APRIL TAG AND CALIBRATION DETECTORS – GSOC 2025

Name and Contact Information

Name: Aditya Kumar
Discord Username: AS1100K
GitHub Username: AS1100K
Phone: +91 [redacted]
Email: [redacted]
Country: India
Time Zone: IST (GMT +5:30)
Resume: https://adityais.dev/resume.pdf
Website: https://adityais.dev
LinkedIn: linkedin.com/in/as1100k

Title

April Tag and Calibration Detectors

Synopsis

April Tags are widely used in robotics for precise localization, camera calibration, and augmented reality applications. Their ability to provide robust fiducial markers makes them valuable for tasks such as robot navigation, visual SLAM, and pose estimation. This project aims to implement efficient detection algorithms for both synthetic and natural features in Rust. It will include an April Tag detector—optimized for calibration and structured environments—and an ORB-based detector for natural feature extraction, which is crucial for robust visual localization and Visual SLAM applications. Both modules will be integrated into the kornia-rs ecosystem, thereby providing a comprehensive toolkit for camera calibration, localization, and robotics.

Why Kornia-rs?

There are a few reasons why I choose kornia-rs as my possible mentor open-source organization. I have always been interested in Robotics and Computer Vision. I always find research papers explaining the core fundamentals of algorithms used in Computer Vision. Secondly, kornia-rs is a low-level 3D Computer Vision library, it aligns with my interest in writing efficient low-level code.

Deliverables

In this section, I list the milestones that I'd like to achieve in the summer. Broadly speaking, the final deliveries should look like:

- A *kornia-apriltag* crate that supports efficient April Tag detection.
- Various tests that will ensure everything is working as expected.
- Benchmark against OpenCV, Nvidia VPI and Aprilgrid-rs.
- Implement highly customizable and efficient Camera Calibration.
- Implementation of an ORB-based feature detector and benchmarking it with OpenCV and Nvidia VPI.
- Work together with *Hauke Strasdat (author of sophus-rs)* to show an integration of a calibration system using the April Tag detector and/or using FAST features for a Visual SLAM implementation.
- Hopefully also add support for GPU by coordinating with Contributor of *"Implement GPU image transforms"* and the mentor.
- Add a simulation-based example using kornia, copper, and bevy to demonstrate *kornia-apriltag* along with camera calibration.

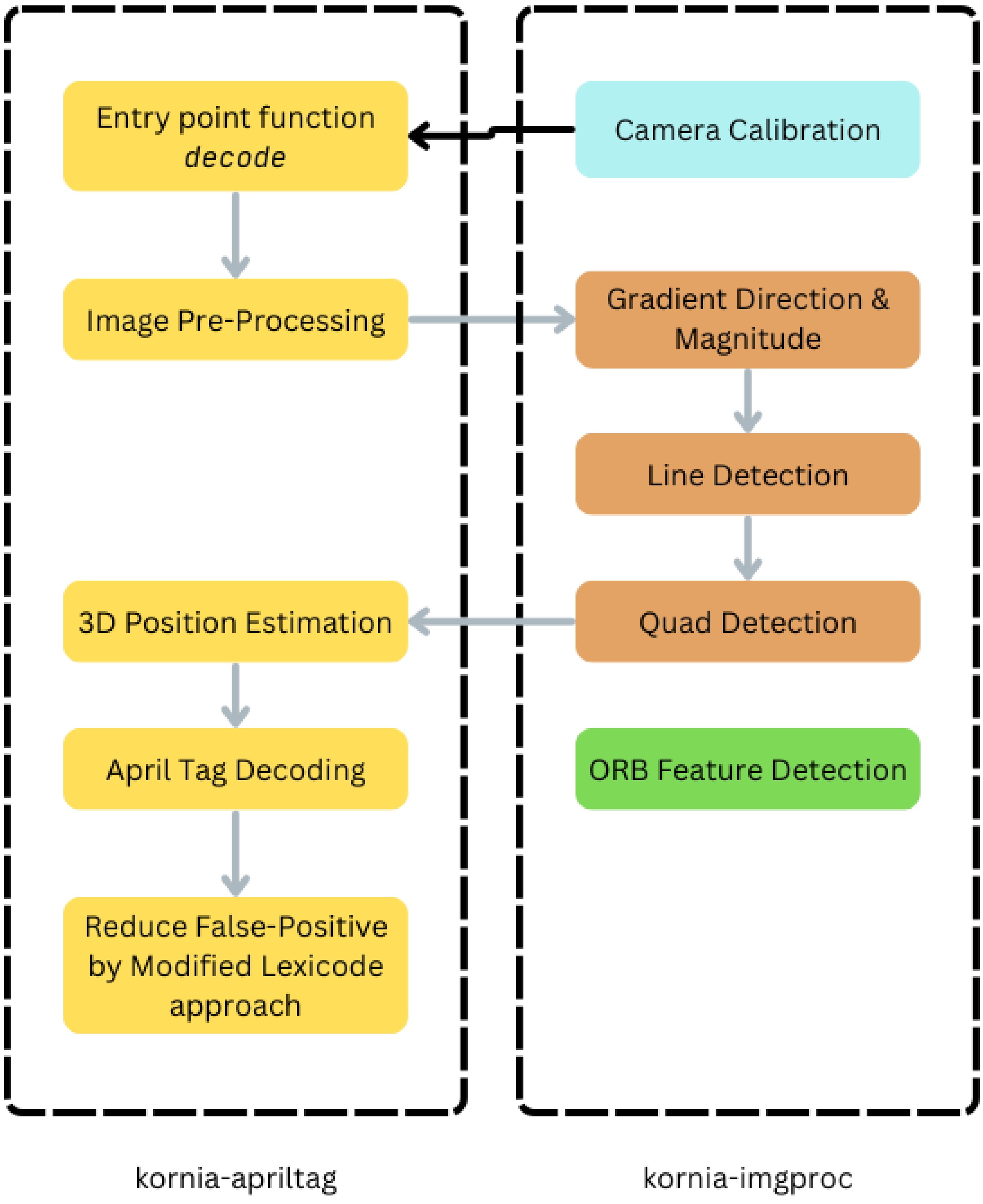
API Design Proposal

In this section, I propose the API structure for *kornia-apriltag* crate along with Camera Calibration and Visal SLAM implementation that ensure it to be efficient, highly customizable and utilise existing crates of kornia-rs repositories.

Table of Contents:

High Level Overview:

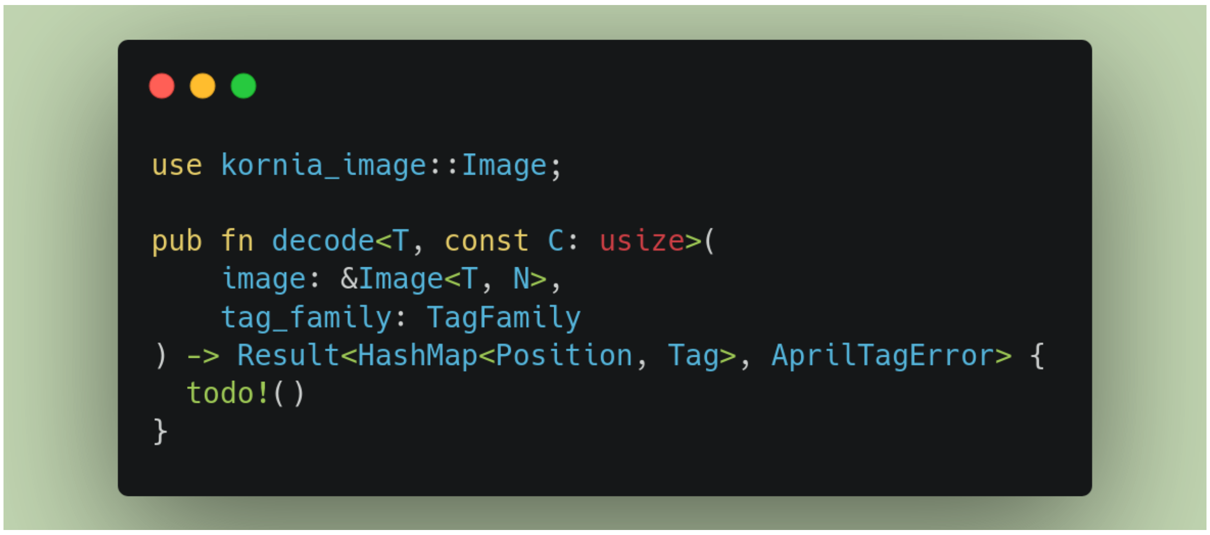
The below image shows a high-level overview of how this project will work.


Entry point function *decode*
Camera Calibration
Image Pre-Processing
Gradient Direction & Magnitude
Line Detection
3D Position Estimation
Quad Detection
April Tag Decoding
ORB Feature Detection
Reduce False-Positive by Modified Lexicode approach
kornia-apriltag
kornia-imgproc

Technical Overview (April Tag Detection):

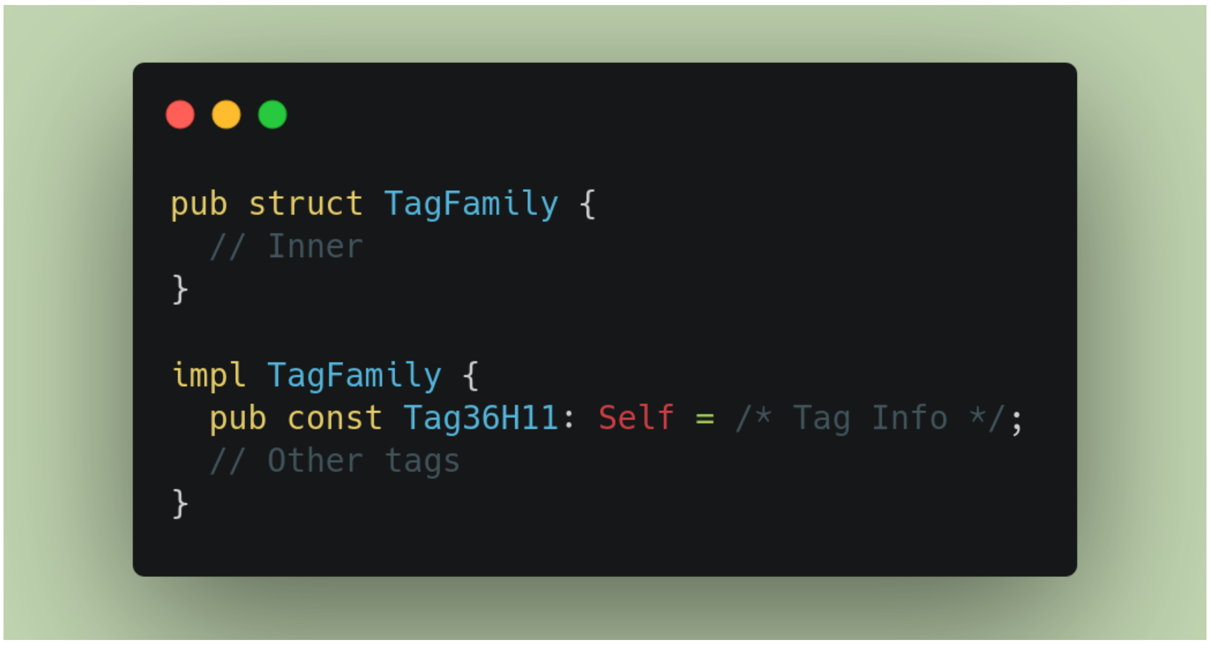
In this section, I will go through some API design decisions that I propose and why.

The entry point of this crate will be *decode* function that will take two parameters, 1) *image* with C number of color channels and 2) *tag_family* which will contain information regarding which tag should be scanned and decoded. The following is the proposed syntax of *decode* function:



```rust
use kornia_image::Image;

pub fn decode<T, const C: usize>(
    image: &Image<T, N>,
    tag_family: TagFamily
) -> Result<HashMap<Position, Tag>, AprilTagError> {
  todo!()
}
```

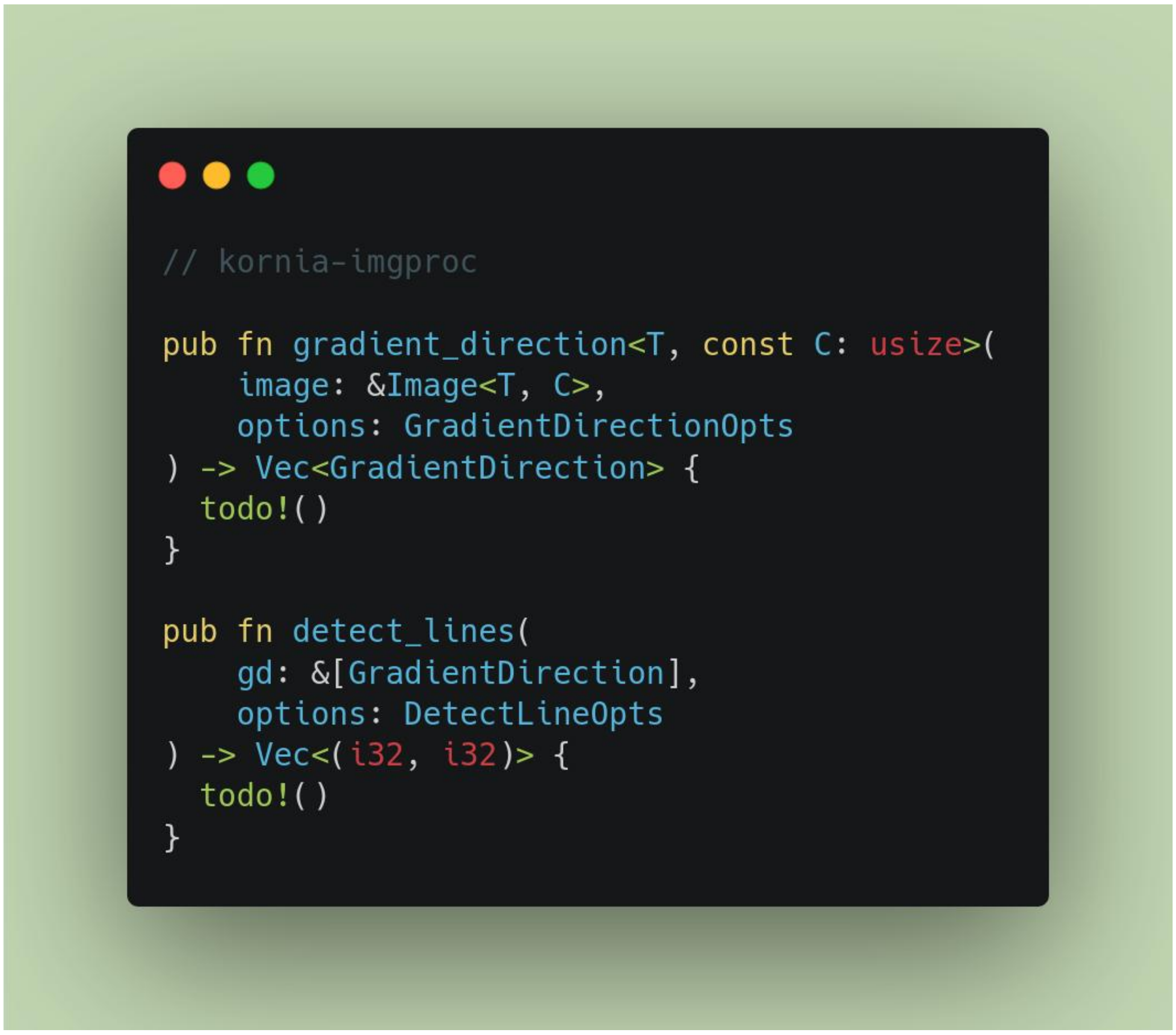
Instead of supporting hardcoded family tags, this crate will allow an option to even add your own custom specification of tag. Not only that, but this will also consist of some other options that can be used to configure multiple steps. The following is the proposed syntax of *TagFamily*:



```rust
pub struct TagFamily {
  // Inner
}

impl TagFamily {
  pub const Tag36H11: Self = /* Tag Info */;
  // Other tags
}
```

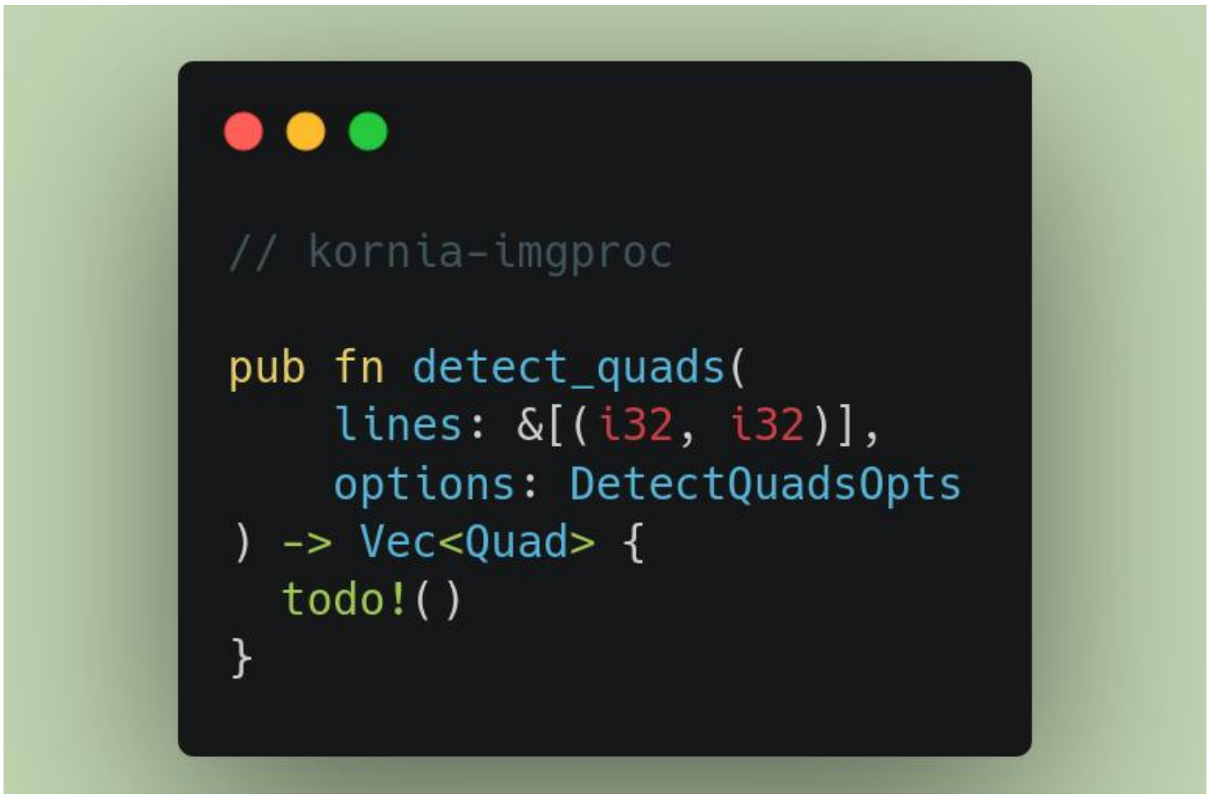
The first step is image pre-processing, which involves computing the Gradient Direction and Magnitude at each pixel.

And in the next step, detect the lines from *Vec<GradientDirection>*. Both functions will have their options argument to allow customization in values of error, etc. These functions will add in *kornia-imgproc* crate as they can be used outside of April Tag detection. The following is the proposed *gradient_direction* and *detect_line* function syntax:



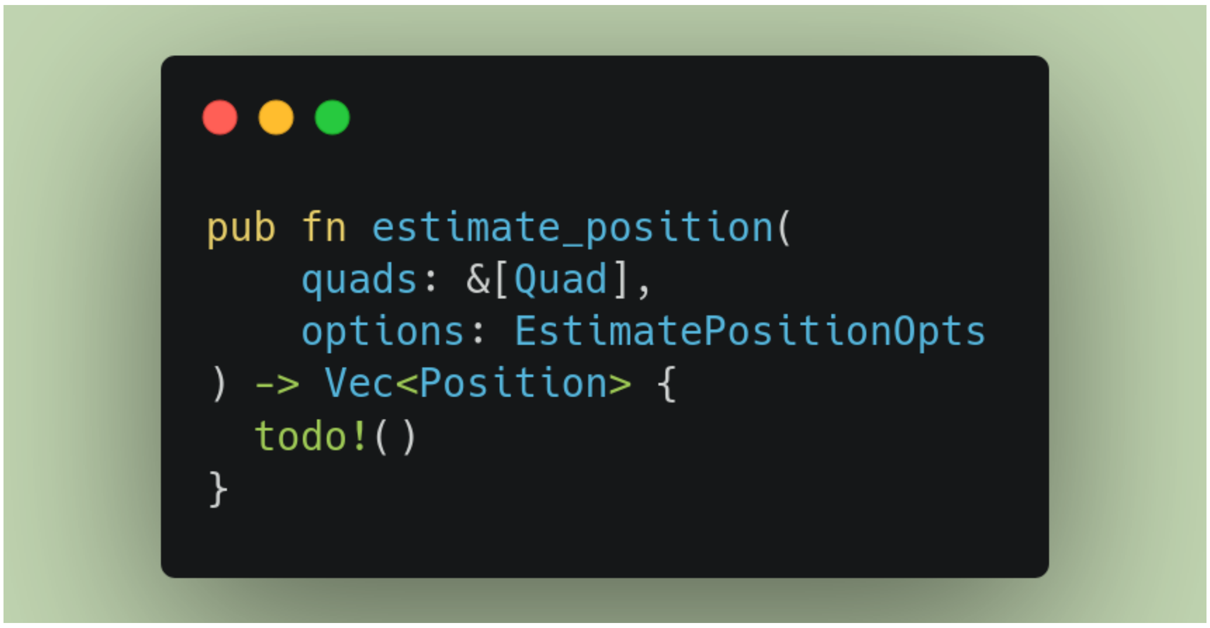
```rust
// kornia-imgproc

pub fn gradient_direction<T, const C: usize>(
    image: &Image<T, C>,
    options: GradientDirectionOpts
) -> Vec<GradientDirection> {
  todo!()
}

pub fn detect_lines(
    gd: &[GradientDirection],
    options: DetectLineOpts
) -> Vec<(i32, i32)> {
  todo!()
}
```

Next step will be to detect Quads *(four sided regions)*. This function will implement Recursive Depth first search algorithm with the depth of four. The following is the proposed *detect_quads* function:



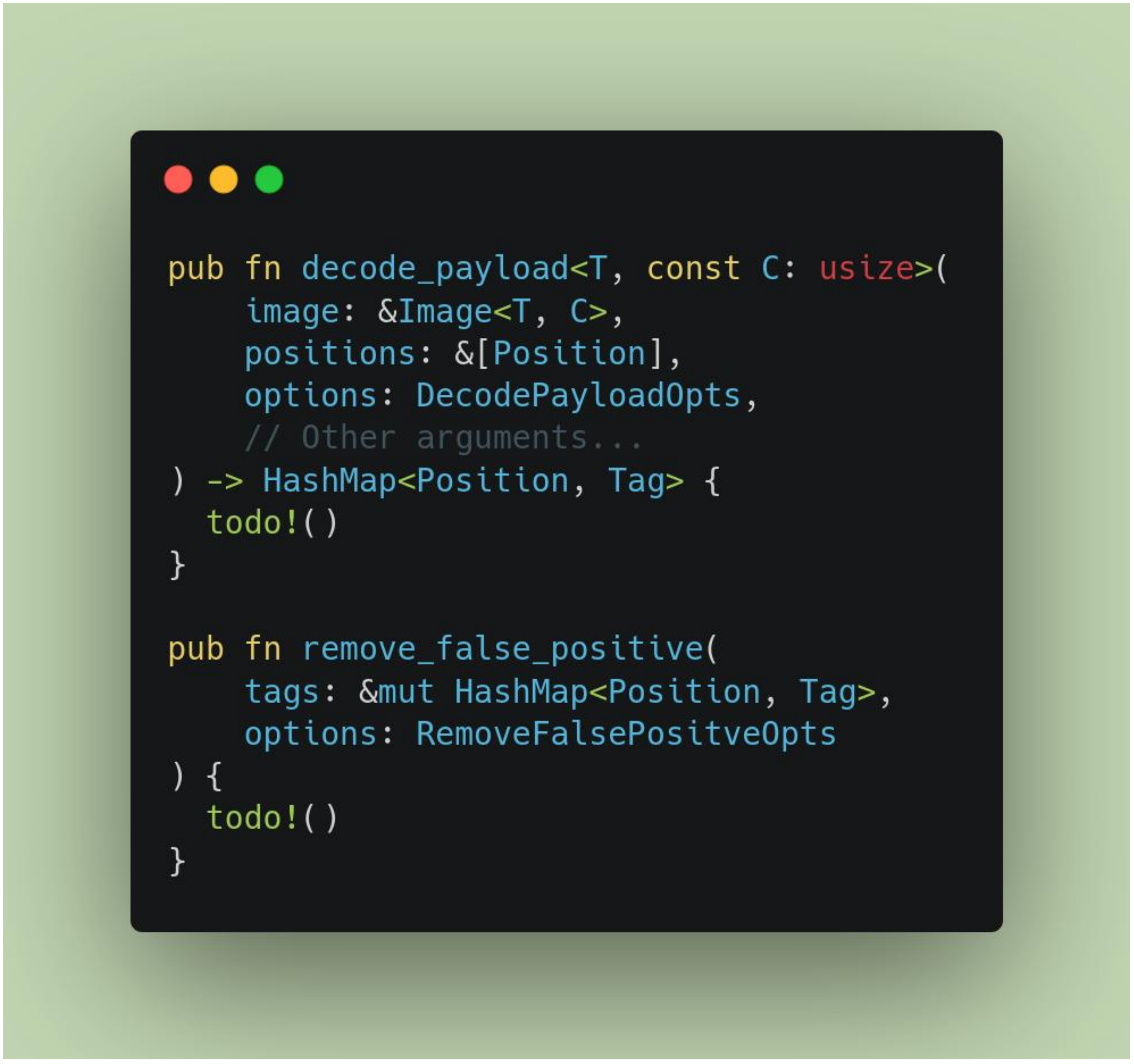
```rust
// kornia-imgproc

pub fn detect_quads(
    lines: &[(i32, i32)],
    options: DetectQuadsOpts
) -> Vec<Quad> {
  todo!()
}
```

Now, we need to estimate the 3D position of the quads (*Or in other words "Homography and extrinsics estimation". The April Tag Paper does a great job explaining it).* This function will be taking a slice of *Quad* along with an options argument that can be used to the inner workings of this function. The following is the proposed syntax of the *estimate_position* function:



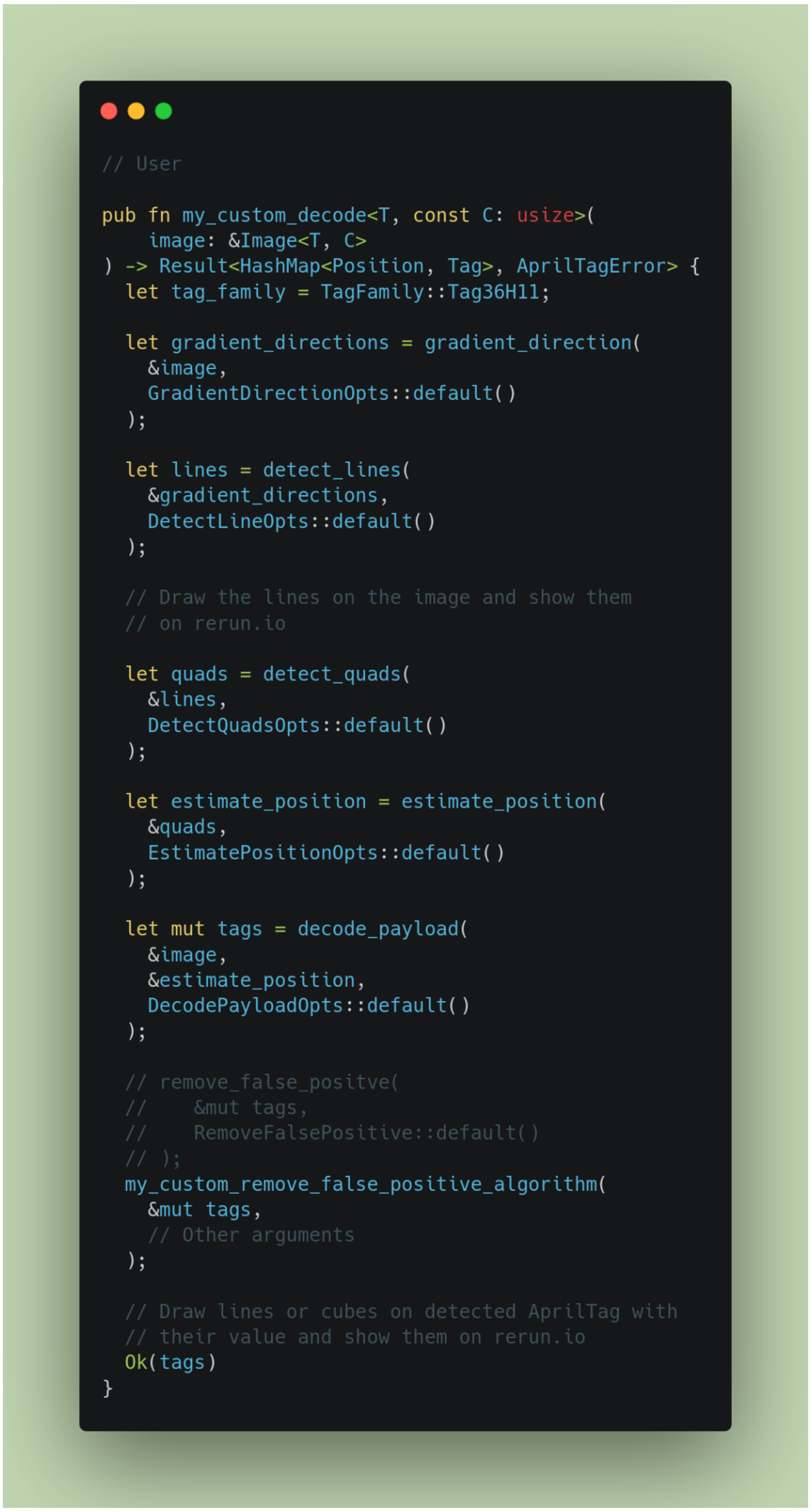
```rust
pub fn estimate_position(
    quads: &[Quad],
    options: EstimatePositionOpts
) -> Vec<Position> {
  todo!()
}
```

The next stage is going to decode the contents of payload. For this we will have a *decode_payload* function that takes a reference to *Image* and slice of *Position* with other arguments and returns *HashMap<Position, Tag>*. This function is going to return all our tags along with their position in 3D space, but there are going to be some false-positive due to camera been rotated to 90 deg, 180deg, 75deg, etc. To make the April Tag detection more accurate, we need to eliminate those false positives. For this we would be using the approach explained in *Coding Section* of April Tag Paper. The following is the proposed syntax of *decode_syntax* and *remove_false_positive* function:



```rust
pub fn decode_payload<T, const C: usize>(
    image: &Image<T, C>,
    positions: &[Position],
    options: DecodePayloadOpts,
    // Other arguments...
) -> HashMap<Position, Tag> {
  todo!()
}

pub fn remove_false_positive(
    tags: &mut HashMap<Position, Tag>,
    options: RemoveFalsePositveOpts
) {
  todo!()
}
```

Our April Tag detection system is now complete, structured for users to implement custom logic using existing functions. For example, a user may want to modify their webcam stream to draw lines of every detected line, create cubes based on tag values, and run them on rerun.io and they want to use their own version of the *remove_false_positive* function, they can do so easily due to the plug-and-play nature of our API. Here is an example function that the user might create:

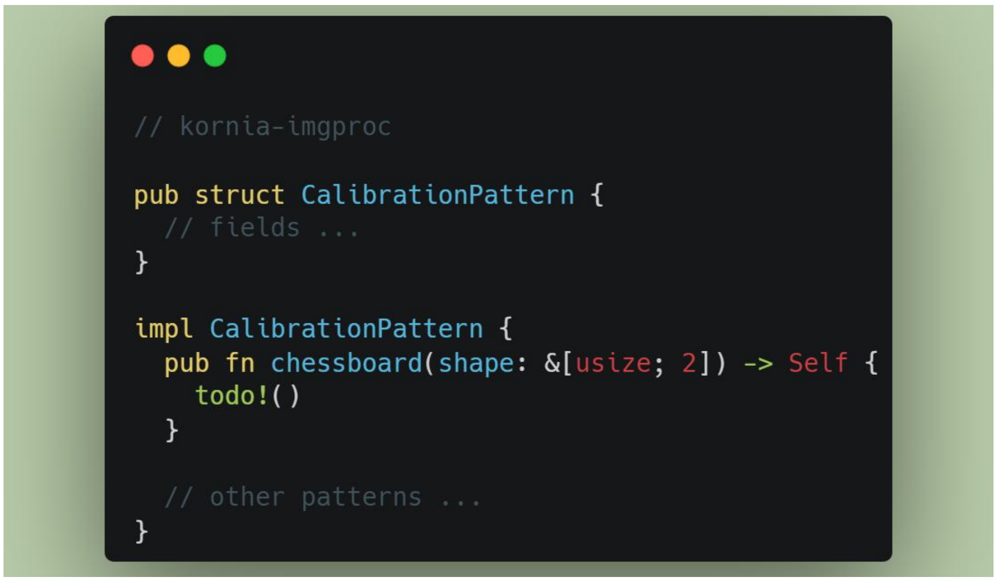


```rust
// User

pub fn my_custom_decode<T, const C: usize>(
    image: &Image<T, C>
) -> Result<HashMap<Position, Tag>, AprilTagError> {
  let tag_family = TagFamily::Tag36H11;

  let gradient_directions = gradient_direction(
    &image,
    GradientDirectionOpts::default()
  );

  let lines = detect_lines(
    &gradient_directions,
    DetectLineOpts::default()
  );

  // Draw the lines on the image and show them
  // on rerun.io

  let quads = detect_quads(
    &lines,
    DetectQuadsOpts::default()
  );

  let estimate_position = estimate_position(
    &quads,
    EstimatePositionOpts::default()
  );

  let mut tags = decode_payload(
    &image,
    &estimate_position,
    DecodePayloadOpts::default()
  );

  // remove_false_positve(
  //    &mut tags,
  //    RemoveFalsePositive::default()
  // );
  my_custom_remove_false_positive_algorithm(
    &mut tags,
    // Other arguments
  );

  // Draw lines or cubes on detected AprilTag with
  // their value and show them on rerun.io
  Ok(tags)
}
```

Technical Overview (Camera Calibration):

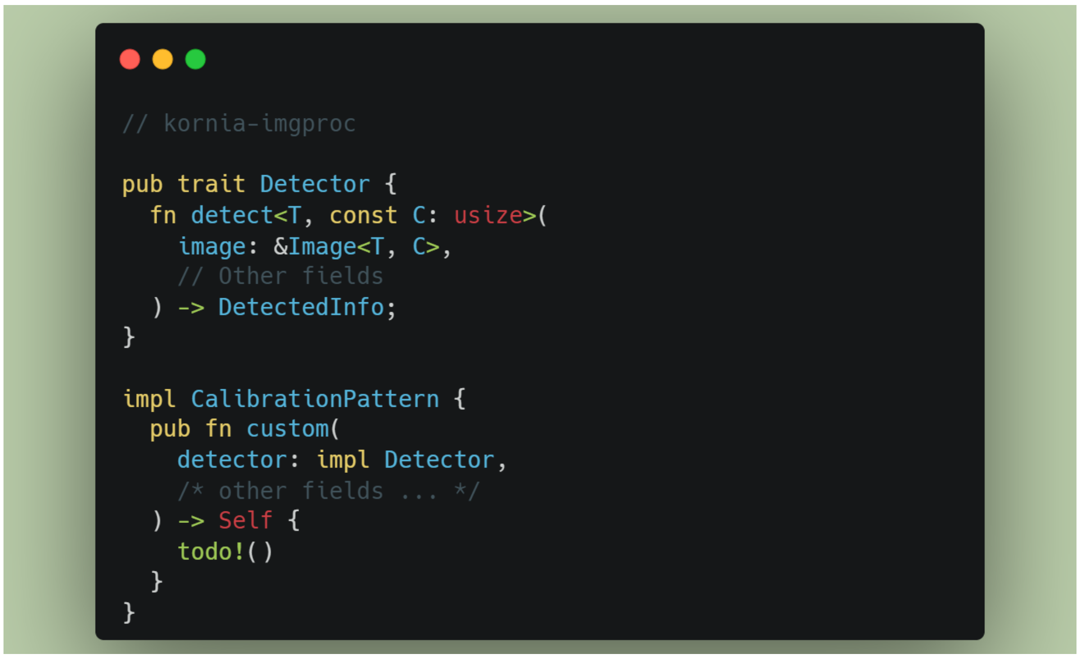
In this section, I will go through some API design decisions that I propose and why. I will also discuss them with the mentors to improve it further and fit a wider use case.

For calibrating the image, we need reference to real work information. For example, if we use the Chessboard method we would need the number of cubes. This maybe not to suitable for every use case, therefore we need our calibration system to provide some default implementations but also provide the ability to customize. The following is the proposed syntax of *CalibrationPattern*:



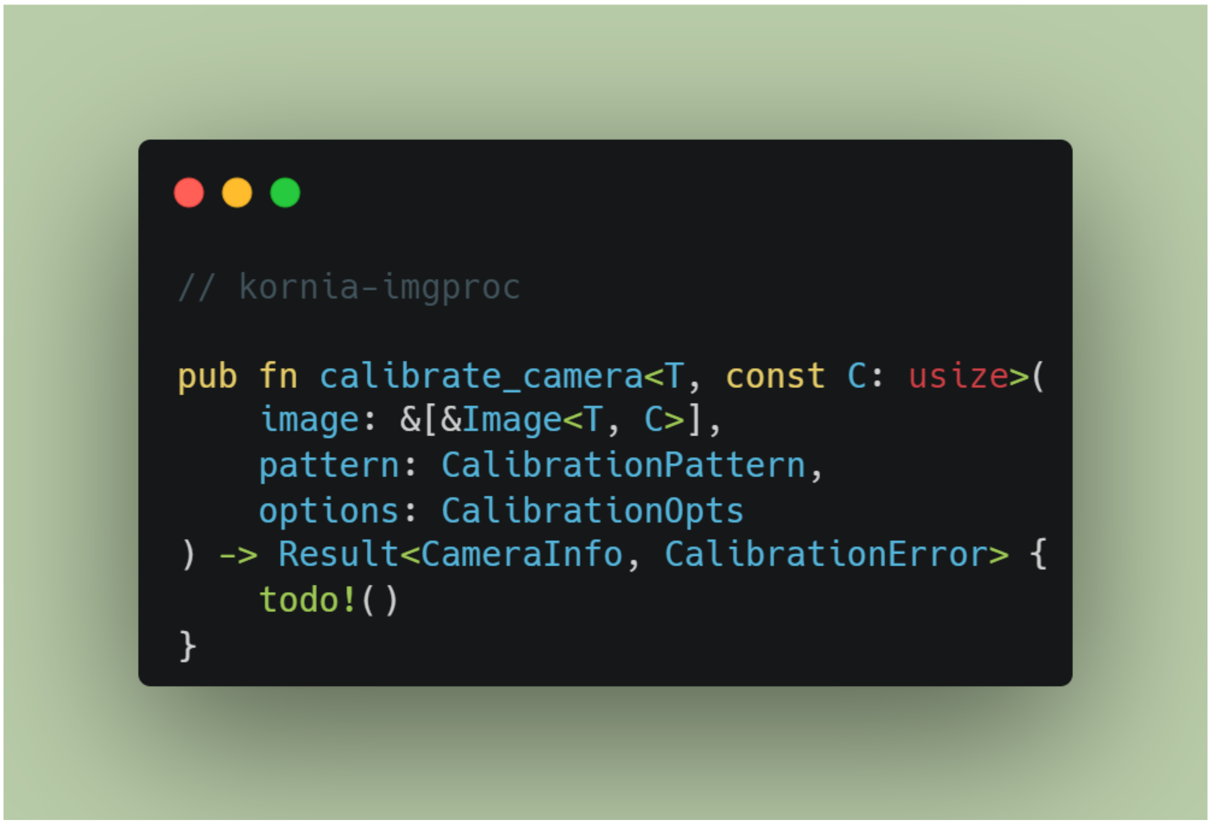
```rust
// kornia-imgproc

pub struct CalibrationPattern {
  // fields ...
}

impl CalibrationPattern {
  pub fn chessboard(shape: &[usize; 2]) -> Self {
    todo!()
  }

  // other patterns ...
}
```

This way the user is not only limited to chessboard for calibration but can use anything. For this we will have a *Detector* trait that will allow the user access to image and with other configurations and they will return *DetectedInfo* which will be used to calibrate the camera. The following is the proposed syntax of *Detector* trait and *custom* function:



```rust
// kornia-imgproc

pub trait Detector {
  fn detect<T, const C: usize>(
    image: &Image<T, C>,
    // Other fields
  ) -> DetectedInfo;
}

impl CalibrationPattern {
  pub fn custom(
    detector: impl Detector,
    /* other fields ... */
  ) -> Self {
    todo!()
  }
}
```

Now, for the main logic of camera calibration will be available through *calibrate_camera*. The following is the proposed syntax of *calibrate_camera* function:



```rust
// kornia-imgproc

pub fn calibrate_camera<T, const C: usize>(
    image: &[&Image<T, C>],
    pattern: CalibrationPattern,
    options: CalibrationOpts
) -> Result<CameraInfo, CalibrationError> {
    todo!()
}
```

This function will be doing the following behind the scenes:

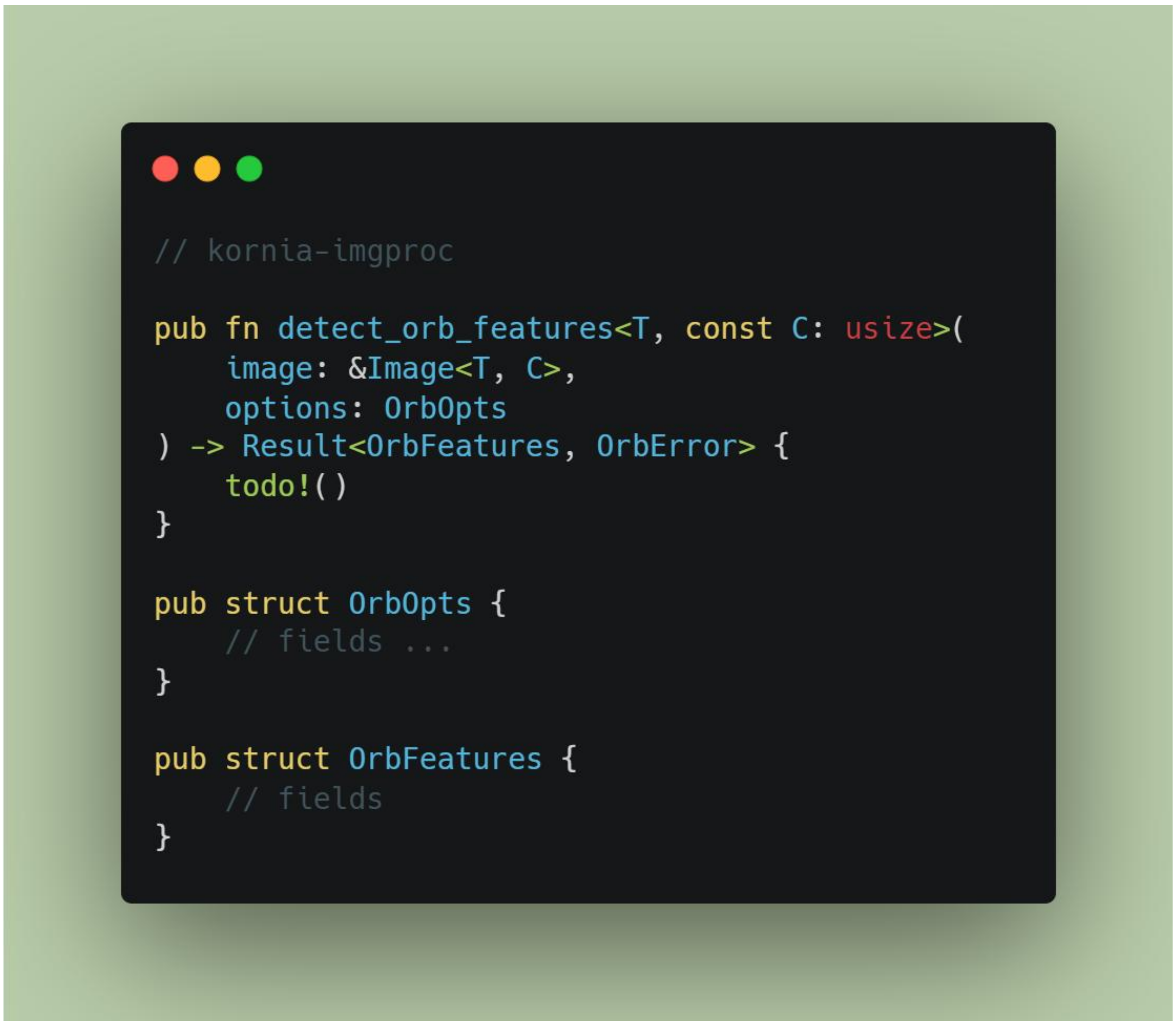
1. **Feature Detection:** This will be done using *CallibrationPattern* type. It will come with the detector logic for common patterns like chessboard. The *CallibrationPattern::custom* method would allow the user to use other detection algorithms like ORB.
2. **Estimation of Initial Parameters**: For this we will be using Camera Calibration by Zhang's Method.
   In the *options* arguments the user can also specify the camera information like focal length, if available. This will make the estimation more accurate.
   Also, the *options* field will accept a closure that if available can override the default estimation method, providing more customizability.
3. **Refinement using iterative optimization:** The calibration process will be applied to multiple images to find the optimal *CamerInfo*.

All the logic of "Estimation of Initial Parameters" and "Refinement using iterative optimization", will be available as public function so that they can be used in different use cases. The calibrate_camera will call these functions inside.

Technical Overview (Visual SLAM Implementation):

In this section, I will explain how I would implement common algorithms like ***ORB*** for visual localization. With these algorithms supported by kornia-rs directly, I will work with Hauke Strasdat *(author of sophus-rs)* to show an integration of a calibration system either by using April Tag Detector or Visual SLAM Implementation.

I will implement ***ORB*** algorithm and will use the existing fast detector implementation in *kornia-imgproc*. I will try to improve it's performance if possible. The following is the proposed syntax of *detect_orb_features*:



```rust
// kornia-imgproc

pub fn detect_orb_features<T, const C: usize>(
    image: &Image<T, C>,
    options: OrbOpts
) -> Result<OrbFeatures, OrbError> {
    todo!()
}

pub struct OrbOpts {
    // fields ...
}

pub struct OrbFeatures {
    // fields
}
```

The function will do the following behind the scenes:

1. Apply FAST to detect key points.
2. Compute the orientation of each key point.
3. Compute the rotated BRIEF descriptor for each key point.
4. Use an image pyramid to ensure scale invariance.

Timeline

Before June 2:

- Get to know more about April Tags Detection by reading papers like AprilTag: A robust and flexible visual fiducial system *by Edwin Olson*.
- Understanding how *aprilgrid-rs* works behind the scenes.
- Learning more about Nvidia VPI, OpenCV and OpenCV Rust Bindings.
- Understanding *sophus-rs*.

Week 1 (June 2 – 8):

- Finalizing How the crate API should look like and aligns with the kornia-rs project by discussing it with mentors.
  My university has set the tentative date for my End-Semester Exams to begin either from mid of May or first Week of June. There is a slight chance that My exam could be during the first week.

Proposed Pull Requests:

S. No.	PR Description	Proposed Creation Date	Expected Merge Date
1.	Add *kornia-apriltag* crate	June 5th, 2025	June 8th, 2025

Week 2 (June 9 – 15):

- Implement line detecting functionality by computing Gradient Direction and Magnitude at every pixel.
- Add Example with rerun.io to showcase detecting line segments functionality.
- Add benchmark against OpenCV Gradient functions.

Proposed Pull Requests:

S. No.	PR Description	Proposed Creation Date	Expected Merge Date
1.	Implement Line detecting functionality	June 13th, 2025	June 15th, 2025
2.	Add benchmark against OpenCV	June 15th, 2025	June 17th, 2025

Week 3 (June 16 – 22):

- Implement Quad *(four sided regions)* Detection.
  My approach is to use Recursive Depth-First search with the depth of four.
- Extend the example to also showcase Quad Detection.
- Add Benchmark to verify the robustness of quad detection and would help us find the optimal configuration that is both fast and accurate.

Proposed Pull Requests:

S. No.	PR Description	Proposed Creation Date	Expected Merge Date

1.	Implement Quad Detection functionality	June 20th, 2025	June 22nd, 2025
2.	Add benchmark for Quad Detection	June 22nd, 2025	June 24th, 2025

Week 4 (June 23 – 29):

- Implement Position Estimation of detected April Tag in 3D Space.
  Or in other words "Homography and extrinsics estimation". The April Tag Paper does a great job explaining it.
- Add Tests to verify its accuracy.
- Extend the example to also showcase Position Estimation.

Proposed Pull Requests:

S. No.	PR Description	Proposed Creation Date	Expected Merge Date
1.	Add Position Estimation in 3D Space of detected April Tag	June 29th, 2025	July 1st, 2025

Week 5 (June 30 – July 6):

- Implement April Tag Payload Decoding.
- Add Integration test to verify everything is correctly working and correct payload content is decoded.
  At this stage, there are high false-positive therefore the test verifying the payload content will be limited.
- Extend the Example to showcase Payload Decoding.

Proposed Pull Requests:

S. No.	PR Description	Proposed Creation Date	Expected Merge Date
1.	Add April Tag Payload decoding	July 6th, 2025	July 8th, 2025

Week 6 (July 7 – 13):

- Add system to remove false positive in April Tag detection.
  In Coding System section of April Tag Paper, the use of modified lexicode along with other strategy, a wide range of false-positive can be removed.
- Add more Tests with wider range of April Tags and verify our Accuracy.

Proposed Pull Requests:

S. No.	PR Description	Proposed Creation Date	Expected Merge Date
1.	Add system to remove false positive in April Tag detection	July 12th, 2025	July 14th, 2025
2.	Add more robust test-suite	July 13th, 2025	July 15th, 2025

(July 14 – 18): Midterm Evaluation

Week 7 (July 14 – 20):

- Review the existing code and fix any bugs or code duplication there might be.
- Complete Documentation
- Add benchmark against OpenCV, Nvidia VPI and Aprilgrid-rs.
- Integrate *kornia-py* with *kornia-apriltag*.

Proposed Pull Requests:

S. No.	PR Description	Proposed Creation Date	Expected Merge Date
1.	Add benchmark against OpenCV, Nvidia VPI and Aprilgrid-rs	July 17th, 2025	July 19th, 2025
2.	Make *kornia-apriltag* available on python using *kornia-py*	July 20th, 2025	July 22nd, 2025

Week 8 (July 21 – 27):

- Implement Camera Calibration
- Add benchmark of Camera Calibration against OpenCV, and Nvidia VPI.

S. No.	PR Description	Proposed Creation Date	Expected Merge Date
1.	Implement Camera Calibration	July 27th, 2025	July 29th, 2025

Week 9 (July 28 – August 3):

- Work together with *Hauke Strasdat (author of sophus-rs)* to show an integration of a calibration system using the April Tag detector and/or using FAST features for a Visual SLAM implementation.
  This task is flexible and can be interchanged with tasks of other week, depending on the availability of Hauke Strasdat.

Week 10 (August 4 – 10):

- Continue Working with *Hauke Strasdat (author of sophus-rs)* to show an integration of a calibration system using the April Tag detector and/or using FAST features for a Visual SLAM implementation.
- Add support for GPU by coordinating with the Contributor of "Implement GPU image transforms" idea and the mentor.

Proposed Pull Requests:

S. No.	PR Description	Proposed Creation Date	Expected Merge Date
1.	Add support for GPU in *kornia-apriltag*	August 10th, 2025	August 12th, 2025

Week 11 (August 11 - 17):

- Add a simulation-based example of *kornia-apriltag* using kornia, copper and bevy. This example will be finalized by communicating with the mentor.

Proposed Pull Requests:

S. No.	PR Description	Proposed Creation Date	Expected Merge Date
1.	Add simulation-based example for *kornia-apriltag* and camera calibration	August 17th, 2025	August 19th, 2025

A buffer of one week has been kept for any unpredictable delay.

Open-Source Contributions

It's not my first time to contributions in Open Source, but in the past few weeks, I have dived into the codebase and made some contributions on kornia-rs. Although this is my first time contributing to kornia-rs, I really enjoyed working on the codebase and by now, I have got familiar with how it works and excited to contribute more.

Kornia-rs:

- PR #327: Add JPEG image encoding and decoding functions
- PR #328: Fix kornia-io docs not building on docs.rs
- PR #330: Fix CI
- PR #332: Extend PNG encoding and decoding support
- PR #336: Decode Images directly from raw bytes
- PR #338: Implement Zero copy images from gstreamer

Copper-rs:

- PR #199: Make config entry to disable the logging in the ron file
- PR #206: Improve the logging parameters user friendliness
- PR #212: Catalog and catch the main proc macro usage error
- PR #215: Fix No space left on device (os error 28) and improve CI speed
- PR #219: Add debug output pane to cu-consolemon
- PR #228: Make cu-consolemon debug pane only active if text_log is true
- PR #263: Add iyes_perf_ui back

Azalea-rs:

- PR #156: Added Left Click Mine Functionality
- PR #158: fix: Chunk Storage Error
- PR #159: Auth Customization Options
- PR #167: Made Hunger and DataComponentPatch public
- PR #168: Added Left Click Mine (1.21)

- PR #188: Feat: Host Documentation for multiple versions/branch

Other:

- rust-lang/rust: PR #134880: Made Path::name only have item name rather than full name
- wikisource/wscontest: PR #76: Add UTC in the contest page
- wikisource/wscontest: PR #77: Make all interface messages translatable
- gerrit.wikimedia.org: MR #1094167: Show user a human readable message when $wgLocaltimezone is set to an invalid timezone

and other various open-source contributions…

Availability

My university has set the tentative date for my End Semester exams to begin either from Mid of May or in the first week of June. I will still be working during those weeks, but my hours spent per week would decrease slightly to 20-25 Hours per week.

After my exams, I will be having my summer break for around 2.5 months, and I don't have any other commitments during this period. So, I can easily devote around 30-35 Hours per week.

I have also kept a buffer of one week for any unpredictable delay and would even devote extra hours to complete the project in time.

Post GSOC

I would have learned a lot by contributing to kornia-rs and wish to continue the same even after GSOC is over. In case some of the PRs need more work, I would be happy to work on them after GSOC and will remain active on discord. I would be happy to be the part of kornia community.