Component Design
24783: Advanced Engineering Computation

Project Name: Monocular Visual SLAM System
Team Name: Will_Code_for_Food
Team Members: Aditya Ramakishran, Neel Pawar, Samiran Gode

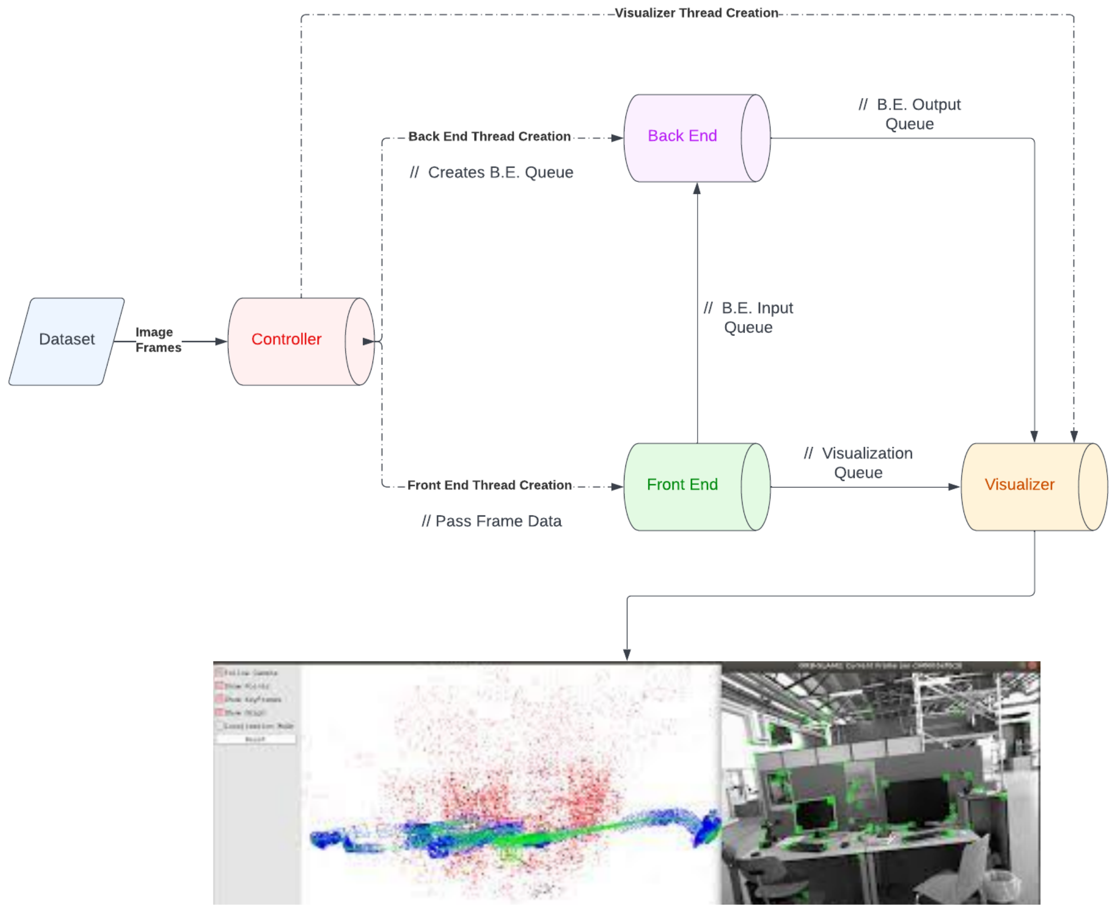
I. Introduction

SLAM is the computational problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent's location within it. Our objective within the scope of this project is to implement a small-scale monocular SLAM system for static environments, using C++.

Our system will detect landmark keypoints and track these across the input image keyframes giving us visual factors (Front-end). We then use these factors to create a pose-graph and optimize it incrementally to give us final camera poses and final landmark positions in a map (Back-end). The objective is to produce the map and the input frames with key points in parallel for the user to view (Viewer)

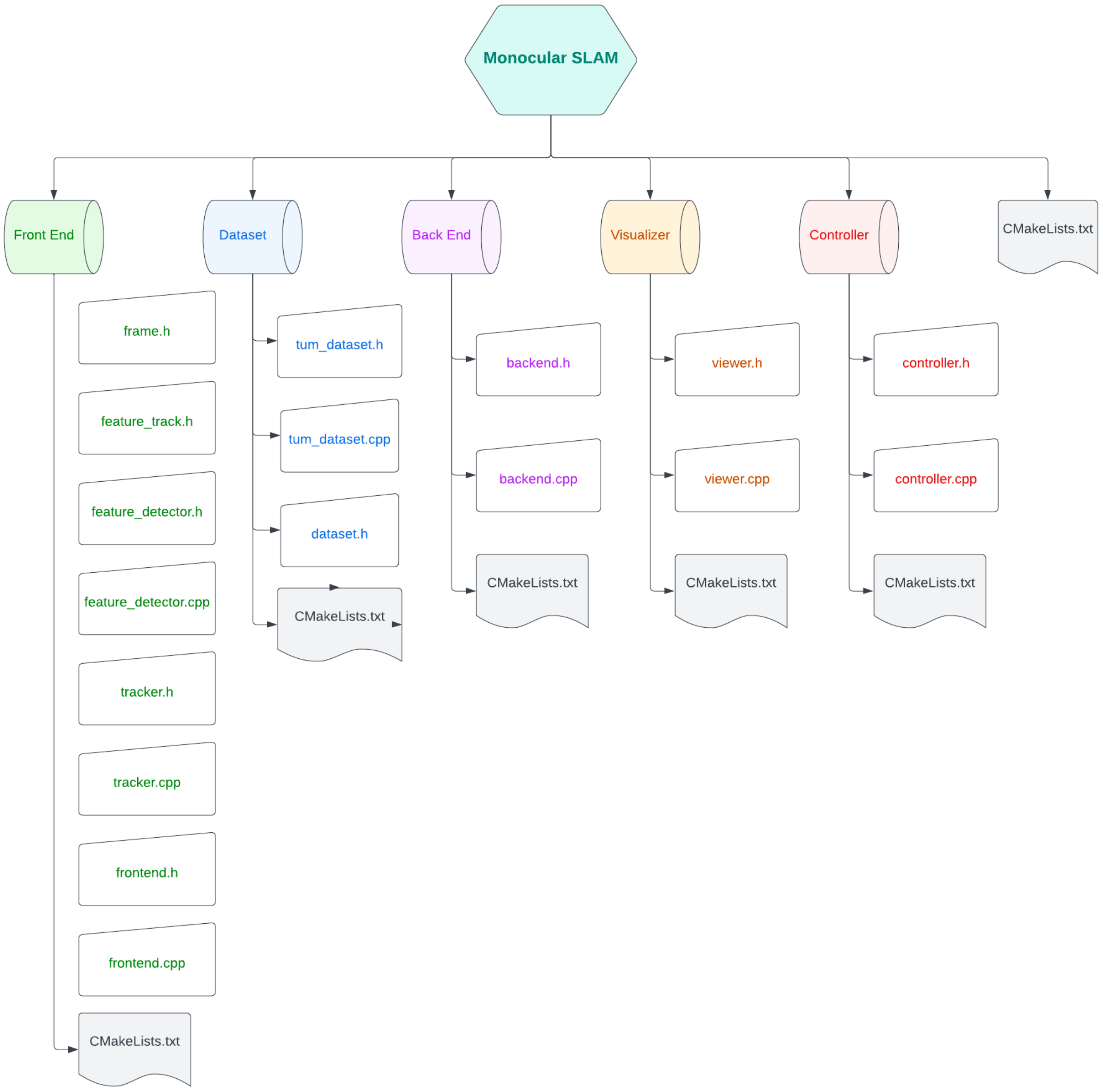
II. Project Pipeline:

Our project pipeline can be represented as follows:



III. Directory Structure:

Our directory structure can be represented as follows:



IV. Component: Dataset

Dataset component will be responsible for reading the input image data. An open source repository of RGB and depth images from the Technical University of Munich (TUM) is used as a dataset source for our project and we extract these image-frames from the TUM repository through the "Dataset" component.

It will have the following subcomponents and functions:

1. Struct Dataset_Item
    a. Max_objects_in_an_image
    b. Timestamp
    c. cv::Mat color_image
2. Size_t length()

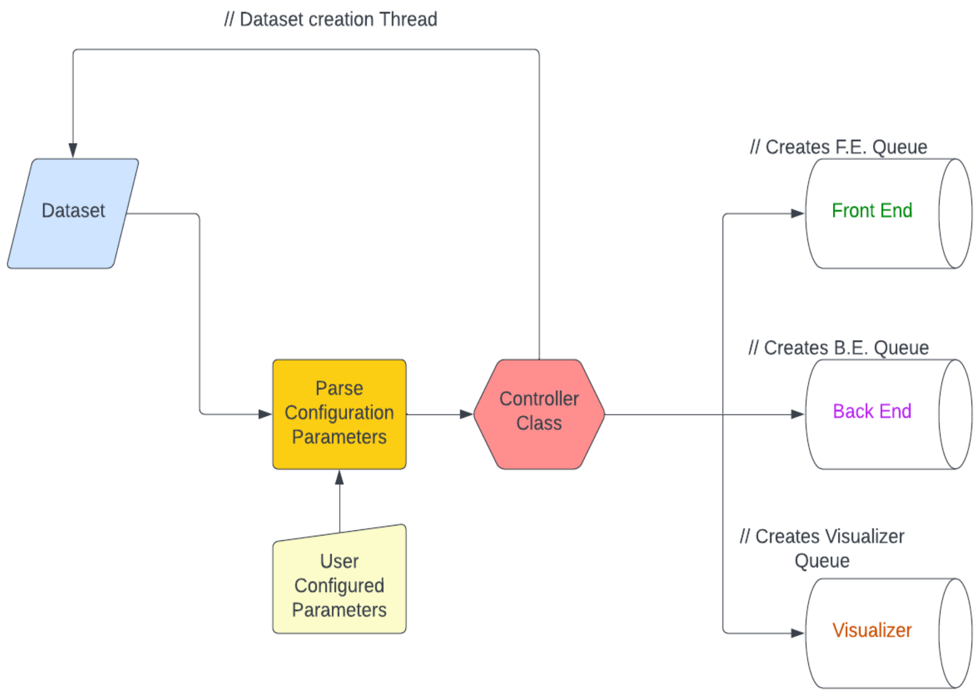
3. DatasetItem getItem(index)
4. <size_t, size_t> getImageSize()
5. vector<pair<Timestamp, gtsam::Pose3>> getGroundTruthPoses()
6. Eigen::Matrix3d getCameraIntrinsics()
7. bool parseDataset() // Will parse the dataset to assert if all the information like camera intrinsics etc. is available, if available, it will read it.
8. vector<std::string> readFile(const fs::path &txt_file);
9. void readIntrinsics(fs::path &intrinsics_file);
10. pair<std::vector<fs::path>, vector<Timestamp>> readImageFile(path &txt_file)
11. vector<pair<utils::Timestamp, gtsam::Pose3>> readGroundTruthFile(const path &txt_file);
    It will read the input information like camera intrinsics, read the file using opencv, and read the ground truth information that we get from the dataset.

Test Strategy:

a. Data missing: We'll write a unit test to check if the data in the file is present and has the correct number of images.
b. Data incorrect: We'll check if the format of the images is correct so that we have the expected output and don't end up with runtime errors
c. Data not read correctly: Since this is an external source, it might have a format which we would be able to read but not might be in the format we write our code in so we might have to check if what we are reading makes sense.

V. Component: Controller

The "Controller" component is the primary interface between the other components. It is responsible for initializing the individual component threads, reading dataset files, parsing configuration parameters and processing and shutdown control


// Dataset creation Thread
Dataset
// Creates F.E. Queue
Front End
// Creates B.E. Queue
Back End
Parse Configuration Parameters
Controller Class
// Creates Visualizer Queue
Visualizer
User Configured Parameters

It will have the following subcomponents and functions:

1. Setup() - *Initializes each queue and thread required in the system*
2. Start() - *will call the "Run()" function once the setup is completed*
3. Run() - *Runs the initialized threads*

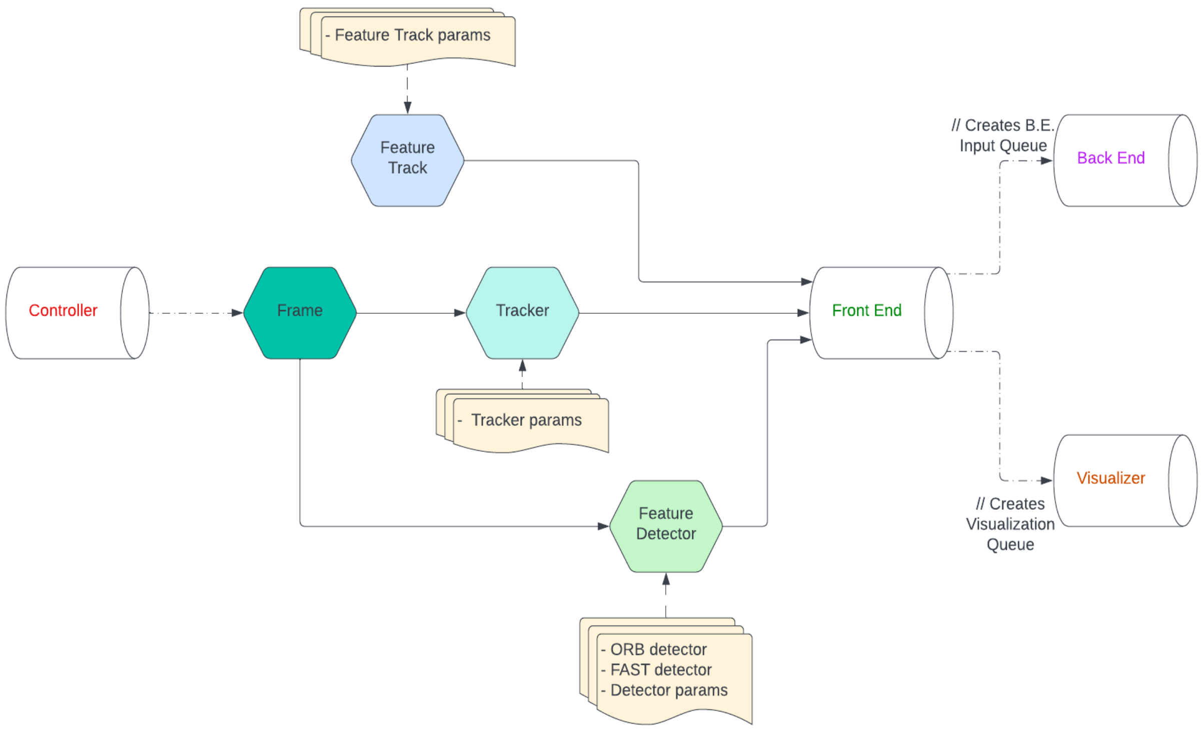
4. Shutdown() - *Joins the threads and shuts down the program*
5. ProcessData() - *Process the input ground truth data and controls the data represented in the viewer*

Test Strategy:

a. Ensuring shutdown operation is called correctly without data loss
b. Checking if the Image frame parameters from dataset are correctly read
c. Checking if the Ground Truth pose and the Sampled pose are correctly extracted

VI. Component: Frontend

The role of the "Frontend" components functionality is to process the already read information using the *Dataset* component and then pass this information to the *Backend* as well as the *Visualizer*. The *Controller* component will initialize the frontend queue. Frontend will have multiple subcomponents as described below;



- ### Subcomponent: Frame

  This subcomponent will include a class called "frame" which will encapsulate all the required information that each frame of the dataset will contain. Controller uses Dataset to create the frontend input queue which the other subcomponents of frontend will use. It has the following methods:

  1. Index
  2. Timestamp
  3. cv::Mat image
  4. Feature status - *status of feature (tracked, invalid or terminated)*
  5. Features
  6. vector<size_t> feature_ages
  7. gtsam::Pose3 frame_T_g;

Testing Strategy

a. Check index and timestamp is correct: It is imperative that the index and time stamp of the image are correct since there is a queue in place which is followed by other components as well.
b. Checking the feature statuses.
c. Checking that the poses are in the correct orientation.

- ### Subcomponent: Feature Detector

  Feature detector detects the features in each frome using the Orb feature detectors or the FAST detectors depending on which the user chooses. This information is then used by the other subcomponents such as feature_tracker to calculate the factors which are then passed to the backend. It implements generic feature detectors supporting different features. And has the following methods.

  1. struct ORBDetectorParams
  2. struct FASTDetectorParams
  3. struct DetectorParams
  4. Void detect(Frame & frame, const cv::Mat & mask) // This will detect the features and store them in a vector.

Testing strategy

a. Checking the keypoint and track_ID are consistent
b. Checking if the detections actually make sense, that is for certain cases there should be no answer and in others it should have only one answer.
c. Making sure the parameters are within reason

- ### Subcomponent: Feature Track

  Feature track does exactly what it says, it quite literally tracks the features across image frames using the KLT tracker implementation from OpenCV. It does so in order to triangulate the points in the image across frames. It has the following member functions and member variables.

  1. void emplace_back(utils::Index _frame_id, const gtsam::Vector2 &_pix_observation)
  2. size_t length()
  3. TrackId track_id
  4. FeatureMeasurements features

- ### Subcomponent: Tracker

  This is an implementation for tracking keypoints across keyframes. And it implements the following members:

  1. gtsam::Pose3 track(Frame &prev_frame, Frame &curr_frame, <gtsam::Rot3> camera2_R_camera1)
  2. cv::Mat visualizeTracks(const Frame &prev_frame, const Frame &curr_frame)
  3. vector<Keypoint> predictOpticalFlow(vector<Keypoint> &prev_keypoints, gtsam::Rot3 &camera2_R_camera1);
  4. cv::Matx33d rejectOutliers()
  5. Eigen::Matrix3d K_;
  6. Eigen::Matrix3d K_inverse_;
  7. cv::Matx33d K_cv_;
  8. cv::Matx33d K_inverse_cv_;
  9. TrackerParams params_;

- **Subcomponent: Frontend**

  Frontend factory to load different types of detector + tracker combinations. It is a class to combine all the above in a fashion that would make sense. It contains the following methods and member variables

  1. Input_queue
  2. output_vis_queue
  3. backend_queue
  4. FrontendParams params_
  5. unique_ptr<std::thread> process_thread_
  6. void run()
  7. void processFrame(shared_ptr<Frame> & frame)
  8. pair<size_t, size_t> searchFeaturesInTracks(Frame & frame)
  9. bool checkFrameQuality(const Frame & frame)
  10. void addTrackedFeatures(const Frame & frame)
  11. std::unique_ptr<FeatureDetector> feature_detector_
  12. std::unique_ptr<FeatureTracker> feature_tracker_
  13. std::shared_ptr<Frame> prev_frame
  14. std::unordered_set<utils::Timestamp> keyframe_timestamps_
  15. FeatureTracks feature_tracks_

  ### Test Strategy

  a. Checking the correctness before combining
  b. Checking the correctness after combining since the final form should match
  c. Frontend can break easily since it combines a lot of things so checking those individual parts is crucial and those will individually be greater than 3 unit tests.

Component: Backend

Backend receives images, parameters, from the frontend and controller pipelines, and processes it for the visualization stage. Primarily it will create a factor graph with smart factors, (which will be done with the help of the GTSAM library). It will further update these factors from the incoming measurements, and create 3D points from the resultant smart factors.

- **Subcomponent: Initialize**

  Mainly, this subcomponent will perform following functionality:

  1. Initialize factor graphs.
  2. Initialize backend state.
  3. Initialize poses and keyframes.
  4. Initialize output queue.

  Function signature followed here will be:

  a. std::shared_ptr
  b. gtsam::NonlinearFactorGraph
  c. gtsam::ISAM2Result

- **Subcomponent: ProcessInput**

  Mainly, this subcomponent will perform following functionality:

  1. Receive features
  2. Generate a point map and trajectory
  3. Push generated data to the output queue.

Function signature followed here will be:

a. std::shared_ptr
b. utils::BackendState

- **Subcomponent: Update**

Mainly, this subcomponent will perform following functionality:

1. Receive features
2. Generate a point map and trajectory
3. Push to an output queue.

Function signature followed here will be:

a. std::shared_ptr
b. gtsam::NonlinearFactorGraph
c. gtsam::FactorIndices
d. boost::make_shared

- **Subcomponent: getPointMap**

Mainly, this subcomponent will perform following functionality:

1. Iterate over landmark ids in the active set and update 3D points from the graph.
2. Check if the corresponding factor is in the factor graph.
3. Match and add 3D points triangulated from frames.

Function signature followed here will be:

a. std::shared_ptr
b. gtsam::TriangulationResult
c. PointMap point_map

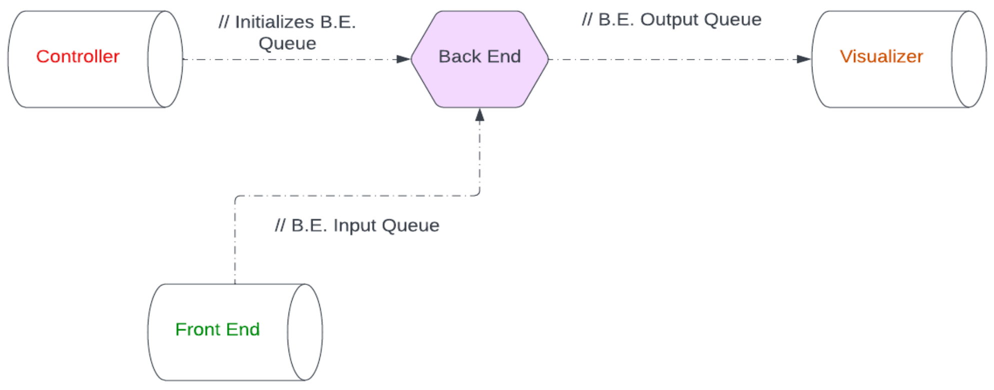
- **Subcomponent: getTrajectory**

Mainly, this subcomponent will perform following functionality:

1. Compute trajectory.
2. Get pose from GTSAM
3. Push to trajectory.

Function signature followed here will be:

a. std::shared_ptr
b. gtsam::Values::Filtered
c. gtsam::Pose3&
d. utils::Index



Test Strategy:

a. Checking if the GTSAM factors are accurate.
b. Checking if the input from the frontend to backend is correct.
c. Checking the backend update and initialisation.

Component: Viewer

Viewer component will be responsible for showing the trajectory and mapped landmarks. Viewer is also responsible for showing the image frames with the detected and tracked features over time. Following are the member variables and functions.

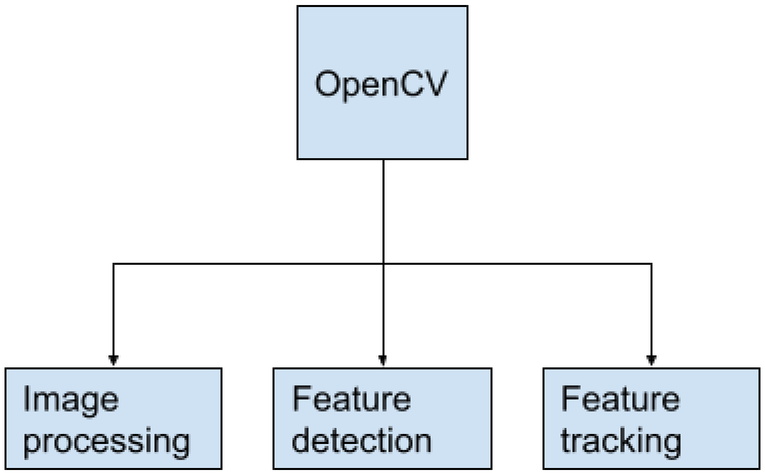
1. pangolin::View ui_view_;
2. pangolin::View main_view_;
3. pangolin::View image_view_;
4. pangolin::View map_view_;
5. pangolin::GlTexture rgb_texture_;
6. cv::Mat prev_vis_image_;
7. gtsam::Pose3 prev_camera_;
8. void handleImageView();
9. void handleMapView();
10. void drawCamera()
11. void drawTrajectory()

Test Strategy:

a. Viewer is pangolin based which is based on top of OpenGL so tests like checking if the final output windows have the correct views or not etc.
b. Checking if the backend queue input makes sense.
c. Checking if the frontend queue is correct and makes sense.

Component: OpenCV

OpenCV will primarily be used for image processing and vision algorithms on the image data as shown below.

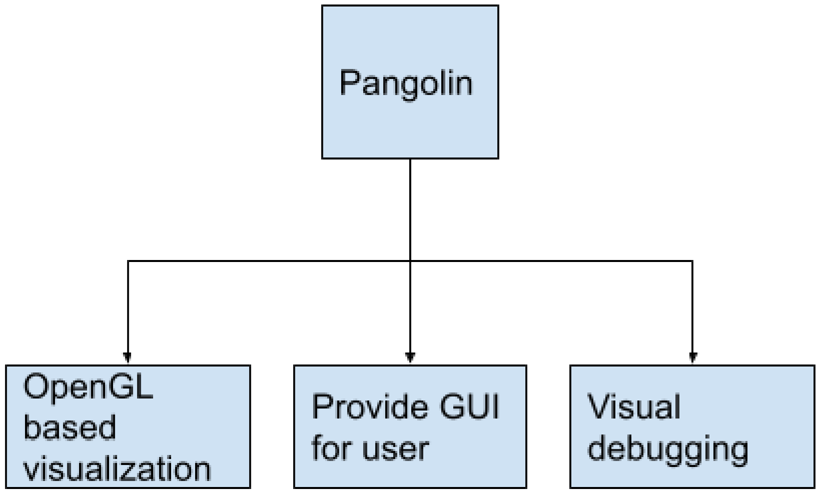


We decided to use OpenCV based on requirements in the following functions:

1. Reading in images in a space efficient array-format
2. Feature and keypoint detection
3. Tracking detected features across multiple frames

Component: Pangolin

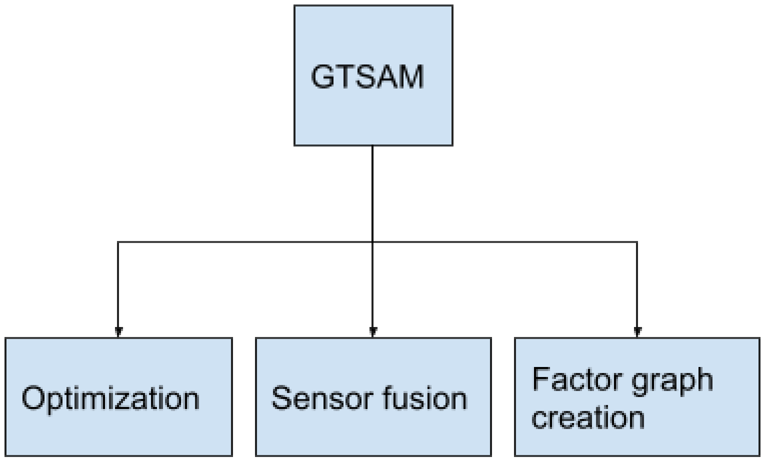
Pangolin is primarily an OpenGL-based library which is used for visualization of computed image results.



1. We will use Pangolin to show windows where we will display the different tracked features, camera projections and keypoints.
2. It will help visually debug the results and also provide a GUI for the end-user.

Component GTSAM

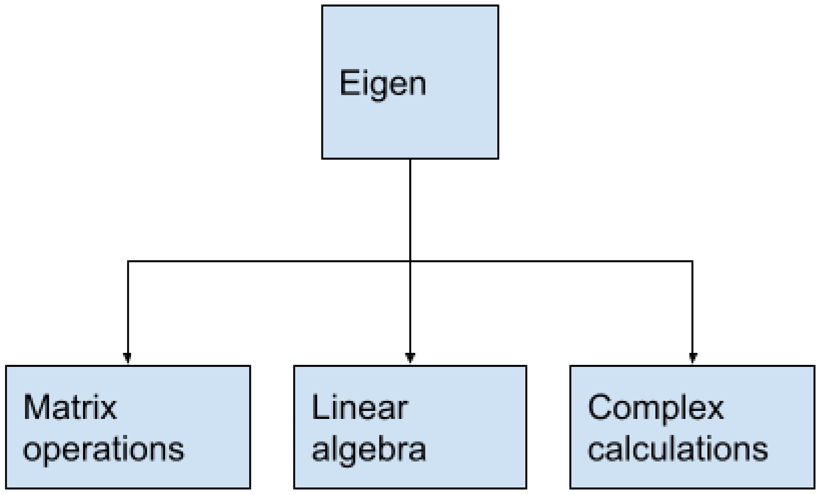
GTSAM is an open source library used for SLAM and Structure-for-Motion.



1. Generates a Nonlinear Factor graph in the backend component
2. Updates the graph as new factors are added every frame.
3. We use ISAM2 to get the final trajectory and landmark locations

Component: Eigen

Eigen is a mathematical library used for performing linear algebra useful for operating on complex matrix data.

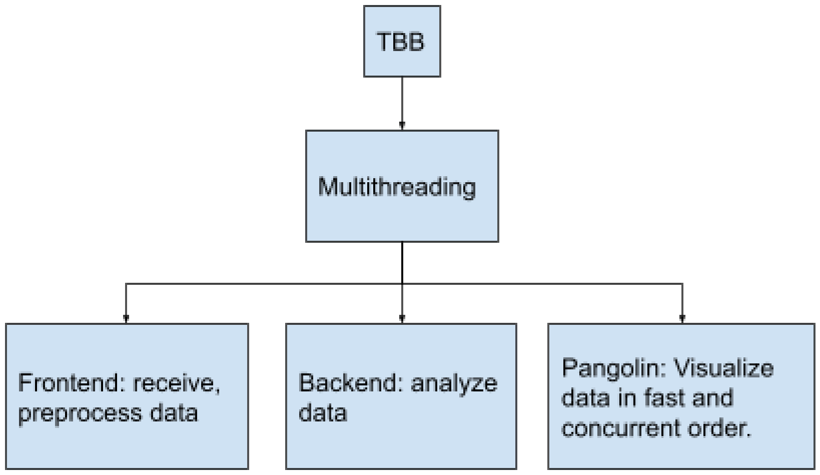


1. We will use Eigen to perform linear algebra on input images, camera parameters, resultant detections and matrices.

It will help with complex and repetitive operations, for example projections, singular value decompositions.

Component: TBB

Threading Building Blocks (TBB) is a library used for high-level parallel programming and multithreading.


TBB
Multithreading
Frontend: receive, preprocess data
Backend: analyze data
Pangolin: Visualize data in fast and concurrent order.

1. Frontend: Ordered queueing of collected image data in FIFO, with multithreading to improve speed complexity.
2. Backend: Create a queue to process received input, output image sequences in FIFO.
3. Pangolin: Visualizing frames, ie. multithreading of GUI in same order as received and processed

Component Responsibles:

1. **Dataset:** Samiran
2. **Frontend:** Neel
3. **Backend:** Aditya
4. **Viewer:** Neel
5. **Controller:** Aditya
6. **Overall Integration:** Samiran