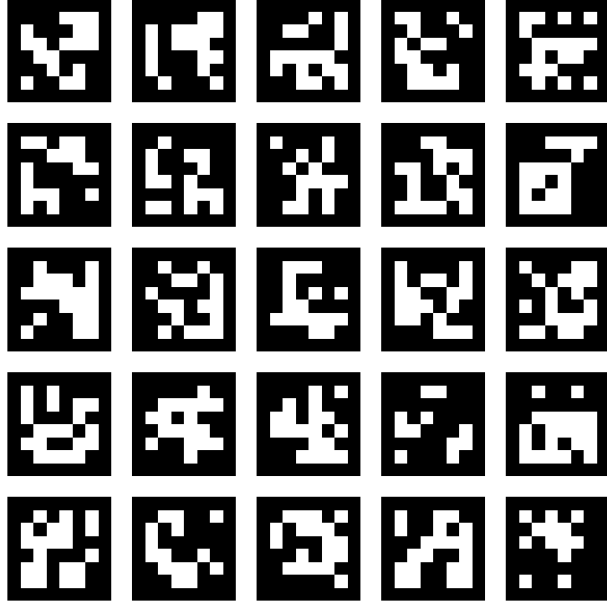
Figure 2: Expected baseline ArUco pattern for functional tests STFR-IS-01, STFR-KP-01, STFR-FD-01, and STFR-FM-01

Control: Automated

Initial State: Uninitialized

Input: Generated binary patterned fiducial markers, named ArUco markers, will be used as simplified targets for corner detection. The script that is used to generate these images is provided in the src code, gen_aruco.py.A depiction of these markers is shown in Figure 1. This script generates an initial image with contains 25 6×6-bit ArUco markers. It uses a seed of 42 to perform scaling, rotational, and translational transforms on the baseline marker. These transforms range between $-45°$ and $45°$, 30 pixels of lateral translation, and a scaling factor between 0.5 and 0.9. A total of 20 transformed images are produced as inputs for the test. Each image I is defined as a 2D image of 600×600 pixel resolution. A Gaussian smoothing kernel of 5 and standard deviation of 1.0 will be used to smooth the edges of the imagery.

Output: 20 output images as a 2D array, each of which have a descrip-

tive name that clearly identifies the input image from which it originates. In this case, all images should use the convention of 'aruco_xxx.png', where 'xxx' represents the enumerated ID of the ArUco marker. The output image, I', should be of equal size and the same data type as its corresponding image I. All images should be saved under a distinct folder to prevent mixing of input and output imagery.

How test will be performed: Automated Tools (i.e. Pytest). If the ArUco imagery is not provided in the src/tests folder, then the user needs to run the gen_aruco.py script to generate the test imagery. Once complete, the user needs to only run STFR-IS-01.py via Pytest while in the src folder to run the test. The pytest file then checks each generated image to confirm that it has the same size and image datatype as the original input image. If each image has a resolution of 600×600 pixels and is an image of type unit8, then the test is considered a pass.

Keypoint Detector

1. **STFR-KP-01**
   Requirements Addressed: R2, R6, R10

   Control: Automatic

   Initial State: Uninitialized

   Input: A collection of 20 smoothed PNG images I' with a resolution of 600×600 that contain the same ArUco markers as outlined in System Test STFR-IS-01. The user assigns a pixel intensity threshold (t) of 60.

   Output: A flag which indicates that corner detection is active. A 2D array of size $k \times m$, where k is the quantity of identified keypoints and the first and second columns are populated with the horizontal and vertical coordinates for each keypoint. Each entry in the array should be rounded to a positive integer value. Any outstanding columns may be populated with associated metadata as a product of the selected method of keypoint detection. For example, oriented FAST (o-FAST) differs from standard FAST in that it also provides an measure of rotation for each keypoint as an additional column. Each array should have a unique title that clearly defines the original image from which

its keypoints were identified. All output arrays should be saved to a unique folder to separate them from the input images.

How test will be performed: Automated Tools (i.e. Pytest)If the smoothed ArUco imagery is not provided in the src/tests folder, then the user needs to run the gen_aruco.py script to generate the test imagery. Once complete, the user needs to only run STFR-KP-01.py via Pytest while in the src folder to run the test. The Pytest file then checks each row of the generated CSV file to confirm that each x and y coordinate is rounded to an integer value. If any keypoint does not contain either an x or y coordinate, a flag is raised and is reported. If all x and y coordinates are rounded to the nearest integer, then the test is deemed to be a pass.

Feature Definition

1. **STFR-FD-01**
   Requirements Addressed: R3, R4, R7, R11

   Control: Automatic

   Initial State: Uninitialized

   Input: A collection of 20 PNG images with a resolution of 600×600, where each image features 25 6×6 ArUco markers within the scene. A $k \times 1$ array will be provided for each image, where k is the number of keypoints within each image. A target bin size (d_{total}) of 100 will be used to define the target number of feature descriptors. A patch size (p) of 31 will be used to define the adjacent region of a keypoint for which to define descriptors.

   Output: D_{bin}, an array of 32 byte binary values. The width of this array may be increased if associated horizontal and vertical coordinates for the corresponding keypoint may be affixed to each descriptor.

   Test Case Derivation: ORB: An efficient alternative to SIFT or SURF. This publication outlines methods to develop and the corresponding assessment of binary feature descriptors.

   How test will be performed: Automated Tools (i.e. Pytest). If the smoothed ArUco imagery is not provided in the src/tests folder, then the user needs to run the gen_aruco.py script to generate the test

imagery. Once complete, the user needs to only run test_STFR-FD-01.py via Pytest while in the src folder to run the test. The Pytest file then checks each row of the generated CSV file to confirm that each descriptor has a horizontal and vertical coordinate, rounded to the nearest integer. All 256 bits of the binary descriptor are then checked to ensure that there is no data bit loss from conversion to CSV. If associated horizontal or vertical coordinates are identified as missing, then the test fails. If there is any loss of databit within the 32 byte descriptor, then the test fails. If all identified descriptors pass both the coordinates and descriptor bitwise checks, then the test is defined to be a pass. D_{bin} should be within 10% of the target number of descriptors, d_{total}.

4.1.2 Feature Comparison

This test section is intended to assess that the system can accept new parameters from users. It addresses requirements R8 and R12 through R15 in Section 5.1 of the **SRS**.

Descriptor Comparison

1. **STFR-FM-01**
   Requirements Addressed: R8, R12, R13, R14, R15

   Control: Automated

   Initial State: Uninitialized

   Input: 20 descriptor arrays of size $D_{bin} \times 4$, where each row represents a feature descriptor, with the horizontal and vertical coordinates, descriptor ID, and image of orgin as the columns of the array. Each descriptor ID is a 32 byte binary string known as a binary descriptor. Each array should be labeled with a descriptive name that clearly references the instance of both the camera and pose at the time of image capture. For this test, set $D_{bin} = 100$ for each descriptor array. A maximum of 30 match candidates will be permitted with a maximum permissible Hamming Distance of 25.

   Output: A flag that indicates that Hamming Distances were compared. A dynamically-sized array, stored as a Pandas dataframe, where

11

the length of the array is the number of matched features. The columns of the array will include the properties as follows.

- Binary descriptors of both features as 32 byte binary strings
- Corresponding image IDs for each feature
- Matching scores between both features
- The Hamming Distance of each candidate match between features

Note that we do not compare the identified Hamming Distance as there is no baseline Hamming Distance for which to compare.

How test will be performed: Automated Tools (i.e. Pytest) If the smoothed ArUco imagery is not provided in the src/tests folder, then the user needs to run the gen_aruco.py script to generate the test imagery. Once complete, the user needs to only run test_STFR-FM-01.py via Pytest while in the src folder to run the test. The Pytest file then checks each row of the generated CSV file to confirm that for both pairs of descriptors that compose a candidate match, each descriptor has a horizontal and vertical coordinate, rounded to the nearest integer. All 256 bits of the binary descriptor are then checked to ensure that there is no data bit loss from conversion to CSV. Finally, the calculated distance between each descriptor is assessed to confirm that it is a positive integer. If associated horizontal or vertical coordinates are identified as missing, then the test fails. If there is any loss of databit within the 32 byte descriptor, then the test fails. If the distance between descriptors is not a positive integer, the test fails. If all identified descriptors pass both the coordinates and descriptor bitwise checks, and the distance is a positive integer, then the test is defined to be a pass.

4.2 Tests for Nonfunctional Requirements

4.2.1 Reliability

The IFC software is envisioned to be used as part of the front end of a larger camera calibration pipeline, where the outputs of the IFC software need to produce consistent results for use in a back-end optimization module. Therefore, the IFC software needs to undergo verification to ensure that it

can match the same features between two images. The following system test satisfies **NFR1** of the SRS.

Repeatability

1. **STNFR-RE-1**
   Type: Dynamic

   Initial State: Uninitialized

   Input/Condition: Two datasets of imagery data. These datasets contain the same collection of images, but are arranged in a different order.

   Output/Result: The feature matching pipeline will be use to identify features and compare them amongst images for both datasets. For both datasets, a dynamically-sized array will be output. These arrays will contain the feature descriptors and the image IDs of origin, as specified in as specified in STFR-DC-01. One array will be reordered to reflect the order of the other array. Once the second array has been reordered, the arrays will be compared in terms of their length and contents. For example, in images A and B, they will be processed under the pseudonym of B' and A', respectively. That is, for each feature of A, its descriptor and x-y coordinates are equivalent to those of B'. Similarly, the coordinates and descriptor of B are equivalent to A'. The features of A and B will be compared to identify matches. Then the features of A' and B' will be compared as well. The test will be considered a pass if the identified matches between A and B share the same descriptors and coordinates as the identified matches between B' and A', respectively.

   How test will be performed: PyTest. Two sample images from the Lego library have been duplicated and reordered with new names. Paths to the original and transformed images are outlined in the test_STFR-RE-01.py file. A script will assess that for the corresponding transformed image, that the same coordinates and 32-byte feature descriptor are identified as a match candidate with the same coordinates and 32-byte descriptor of the transformed image for the complementary image.

4.2.2 Usability

As part of a larger pipeline, the IFC software needs to be simple for its users to integrate into the calibration system. This may be reflected in the perception of how simple it is for the user to implement its on their own system and to run the program as its own module. This usability criterion is reflected in **NFR2** of the **SRS**.

User Experience: End-to-End Feature Matching

1. **STNFR-UX-1**
   Type: Dynamic

   Initial State: Uninitialized

   Output/Result: A dataframe of identified features, as outlined in STFR-DC-01. Once the test has been completed, the user will be asked to express their opinion of their experience of the system, and to remark on the following topics.

   - the degree of perceived difficulty in preparing the input files
   - the degree of perceived difficulty in assignment of the camera properties
   - the user's overall satisfaction with using the program

   How test will be performed: Dynamic Test by User

The characteristics of the desired user are outlined in the **SRS**. However, for the intent of testing, the user may be defined as the Project Supervisor or Domain Expert. The user will be provided a collection of images from different cameras. Each image will be labelled with a descriptive title that clearly defines what camera the image originates from. The user shall be responsible to insert all images into a common folder.

   The user shall execute the following procedure in the following order.

1. The user shall open the directory to the IFC configuration file.

2. In the IFC program configuration file, the user shall:

   (a) Update the input location as the folder of all the input images.

(b) Update the system directory with the desired location of the matched feature array.

(c) Assign a directory for altered imagery, if desired.

(d) Input the number of cameras per the sized camera data.

(e) Provide the following details for each camera:

  i. A descriptive name of the camera that matches the prefix of the name of its imagery data.

  ii. The resolution of each camera.

3. The user shall close the IFC configuration file.

4. The user initiates execution of the feature comparison pipeline.

5. The user waits for the image comparison process to complete.

6. After completion, the user opens the output array directory.

7. The user reviews the output array to verify that there are no errant characters.

8. The user closes the output array.

Though not within the current scope of the VnV plan, a separate test may be performed where the user performs the same steps as STNFR-UX-1, and additionally modifies the Gaussian Kernel standard distribution, intensity threshold, and patch size parameters to values within the allowable operational boundaries.

4.2.3 Maintainability

Verification of **NFR3**, outlined in **SRS**, has been delegated as future work in favour of other verification procedures outlined within this document. This tradeoff was made to satisfy the schedule and developer resource constraints of the Winter 2025 development cycle. It would however, be simple to evaluate if a new detection was to be implement. Once the default feature detection method has successfully been completed, the total number of hours spent to design the feature detection module should be summed together. A factor of 0.3 should be applied to the summed effort to define the maximum

target effort to develop a new method of feature detection. Any time that is dedicated to implement a new method should be added to the running total. A scenario where the total effort exceeds the allowable target may suggest that the current IFC software is difficult to maintain.

4.2.4 Performance

One of the primary objectives of the VnV campaign is to characterize what system resources are required to execute the IFC software, as outline in **NFR4** and **NFR5**. In the case of a simple configuration with only two cameras at two different poses, the process is trivial. However, for configurations that consist of as many as ten cameras across dozens of robot poses, the problem may scale significantly. Therefore, as a general practice, timing and memory usage metrics should be recorded throughout the duration of any of the primary operations of the IFC software. This includes the following system tests in Section 4.1.

1. STFR-IS-01

2. STFR-KP-01

3. STFR-FD-01

4. STFR-DC-01

Timing Metrics
The timestamp will be recorded at the start of each of the major operations (i.e. Image Smoothing, Keypoint Detection, Assignment of Feature Descriptors, and Feature Comparison). A subsequent timestamp will be recorded after the termination of each operation. The difference between the start and termination timestamps will be saved and stored in a structured format (e.g., JSON or CSV) within a designated log directory.

Memory Usage Metrics
The memory usage of the IFC software should be logged with a timestamp and operation identifier and stored in a structured format (e.g., JSON or CSV) within a designated log directory. This data may also be augmented with a separate file that outlines the average memory usage, and the peak memory usage with associated timestamps.

4.3 Traceability Between Test Cases and Requirements

The traceability between each scoped test case and its respective requirements is outlined by 2.

	STFR-IS-01	STFR-KP-01	STFR-FD-01	STFR-FM-01	STNFR-RE-1	STNFR-UX-1
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	o	o	o	o		
NFR5	o	o	o	o		

Table 2: Traceability Matrix Showing the Connections Between Requirements and Instance Models

- 'x' indicates a direct method of verification

- 'o' indicates an indirect method of verification

17

- * Verification of NFR3 is defined as outside of scope per Section 4.2.3

5 Unit Test Description

This section describes the unit tests developed for the Image Feature Correspondences (IFC) system. The tests validate that the system conforms to the functional and nonfunctional requirements specified in the SRS and implemented per the design in MIS. The unit tests are executed using the `pytest` framework and primarily assess correctness, parameter bounds, default behavior, and file outputs.

The test strategy includes both black-box and white-box testing:

- **Black-box testing** ensures that each module produces expected results given a specific input.

- **White-box testing** checks internal method logic, parameter validation, and error handling.

All modules have unit tests verifying both normal and edge case behavior. Each test function name corresponds to its functionality and is self-documenting.

5.1 Unit Testing Scope

Testing of the OpenCV Module itself is considered out of scope.

5.2 Tests for Functional Requirements

Note that the unit tests are not outlined in the numerical order of the modules (i.e. M1, M2, etc.) as it is intended to reflect a gradual increase in complexity of unit testing up to the **Control Module (M2)**. A full breakdown of the unit tests can be found in the **test** folder of the Github Repository.

5.2.1 Module: Specification Parameters Module (M4)

This module is responsible for housing the script that enables the users to configure the methods of image processing will be used in the pipeline and to adjust the respective tuning parameters for each method.

1. `test_kernel_bounds` (from `test_specParams.py`).

   Type: Unit, Automatic

   Initial State: A predefined kernel value `test_k` and kernel bounds `lim_kern_bounds = [3, 15]`.

   Input: An integer representing a Gaussian kernel size.

   Output: Assertion pass/fail depending on whether the kernel size is odd and within the inclusive bounds.

   Test Case Derivation: This test is a simple check to ensure that the select kernel size is compatible with Gaussian filtering operations.

   How test will be performed: Implemented via automated Pytest script and Github Actions. Checks that `test_k` is of type `int`, odd-valued, and lies within the bounds.

2. `test_standard_deviation` (from `test_specParams.py`).

   Type: Unit, Automatic

   Initial State: A defined standard deviation value `test_sigma` and limits `lim_sd_bounds = (0, 10]`.

   Input: A float or integer representing Gaussian standard deviation.

   Output: Assertion pass/fail if the value is not in $(0, 10]$, accounting for floating-point precision.

   Test Case Derivation: Supports **R01**, validating continuous parameter constraints for Gaussian blur.

   How test will be performed: Automated script via Pytest and GitHub Actions. Type-checked and validated using an epsilon margin for numerical stability.

3. `test_fast_bounds` (from `test_specParams.py`).

   Type: Unit, Automatic

   Initial State: FAST detection threshold `test_t` and intensity limits `lim_fast_bounds = [2, 254]`.

Input: An integer intensity threshold value.

Output: Boolean result from assertion checking bounds compliance.

Test Case Derivation: Supports **R02**, ensuring intensity threshold parameterization for corner detection.

How test will be performed: Automated script via Pytest and GitHub Actions. The test asserts that the value lies within bounds and is an integer.

4. `test_bin_bounds` (from `test_specParams.py`).

   Type: Unit, Automatic

   Initial State: Descriptor bin count `test_b`, constrained by `lim_bin_bounds` = [1, 2048].

   Input: Integer representing number of ORB descriptors to retain.

   Output: Passes if within bounds, otherwise assertion error is raised.

   Test Case Derivation: Enforces **R04**, which governs descriptor configuration consistency.

   How test will be performed: Automated script via Pytest and Github Actions. Type and range checks performed on the bin size.

5. `test_patch_size_bounds` (from `test_specParams.py`).

   Type: Unit, Automatic

   Initial State: Descriptor patch size `test_p` bounded by `lim_patch_sz` = [5, 100].

   Input: Integer specifying the spatial extent for descriptor extraction.

   Output: Assertion checks for compliance with defined patch size range.

   Test Case Derivation: Related to **R03**, controlling spatial descriptor characteristics.

   How test will be performed: Automated script via Pytest and Github Actions. The patch size is verified to be an integer and within range.

6. `test_avail_methods` (from `test_specParams.py`).

   Type: Unit, Automatic

   Initial State: Lists of available methods defined in `SpecificationParametersModule.py`: `mthd_is`, `mthd_kpd`, `mthd_fd`, `mthd_ftm`.

   Input: None (test inspects module-level variables).

   Output: Assertion passes if each method list is a non-empty list or tuple.

   Test Case Derivation: Supports **R06**, **R07**, and **R08** to ensure at least one method is implemented for each stage of the pipeline.

   Test Cases:

   - All lists are non-empty and of valid types (expected outcome: pass)
   - `mthd_is = []` (expected outcome: fail)
   - `mthd_fd = "ORB"` (expected outcome: fail)

   How test will be performed: Automated script via Pytest and GitHub Actions. Module is imported and assertions are applied directly to global method lists without requiring user input.

7. `test_selected_methods` (from `test_specParams.py`).

   Type: Unit, Automatic

   Initial State: User-selected method identifiers from `SpecificationParametersModule.py`: `mthd_img_smoothing`, `mthd_kp_detection`, `mthd_kp_description`, `mthd_ft_match`.

   Input: Enumerated integer selections for each image processing stage.

   Output: Assertion passes if all values are integers and ≥ 0.

   Test Case Derivation: Automated script via Pytest and GitHub Actions. Supports ,**R06, R07, R08**, ensuring valid configuration and preventing undefined method selection.

   Test Cases:

   - All values are valid integers ≥ 0 (e.g., 1, 2, 3; expected outcome: pass)

21

- `mthd_img_smoothing = -1` (expected outcome: fail)
- `mthd_ft_match = "brute_force"` (expected outcome: fail)

How test will be performed: Automated script via Pytest and GitHub Actions. Module-level configuration parameters are evaluated and asserted for type and range correctness.

5.2.2 Module: Input Format Module (M3)

This module is responsible for validating user inputs, setting defaults, and enforcing parameter bounds.

1. test_check_parameter_limits_valid

   Type: Functional, Automatic

   Initial State: All parameters, including standard deviation $\sigma = 1.5$, are within valid bounds.

   Input: $\sigma = 1.5$

   Output: No assertion error; standard deviation value accepted.

   Test Case Derivation: Satisfies **R01** where user input for σ falls within allowable range $(0, 10]$.

   How test will be performed: Automated via Pytest and GitHub Actions using parameterized validation checks in `check_parameter_limits()`.

2. test_check_parameter_limits_valid

   Type: Functional, Automatic

   Initial State: Threshold $t = 100$

   Input: $t = 100$

   Output: No assertion error; threshold value accepted.

   Test Case Derivation: Address **R02** where the user-defined input for FAST threshold lies within $[2, 254]$.

   How test will be performed: Automated via Pytest and GitHub Actions using valid parameter combination.

3. test_check_parameter_limits_valid

   Type: Functional, Automatic

   Initial State: Patch size = 20

   Input: Patch size = 20

   Output: No assertion error; patch size is accepted.

   Test Case Derivation: Addresses **R03** to confirm that patch size lies within range [5, 100] and is integer.

   How test will be performed: Automated script via Pytest and GitHub Actions using structured parameter validation.

4. test_check_parameter_limits_valid

   Type: Functional, Automatic

   Initial State: Bin size = 128

   Input: bin size = 128

   Output: No assertion error; bin size accepted.

   Test Case Derivation: Addresses,**R04** to confirm that the bin size lies within [1, 2048].

   How test will be performed: Automated script via Pytest and GitHub Actions by calling `check_parameter_limits()` with all valid values.

5. test_check_method_limits_valid

   Type: Functional, Automatic

   Initial State: A mock list of available methods for each processing stage, [2, 3, 2, 2]

   Input: (img, kp, desc, match) = (1, 2, 1, 2)

   Output: No assertion errors are raised; method selections are accepted as valid.

   Test Case Derivation: Ensures that selected method indices lie within allowed ranges (e.g., $\leq$ max available methods for each category). Supports defaults and user inputs for **R6–R8**.

23

How test will be performed: Automated via Pytest using monkeypatching to mock method availability. Confirms the system accepts enumerated method inputs within valid bounds.

6. test_check_method_limits_invalid

   Type: Functional, Automatic

   Initial State: A mock method list [2, 3, 2, 2] is returned by the system, limiting valid index ranges.

   Input: Various invalid combinations such as:

   - (img = -1), (kp = 4), (desc = 3), (match = -1)

   Output: `AssertionError` is raised when any input exceeds its allowed range or is negative.

   Test Case Derivation: Verifies that invalid enumerations are properly rejected, preventing undefined behavior in the processing pipeline. Supports enforcement of valid method selection for **R6–R8**.

   How test will be performed: Automated script via Pytest and GitHub Actions using monkeypatched configuration. Each input case is run individually and expected to raise an `AssertionError`.

5.2.3   Module: Output Format Module (M5)

These tests confirm that keypoints, descriptors, and matches are saved to CSVs correctly and scale with variable input sizes.

1. test_output_keypoints_variable_size

   Type: Functional, Automatic

   Initial State: The system is provided with a list of synthetically generated keypoints associated with a known image identifier. The number of keypoints is varied across test cases (1, 100, 1000).

   Input: A list of `cv::KeyPoint` objects representing features extracted from a synthetic image.

Output: A CSV file containing one row per keypoint, with all required metadata fields: `x`, `y`, `size`, `angle`, `response`, `octave`, and `class_id`.

Test Case Derivation: This test verifies that the software fulfills **R10** by correctly encoding and exporting keypoints that have been detected per system thresholds.

How test will be performed: An automated Pytest script calls the keypoint output routine, checks that the resulting file exists, confirms the expected number of rows, and asserts that the metadata fields are all present. Execution is handled via GitHub Actions.

2. test_output_descriptors_variable_size

   Type: Functional, Automatic

   Initial State: The system is supplied with a list of synthetic `cv::KeyPoint` objects and a NumPy array of binary descriptors corresponding to a known image identifier. The number of keypoints and descriptors is varied across trials (1, 100, 1000).

   Input: A collection of keypoints and their corresponding ORB descriptors, encoded as 32-element `uint8` arrays.

   Output: A descriptor CSV file that accurately records the number of descriptors and includes all required descriptor metadata fields.

   Test Case Derivation: This test confirms that the IFC software fulfills **R11** by exporting a consistent, valid representation of keypoint descriptors derived from an image frame.

   How test will be performed: An automated Pytest script executes the descriptor output routine, checks for the existence of the CSV file, and validates both the row count and required columns. The test is deployed via GitHub Actions.

3. **test_output_matches_variable_size** (from `test_outputFormat.py`).

   Type: Functional, Automatic

   Initial State: Valid descriptors and matches for two synthetic images.

   Input: Keypoint pairs and matched descriptor arrays for $n = 1000$ features.

Output: A properly formatted CSV file with the correct number of rows and labeled fields.

Test Case Derivation: Satisfies **R14** and **R15**, which require the software to report unique correspondences and include camera/frame identifiers.

How test will be performed: Matches between descriptor sets are synthesized and passed to the output module. The resulting CSV file is parsed and validated for completeness.

5.2.4 Module: Image Plotting Module (M11)

Tests validate that keypoints and feature matches can be drawn to image composites and that directories are created when needed.

1. test_gen_kp_img_with_none_keypoints

   Type: Functional, Automatic

   Initial State: A blank image is provided, and no keypoints are passed to the plotting function.

   Input: A 3-channel grayscale image (all zeros) and `None` for keypoints.

   Output: The function returns a valid image with the same shape as the input.

   Test Case Derivation: Confirms that keypoint visualization gracefully handles empty keypoint input without error.

   How test will be performed: Pytest executes the image rendering function and asserts that the result is a valid NumPy array of expected dimensions.

2. test_gen_kp_img_no_flag

   Type: Functional, Automatic

   Initial State: A small image and a list containing one keypoint.

   Input: A grayscale image and a list of `cv::KeyPoint` with no OpenCV draw flags.

   Output: An image with the keypoint drawn in the default style.

Test Case Derivation: Ensures default rendering path functions correctly for basic keypoint visualization.

How test will be performed: Pytest evaluates whether a valid image of the same shape is returned, confirming that drawing occurred without error.

3. test_gen_kp_img_rich_keypoints

   Type: Functional, Automatic

   Initial State: A grayscale image and a keypoint are passed, with the rich keypoint visualization flag enabled.

   Input: One keypoint and `cv::DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS`.

   Output: An annotated image with detailed keypoint rendering (e.g., size, orientation).

   Test Case Derivation: Verifies that the rich visualization mode of OpenCV is supported and produces valid output.

   How test will be performed: Pytest confirms output is a valid NumPy array and has the same spatial shape as the input image.

4. test_gen_kp_img_with_none_image

   Type: Functional, Automatic

   Initial State: No image is passed to the visualization function.

   Input: `None` as the input image and an empty list of keypoints.

   Output: An OpenCV error is raised, indicating that image input is required.

   Test Case Derivation: Ensures input validation is enforced and invalid image input is handled via exception.

   How test will be performed: Pytest uses `raises()` context to assert that a `cv::error` is thrown.

5. test_gen_matched_features_success

   Type: Functional, Automatic

Initial State: Two synthetic images, a pair of keypoints, and one match are provided.

Input: Two images, keypoint sets, and a `cv::DMatch` object.

Output: A composite image visualizing the matched keypoints.

Test Case Derivation: Confirms that the software can render visual correspondence between two keypoints using OpenCV's drawing routines.

How test will be performed: Pytest checks the output image is valid and has a height greater than zero (indicating combined image was generated).

5.2.5 Module: Image Smoothing Module (M7)

Gaussian filtering is tested for correct kernel size, standard deviation, and type safety.

1. `test_smooth_image_valid_gaussian` (from `test_imagesmooth.py`).

   Type: Functional, Automatic

   Initial State: A synthetic noisy grayscale image.

   Input: Method index = 1; kernel size = 5; standard deviation $\sigma = 1.0$.

   Output: A smoothed image with unchanged dimensions and type.

   Test Case Derivation: Satisfies **R9**, which requires the system to perform noise reduction based on user-specified standard deviation.

   How test will be performed: Random noise is applied to a grayscale image, Gaussian smoothing is applied, and differences from the input are confirmed.

5.2.6 Module: Keypoint Detection Module (M8)

Keypoint detection via ORB is tested for valid configurations, disabled methods, and behavior with missing components.

1. `test_detect_keypoints_with_valid_orb` (from `test_kpdetect.py`)

   Type: Functional, Automatic

   Initial State: A grayscale image and a configured ORB object.

Input: ORB method index = 1; input image of shape 100×100.

Output: A list of keypoints.

Test Case Derivation: According to **R10**, the system must detect keypoints using the provided image and method.

How test will be performed: Automated script via Pytest and GitHub Actions using structured parameter validation. ORB is initialized and applied to a blank image. It verifies the returned keypoints are instances of `cv2.KeyPoint`.

5.2.7 Module: Feature Descriptor Module (M9)

1. `test_compute_descriptors_valid` (from `test_featdesc.py`)

   Type: Functional, Automatic

   Initial State: A grayscale image and a set of detected keypoints.

   Input: Method index = 1; keypoints detected from an ORB detector.

   Output: A tuple (keypoints, descriptors), where descriptors is a NumPy array.

   Test Case Derivation: According to **R11**, the system must generate valid descriptors from previously identified keypoints.

   How test will be performed: Automated script via Pytest and GitHub Actions using structured parameter validation. Random image data is used to generate keypoints. These are passed to the descriptor method, and the output types and dimensions are validated.

5.2.8 Module: Feature Matching Module (M10)

Tests ensure descriptors from different images are correctly matched using brute-force Hamming distance and validated for 1-to-1 correspondence.

1. `test_match_features_no_loss` (from `test_featmatches.py`)

   Type: Functional, Automatic

   Initial State: Identical descriptors from two images.

   Input: Descriptor arrays `desc1` and `desc2` (identical), and a matcher configured for Hamming distance.

Output: A list of matches, one per descriptor.

Test Case Derivation: Satisfies **R12**, ensuring all descriptors from matching images are paired correctly when descriptors are identical.

How test will be performed: Automated script via Pytest and GitHub Actions using structured parameter validation. Two identical descriptor sets are matched with a brute-force matcher using cross-checking. One-to-one matching is expected.

5.2.9 Module: Output Verification Module (M6)

These tests are intended to address **R13**.

1. test_check_match_uniqueness_valid

   Type: Functional, Automatic

   Initial State: Two distinct image IDs are provided, with an empty match list.

   Input: `img_query = "img001"`, `img_train = "img002"`, and `matches = []`.

   Output: The function returns `True` with no warnings.

   Test Case Derivation: Satisfies **R13** by ensuring that valid matches between distinct images pass silently, confirming proper inter-image correspondence.

   How test will be performed: Pytest executes the uniqueness check and asserts that the return value is `True` and no warning is issued.

2. test_check_match_uniqueness_warns_for_same_ids
   Type: Functional, Automatic

   Initial State: Two identical image IDs are passed as both query and train identifiers, with an empty match list.

   Input: `img_query = "img001"`, `img_train = "img001"`, `matches = []`.

   Output: The function returns `True` and raises a `UserWarning` about non-unique image match.

Test Case Derivation: Satisfies **R13** by confirming that the system detects invalid intra-image match attempts and flags them appropriately.

How test will be performed: Pytest asserts that a `UserWarning` is raised and that the return value remains `True` for compatibility.

3. test_check_match_uniqueness_warns_with_matches

   Type: Functional, Automatic

   Initial State: Two identical image IDs are passed along with a list of dummy matches.

   Input: `img_query = "imgA"`, `img_train = "imgA"`, `matches = [match1, ..., match5]`.

   Output: A `UserWarning` is issued despite the presence of match data.

   Test Case Derivation: Reinforces **R13** by demonstrating that even non-empty match lists are flagged if the query and train image IDs are not unique.

   How test will be performed: Pytest checks for the expected warning condition using `pytest.warns()` and validates that the warning mechanism functions regardless of match list content.

5.2.10   Module: Control Module (M2)

Tests the end-to-end IFC pipeline using synthetic image inputs. This includes validation of intermediate image outputs (grayscale and smoothed), feature detection, descriptor generation, match computation, and their respective CSV and image outputs.

1. `test_pipeline_synthetic_images` (from `test_main.py`)

   Type: Functional, Automatic

   Initial State: Two synthetic RGB images created with distinct geometric features.

   Input: Two test images (`synthetic_01.png`, `synthetic_02.png`) and a patched configuration with default parameters.

   Output:

- Grayscale images in `gsImagery/`

- Smoothed images in `gkImagery/`

- Keypoints in `kpDetection/*.csv` and `*.png`

- Descriptors in `fDescriptors/*.csv` and `*.png`

- Matches in `fMatches/*.csv` and `*.png`

Test Case Derivation: Addresses **R10** through **R12**, ensuring that all required outputs of the image processing pipeline are generated and correctly formatted.

How test will be performed: Automated script via Pytest and GitHub Actions using structured parameter validation. The full pipeline is executed on two synthetic images that are used as inputs, with defaults of the system methods and parameters being used. The test asserts that all expected files are created and accessible in their respective output directories. Filenames and structures are verified using deterministic patterns derived from the test inputs.

5.3 Tests for Nonfunctional Requirements

5.3.1 Control Module (M2)

1. test_timing_metrics_recorded

   Type: Functional, Dynamic, Automatic

   Initial State: The IFC system is configured to execute the full image processing pipeline (grayscale conversion, smoothing, keypoint detection, descriptor extraction, matching), with metric logging enabled.

   Input: A small batch of synthetic images with known dimensions and content.

   Output: A structured log (e.g., CSV or JSON) reporting elapsed time (in milliseconds) for each of the following: grayscale conversion, smoothing, keypoint detection, descriptor extraction, and matching.

   Test Case Derivation: This test ensures the software satisfies **FR4** by confirming that each stage of the pipeline logs its processing time explicitly.

How test will be performed: Automated test script wraps each pipeline call with a timing logger (e.g., `time.perf_counter()`), collects the reported values, and asserts presence, validity (non-negative), and uniqueness of timing data for each processing stage. Deployed via Pytest and validated under CI with GitHub Actions.

2. test_memory_metrics_recorded

Type: Non-Functional, Dynamic, Automatic

Initial State: The system executes a full image processing task with memory profiling tools enabled (e.g., `tracemalloc` or `psutil`).

Input: A single grayscale image passed through all processing stages (smoothing, feature detection, description, and matching).

Output: A report containing average and peak memory usage (in kilobytes or megabytes) for each processing operation.

Test Case Derivation: Ensures compliance with **NFR5** by verifying that memory usage statistics are gathered and reported correctly.

How test will be performed: The test is run with a memory profiling context around each major function. After execution, the test checks that memory metrics exist for each stage, are numeric, and include both mean and peak usage values. Can be executed in Pytest with plugins such as `pytest-monitor` or custom logging using `psutil`.

5.4   Traceability Between Test Cases and Modules

A full outline of the traceability between the unit tests and the modules as outlined in the MIS is provided in Table 3.

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

6   Appendix

Not Applicable.

6.1   Symbolic Parameters

No symbolic constants have been identified in this document.