# Modern C++ for Robust Bundle Adjustment

## From Outliers to Optimization

VISHNU SUDHEER MENON

Cppcon
The C++ Conference

2025
September 13 - 19

# Outline of the talk

- What is Visual Reconstruction?
- Bundle adjustment in the large
- SFM pipeline
- Data Parser
- Outlier Rejection
- Optimization
- Results

# Problem Statement

How do we design a visual reconstruction library that is adaptable, robust, scalable across datasets and optimization challenges?

- Dataset Diversity
  Works across different dataset formats and sensor types
- Flexible Filtering
  Plug-and-play support for methods like RANSAC, outlier rejection
- Optimizable Core
  Bundle Adjustment that converges reliably
  Efficient for both small and large datasets
- Scalability Requirement
  Avoids runtime explosion as dataset size grows

# What is Visual Reconstruction?

Recovering 3D structure of a scene from 2D images

- Infer geometry and structure of the world.
- Using primarily camera inputs.

Inputs/Outputs

- Inputs: single or multiple images, video sequences
- Outputs: 3D reconstruction (point clouds, meshes, dense models)

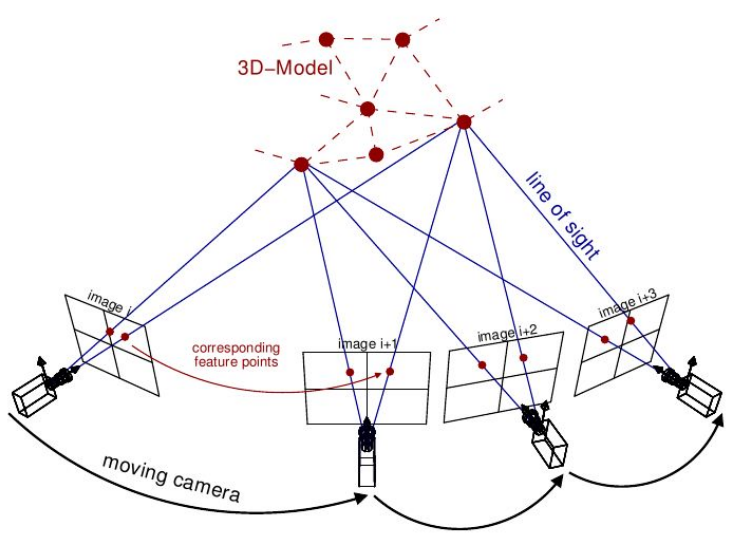
Applications

- Robotics & autonomous navigation
- Digital twins & mapping

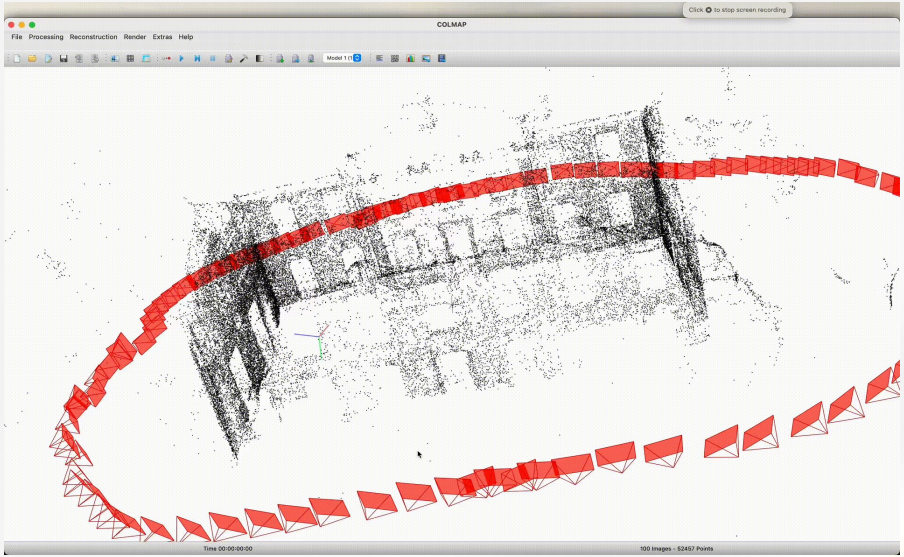
Challenges

- Occlusions & missing data
- Noise and outliers in features & correspondences
- Large-scale optimization can be computationally expensive

# What is Visual Reconstruction?

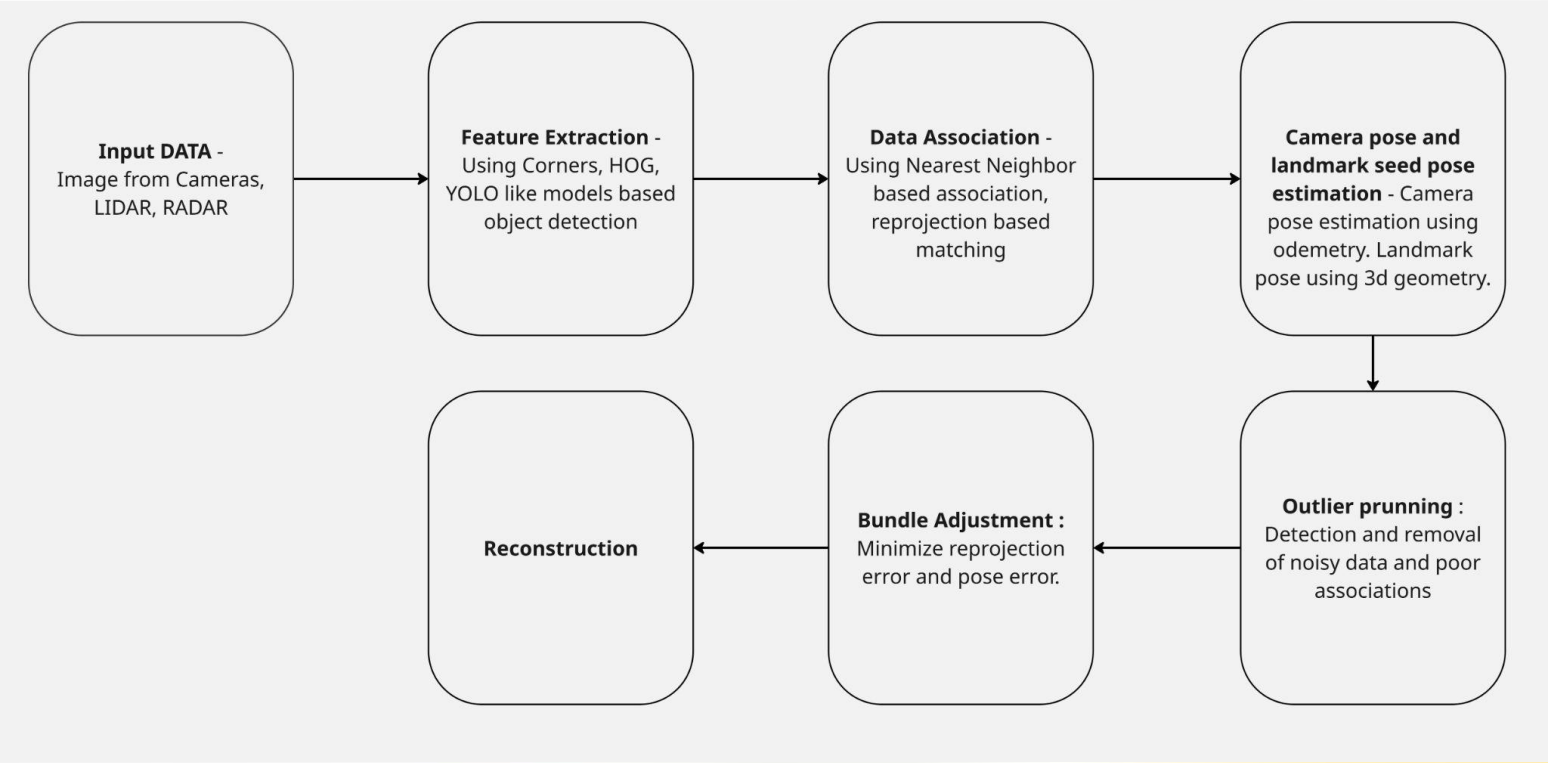



http://www.theia-sfm.org/sfm.html




Garrard-Hall using GLOMAP
https://medium.com/@sugunsegu/understanding-glomap-global-structure-from-motion-64b052be7927

# What is Visual Reconstruction?



# Bundle Adjustment in the Large (BAL)

- Benchmark for bundle adjustment
- Provides realistic 3D reconstruction problems from images
- Includes ground-truth data association for fair comparisons
- Designed for mid to large-scale optimization problems

## Data Format

```
<num_cameras> <num_points> <num_observations>
<camera_index_1> <point_index_1> <x_1> <y_1>
...
<camera_index_num_observations> <point_index_num_observations> <x_num_observations> <y_num_observations>
<camera_1>
...
<camera_num_cameras>
<point_1>
...
<point_num_points>
```

https://grail.cs.washington.edu/projects/bal/

# Bundle Adjustment in the Large (BAL)



## Data Format

- Observations are associated with landmarks.
  - Single observation per landmark per camera.
- Seed camera poses with intrinsics.
- Seed landmark poses.

```
<num_cameras> <num_points> <num_observations>
<camera_index_1> <point_index_1> <x_1> <y_1>
...
<camera_index_num_observations> <point_index_num_observations> <x_num_observations> <y_num_observations>
<camera_1>
...
<camera_num_cameras>
<point_1>
...
<point_num_points>
```

https://grail.cs.washington.edu/projects/bal/

# Bundle Adjustment in the Large (BAL)

**Ladybug**

Cameras : 1723

Points : 156502

Observations : 678718

Reprojection error

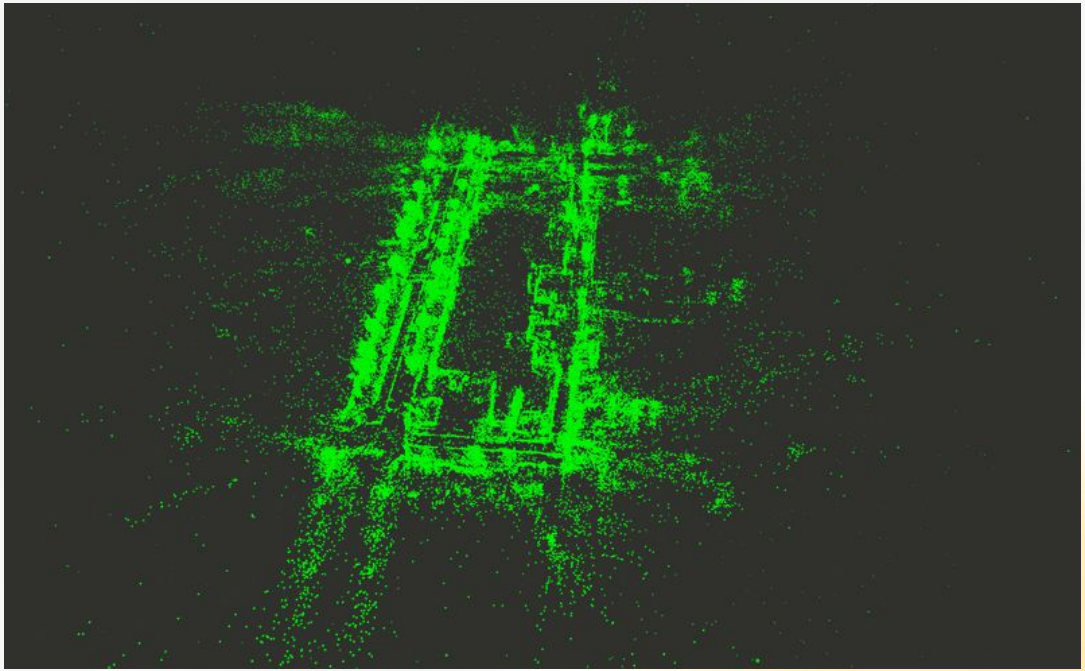
- μ=3.86, σ=8.61, med=1.67, max = 1693.285

Observations per Camera

- μ=393.6, σ=195.87, med=395, max=970

Observations per Point

- μ=4.34, σ=4.99, med=3, r=[2.000, 145.00]



https://grail.cs.washington.edu/projects/bal/

# Bundle Adjustment in the Large (BAL)

**Trafalgar Square dataset**

Cameras : 257

Points : 65132

Observations : 225911

Reprojection error

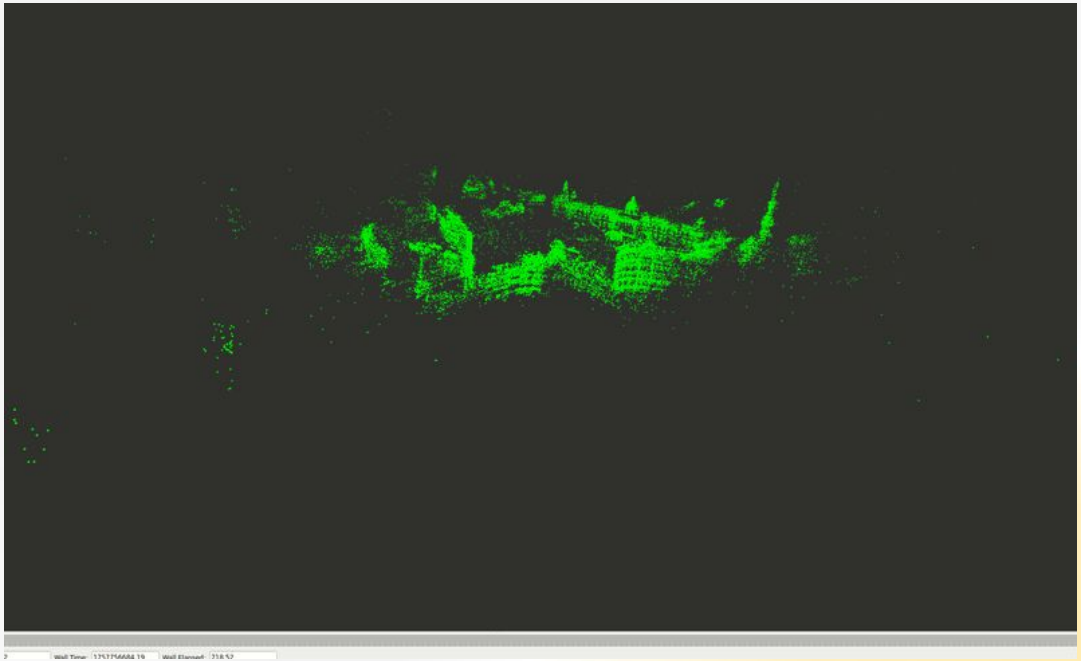
- μ=8.3, σ=12.2, med=3.5, max=456.01

Observations per Camera

- μ=876.5, σ=792.29, med=625, max= 4173

Observations per Point

- μ=3.47, σ=3.2, med=2, range=[2, 42]



https://grail.cs.washington.edu/projects/bal/

# Bundle Adjustment in the Large (BAL)

## Challenges

- Large Dataset, risks high processing time
  - Ladybug has 1723 images with 678718 observations
  - Trafalgar Square has 257 images with 225911 observations
- Noisy data and outlets can lead to non-convex optimization.

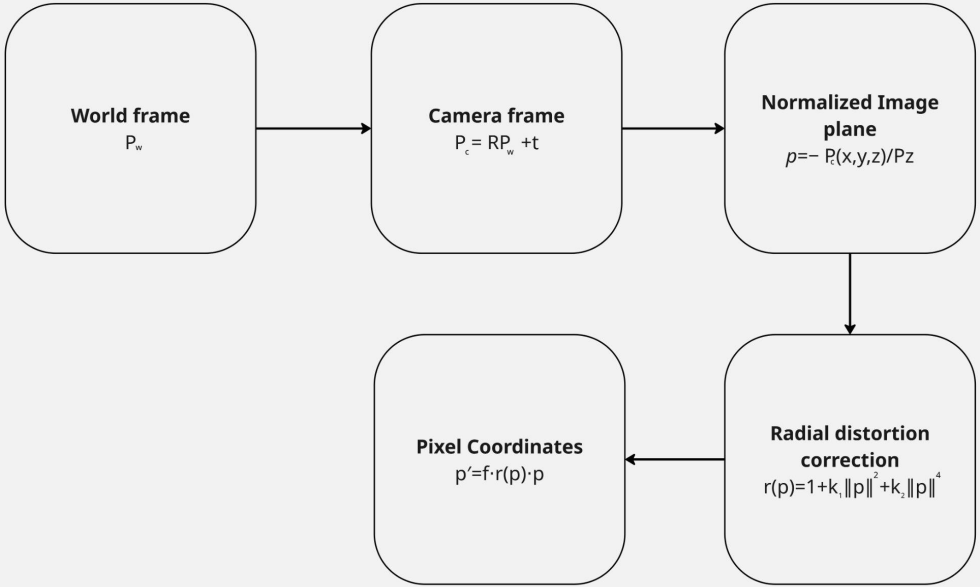
https://grail.cs.washington.edu/projects/bal/

# BAL Camera Model

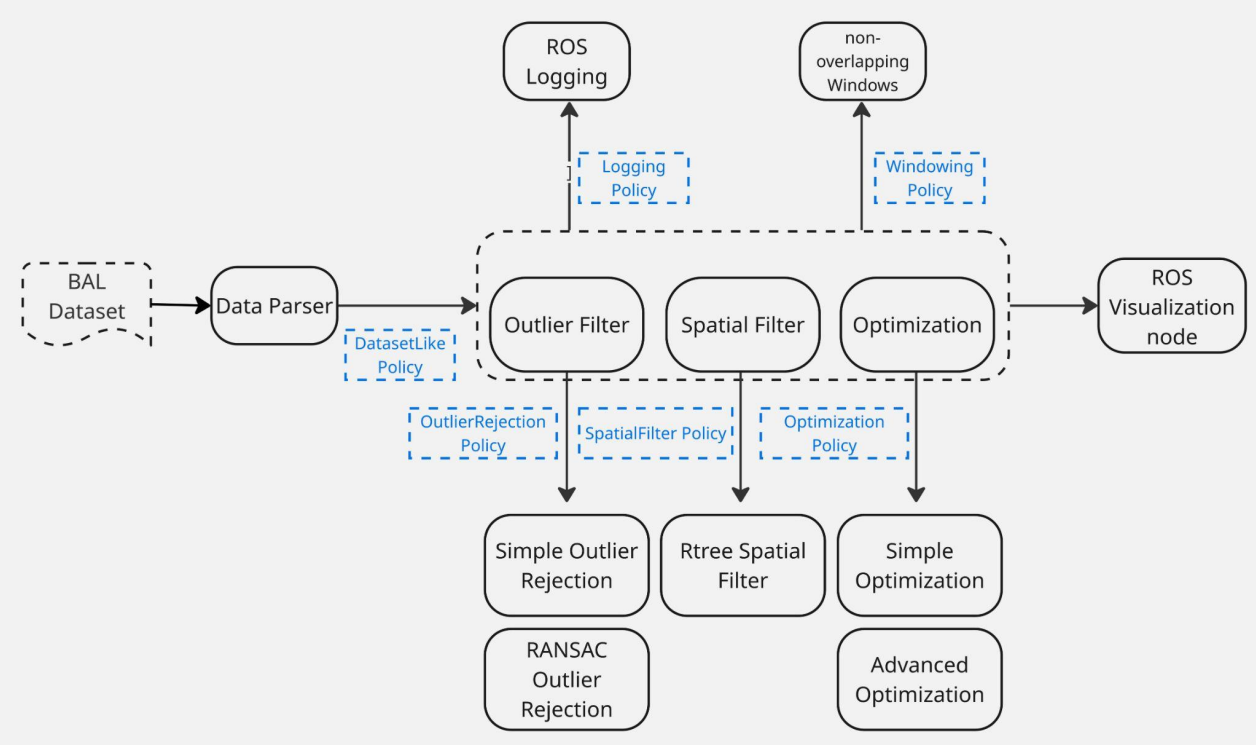
## Parameters Estimated per Camera

- Rotation R, Translation t
- Focal length f
- Radial distortion k1,k2

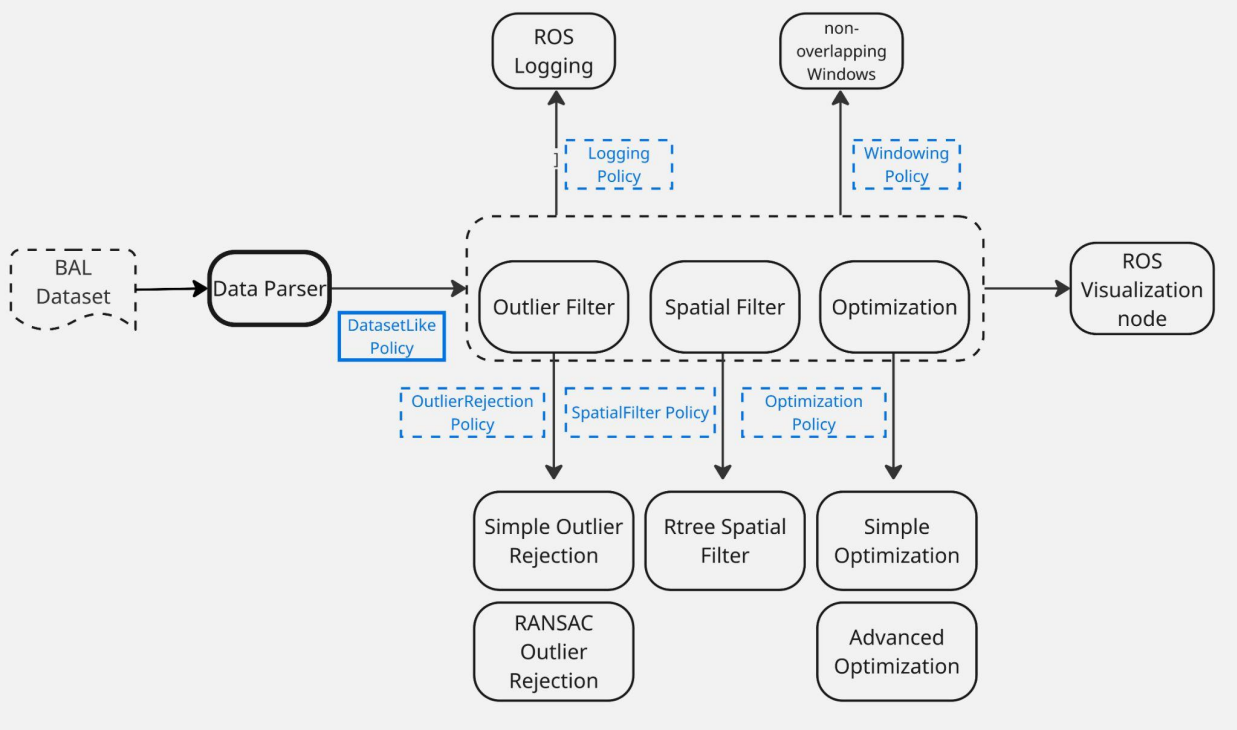
## Projection Pipeline



# SFM Pipeline



# SFM Pipeline



# Data Parser

```cpp
[[nodiscard]] std::expected<BALDataType, ParseError> extract_data()
{
  try
  {
    auto cameras_landmarks = load_data();
    if (!cameras_landmarks)
    {
      return std::unexpected{cameras_landmarks.error()};
    }
    auto [cameras, landmarks] = std::move(*cameras_landmarks);

    if (config_.perform_filtering)
    {
      filter_observations(cameras, landmarks);
    }

    if (config_.normalize_data)
    {
      normalize(cameras, landmarks);
    }

    if (cameras.empty() || landmarks.empty())
    {
      return std::unexpected{ParseError::InsufficientData};
    }

    auto result = convert_to_policy_format(cameras, landmarks);
    return result;

  } catch (const std::exception&)
  {
    return std::unexpected{ParseError::CorruptedData};
  }
}
```

- nodiscard used to prevent accidental call to extract_data()
- Return type wrapped with std::expected to accommodate exception handling
- Performs filtering, normalization and conversion to policy based datatype.

# Data Parser : preprocessing

```cpp
void filter_observations(CameraVector& cameras, LandmarkVector& landmarks) noexcept
{
  if (config_.depth_threshold <= 0.0) return;
  auto observation_filter = [this, &cameras](auto& landmark)
  {
      std::unordered_map<int, typename Traits::Vec2> valid_obs;
      for (const auto& [cam_idx, obs] : landmark.obs)
      {
          if (cam_idx < 0 || cam_idx >= static_cast<int>(cameras.size()))
          {
              continue;
          }

          const auto& camera = cameras[cam_idx];
          auto p_cam = camera.T_c_w * landmark.p_w;

          if (is_valid_observation(p_cam, config_.depth_threshold))
          {
              valid_obs.emplace(cam_idx, obs);
          }
      }

      landmark.obs = std::move(valid_obs);
  };

  std::for_each(std::execution::par, landmarks.begin(), landmarks.end(), observation_filter);

  auto insufficient_observations = [this](const auto& landmark)
  {
      return landmark.observation_count() < config_.min_observations_per_landmark;
  };

  std::erase_if(landmarks, insufficient_observations);
}
```

- Parallel execution using for::each to speed up the processing (c++ 17).
    - Check if the depth of the landmark in camera frame is positive.
- std::erase_if used to remove any empty landmarks.

# Data Parser : preprocessing



```cpp
void normalize(CameraVector& cameras, LandmarkVector& landmarks) noexcept {
    if (landmarks.empty()) return;

    typename Traits::Vec3 median = Traits::Vec3::Zero();

    for (int dim = 0; dim < 3; ++dim) {
        auto coords = landmarks | std::views::transform([dim](const auto& landmark) { return landmark.p_w(dim);});
        median(dim) = compute_median(coords);
    }

    auto deviations = landmarks
        | std::views::transform([&median](const auto& landmark) {
            return (landmark.p_w - median).template lpNorm<1>();});

    Scalar median_dev = compute_median(deviations);
    Scalar scale_factor = static_cast<Scalar>(config_.scale) / median_dev;

    auto landmark_normalizer = [scale_factor, &median](auto& landmark) {
        landmark.p_w = scale_factor * (landmark.p_w - median);
    };

    auto camera_normalizer = [scale_factor, &median](auto& camera) {
        auto T_w_c = camera.T_c_w.inverse();
        T_w_c.translation() = scale_factor * (T_w_c.translation() - median);
        camera.T_c_w = T_w_c.inverse();
    };

    std::for_each(std::execution::par, landmarks.begin(), landmarks.end(), landmark_normalizer);
    std::for_each(std::execution::par, cameras.begin(), cameras.end(), camera_normalizer);
}
```

- Normalizes using Median absolute deviation(MAD).
  - MAD uses median which is more resilient to outliers.
  - Value becomes a factor of median deviation.
- Parallelized std::for_each using intel TBB.

# Concepts : DatasetLike Policy

```cpp
template<typename T>
concept CameraModelLike = requires(T model, const Eigen::Vector3d& point)
{
  {model.project(point)} -> std::convertible_to<Eigen::Vector2d>;
  {model.focal_length()} -> std::convertible_to<Eigen::Vector2d>;
  {model.principal_point()} -> std::convertible_to<Eigen::Vector2d>;
  {model.distortion_coefficients()} -> std::convertible_to<Eigen::Vector2d>;
};

template<typename T>
concept CameraLike = requires(T camera)
{
  typename T::Traits;
  typename T::CameraModelType;

  { camera.intrinsics } -> CameraModelLike;
  { camera.translation() } -> std::convertible_to<typename T::Traits::Vec3>;
  { camera.rotation() } -> std::convertible_to<typename T::Traits::SO3>;
  { camera.id } -> std::convertible_to<int>;
};

template<typename T>
concept ObservationLike = requires(T obs)
{
  typename T::Traits;
  { obs.camera_index } -> std::convertible_to<int>;
  { obs.point_index } -> std::convertible_to<int>;
  { obs.pixel_coordinates() } -> std::convertible_to<typename T::Traits::Vec2>;

  { obs.outlier } -> std::convertible_to<bool>;
  { obs.outlier_reason } -> std::convertible_to<std::string>;
};
```

**CameraModelLike**

**function :**
- project()

**ObservationLike**

**variable :**
- camera_index
- point_index
- outlier
- outlier_reason

**function :**
- pixel_coordinates()

**CameraLike**

**variable :**
- intrinsics
- id

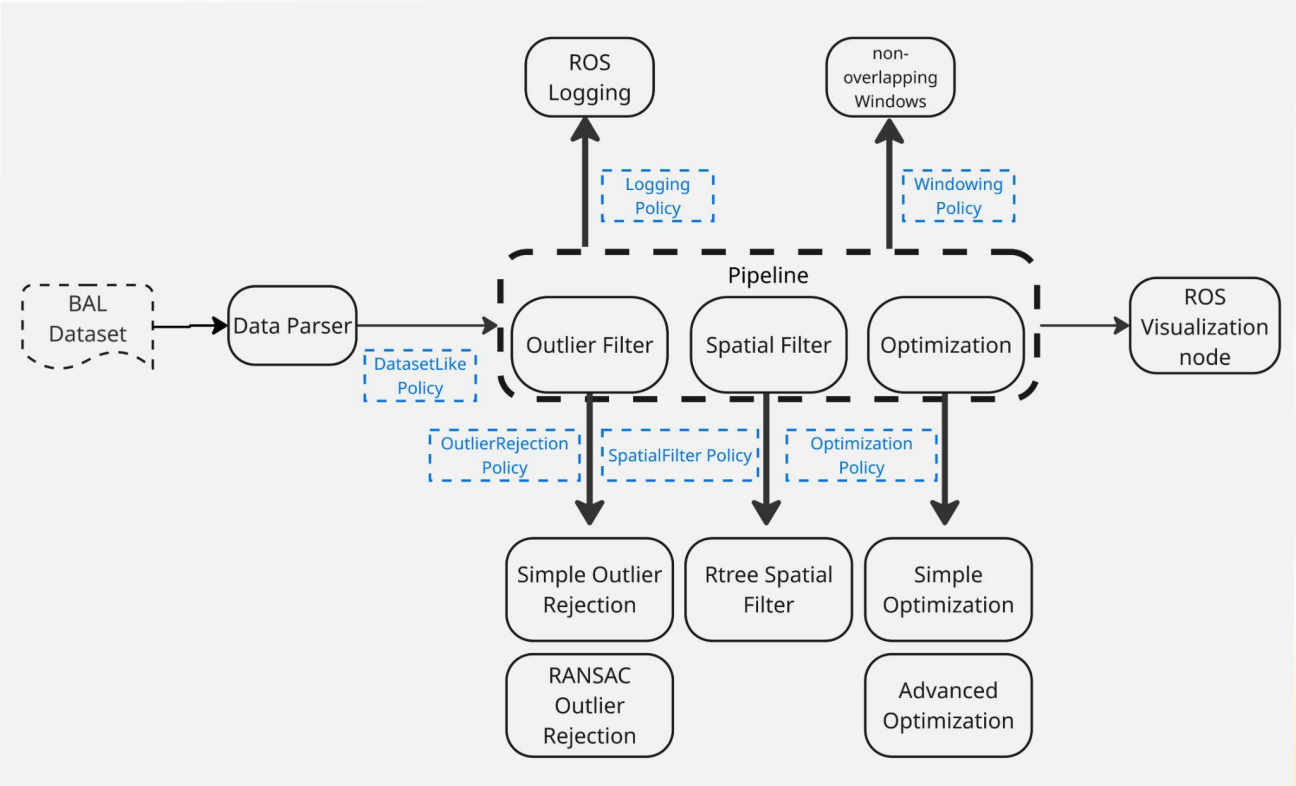
**function :**
- translation()
- rotation()

# Concepts : DatasetLike Policy

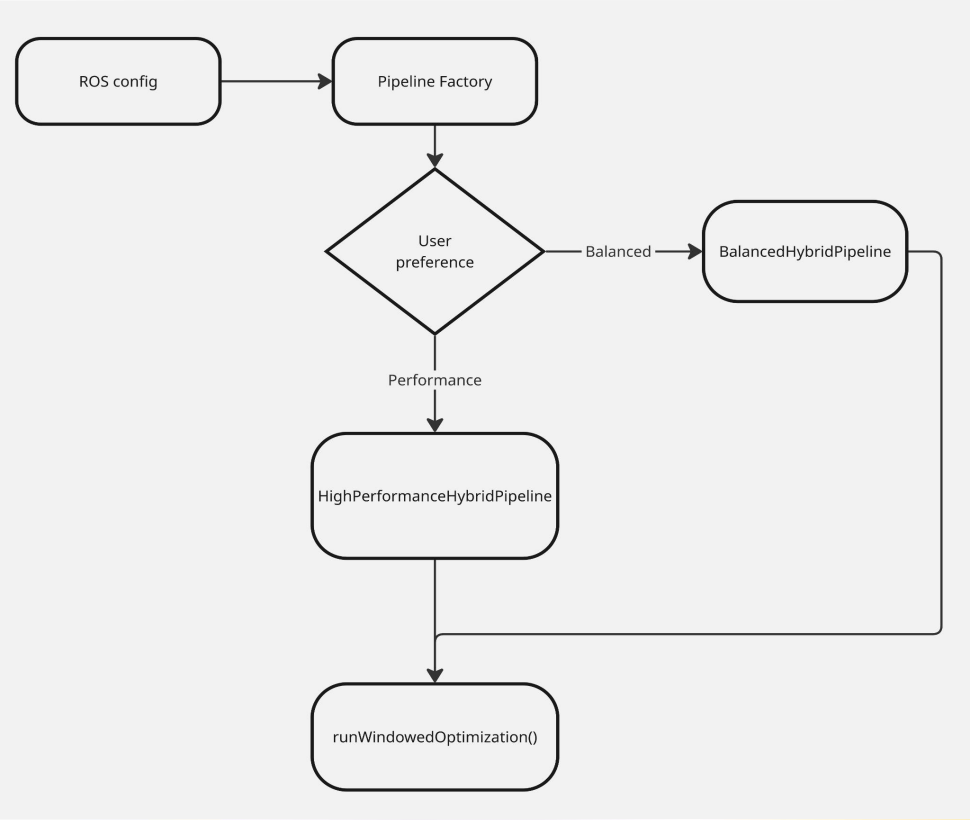

```cpp
template<typename T>
concept DatasetLike = requires(T dataset)
{
  typename T::ScalarType;
  typename T::ObservationType;
  typename T::CameraType;
  typename T::Traits;

  // Functions for modifying data
  { dataset.observations() } -> std::same_as<std::vector<typename T::ObservationType>&>;
  { dataset.cameras() } -> std::same_as<std::vector<typename T::CameraType>&>;
  { dataset.points() } -> std::same_as<std::vector<typename T::Vec3>&>;

  // Functions for analysis
  { dataset.num_observations() } -> std::convertible_to<size_t>;
  { dataset.num_cameras() } -> std::convertible_to<size_t>;
  { dataset.num_points() } -> std::convertible_to<size_t>;
  { dataset.empty() } -> std::convertible_to<bool>;

  // maps for easy access
  { dataset.camera_to_observations_map() } -> std::same_as<std::unordered_map<int, std::vector<int>>&>;
  { dataset.point_to_observations_map() } -> std::same_as<std::unordered_map<int, std::vector<int>>&>;

  // concepts for the data types
  requires ObservationLike<typename T::ObservationType>;
  requires CameraLike<typename T::CameraType>;
};
```

- std::same_as is used as a strict requirement.
- std::convertible_to is more flexible, just needs the return type to be convertible to the specified type.
- Use *requires* to specify the datatype.

# Pipeline : Setup



# Pipeline : Factory



# Pipeline : Factory

```cpp
/**
 * @brief Hybrid Pipeline class for bundle adjustment
 */
template<
    typename Scalar,
    typename DatasetType,
    typename WindowingPolicyType,
    typename SpatialFilterPolicyType,
    typename OutlierRejectionPolicyType,
    typename OptimizationPolicyType>
requires
    DatasetLike<DatasetType> &&
    policies::WindowingPolicy<WindowingPolicyType, DatasetType> &&
    policies::SpatialFilterPolicy<SpatialFilterPolicyType, DatasetType> &&
    policies::OutlierRejectionPolicy<OutlierRejectionPolicyType, DatasetType> &&
    policies::OptimizationPolicy<OptimizationPolicyType, DatasetType>
class HybridPipeline {
public:
    /**
     * @brief Constructor to initialize the pipeline
     */
    explicit HybridPipeline(std::shared_ptr<rclcpp::Node> node, DatasetType data) noexcept
    : node_(node),
      optimized_data_(data),
      data_(std::move(data)),
      config_(loadHybridPipelineConfig(node))
    {
      data_stats_ = std::make_unique<DataStats<DatasetType>>(node_, config_.outlier_threshold, config_.verbose_logging);
      policies::ROSLoggingPolicy logging_policy(node_);
      windowing_policy_ = std::make_unique<WindowingPolicyType>(node_, logging_policy);

      spatial_filter_policy_ = std::make_unique<SpatialFilterPolicyType>(node_, logging_policy);
      outlier_rejection_policy_ = std::make_unique<OutlierRejectionPolicyType>(node_, logging_policy);
      optimization_policy_ = std::make_unique<OptimizationPolicyType>(node_, logging_policy);
    }

    virtual ~HybridPipeline() = default;
```

- Concepts used to specify requirements on templated types.

# Pipeline : Factory

```cpp
/**
 * @brief Balanced pipeline configuration
 */
template<typename Scalar = double>
class BalancedHybridPipeline : public HybridPipeline<
    Scalar,
    BALData<Scalar>,
    policies::implementations::NonOverlappingWindows<policies::ROSLoggingPolicy>,
    policies::implementations::RTreeSpatialFilter<policies::ROSLoggingPolicy>,
    policies::implementations::SimpleOutlierRejection<policies::ROSLoggingPolicy>,
    policies::implementations::CeresOptimization<policies::ROSLoggingPolicy>>
{
public:
    explicit BalancedHybridPipeline(std::shared_ptr<rclcpp::Node> node, BALData<Scalar> data) noexcept
        : HybridPipeline<Scalar,
                         BALData<Scalar>,
                         policies::implementations::NonOverlappingWindows<policies::ROSLoggingPolicy>,
                         policies::implementations::RTreeSpatialFilter<policies::ROSLoggingPolicy>,
                         policies::implementations::SimpleOutlierRejection<policies::ROSLoggingPolicy>,
                         policies::implementations::CeresOptimization<policies::ROSLoggingPolicy>>(node, std::move(data))
    {}

    std::string getPipelineTypeName() const {
        return "Balanced HybridPipeline";
    }
};

/**
 * @brief High-performance pipeline configuration
 */
template<typename Scalar = double>
class HighPerformanceHybridPipeline : public HybridPipeline<
    Scalar,
    BALData<Scalar>,
    policies::implementations::NonOverlappingWindows<policies::ROSLoggingPolicy>,
    policies::implementations::RTreeSpatialFilter<policies::ROSLoggingPolicy>,
    policies::implementations::SimpleOutlierRejection<policies::ROSLoggingPolicy>,
    policies::implementations::AdvancedOptimization<policies::ROSLoggingPolicy>>
{
public:
    explicit HighPerformanceHybridPipeline(std::shared_ptr<rclcpp::Node> node, BALData<Scalar> data) noexcept
        : HybridPipeline<Scalar,
                         BALData<Scalar>,
                         policies::implementations::NonOverlappingWindows<policies::ROSLoggingPolicy>,
                         policies::implementations::RTreeSpatialFilter<policies::ROSLoggingPolicy>,
                         policies::implementations::SimpleOutlierRejection<policies::ROSLoggingPolicy>,
                         policies::implementations::AdvancedOptimization<policies::ROSLoggingPolicy>>(node, std::move(data))
    {}

    std::string getPipelineTypeName() const {
        return "High-Performance HybridPipeline";
    }
};
```

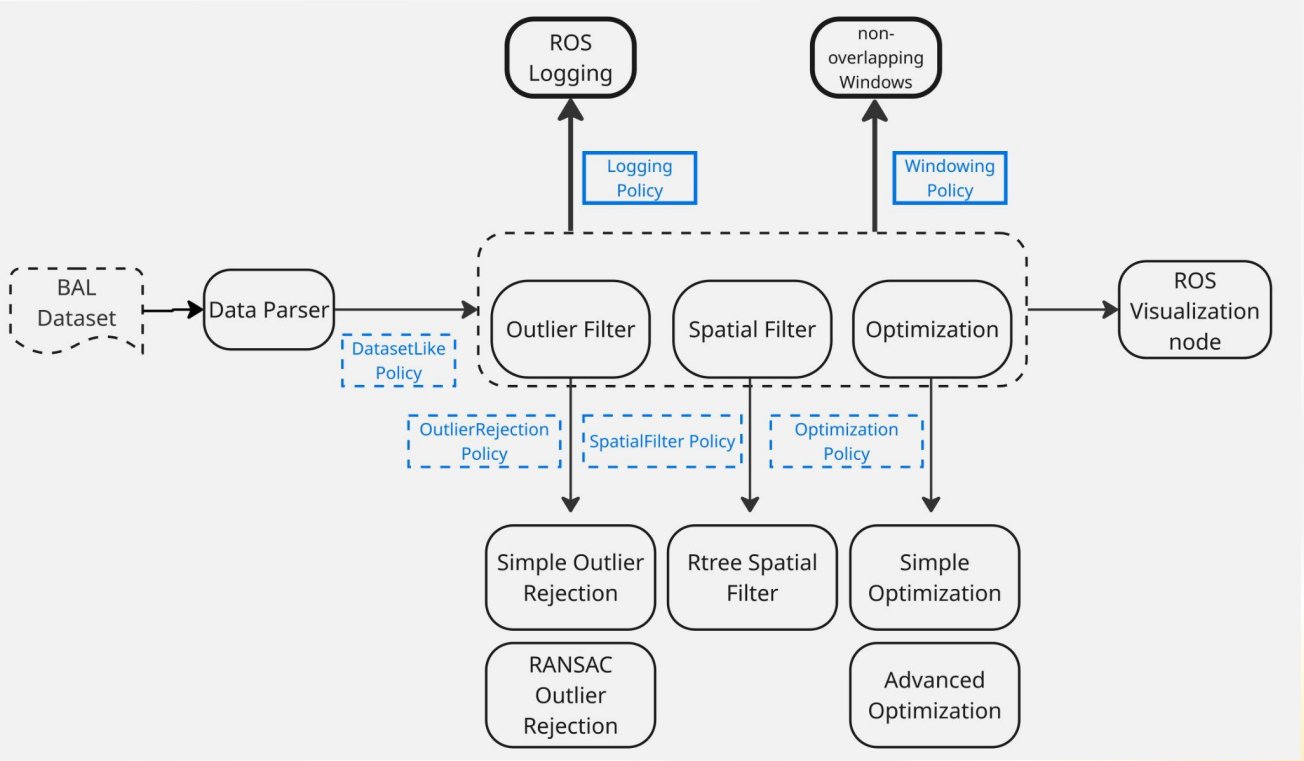
- Derived classes with pre-set template arguments.
- User can select at runtime using ROS params.

```cpp
if (pipeline_type == "balanced")
{
  auto pipeline = bal_cppcon::HybridPipelineFactory<double>::createBalancedPipeline(node, data);
  runPipeline(node, pipeline, data);
} else if (pipeline_type == "high_performance")
{
  auto pipeline = bal_cppcon::HybridPipelineFactory<double>::createHighPerformancePipeline(node, data);
  runPipeline(node, pipeline, data);
}
```

# Pipeline : main

```cpp
void runWindowedOptimization()
{
  // prepare windows based on camera poses
  auto windowing_result = windowing_policy_->generateWindows(optimized_data_);
  const auto& windows = windowing_result.windows;

  for (const auto& window : windows)
  {
    // filter observations in the window using outlier and spatial filter policies
    std::span<const int> camera_window_span(window.camera_indices);
    auto outlier_result = outlier_rejection_policy_->rejectOutliers(optimized_data_, camera_window_span);
    auto spatial_result = spatial_filter_policy_->filterSpatially(optimized_data_, camera_window_span);
  }

  for (const auto& window : windows)
  {
    // optimize the window using the optimization policy
    std::span<const int> camera_window_span(window.camera_indices);
    auto optimization_result = optimization_policy_->optimize(optimized_data_, camera_window_span);
```

# Logging and window policy



# Logging and window policy

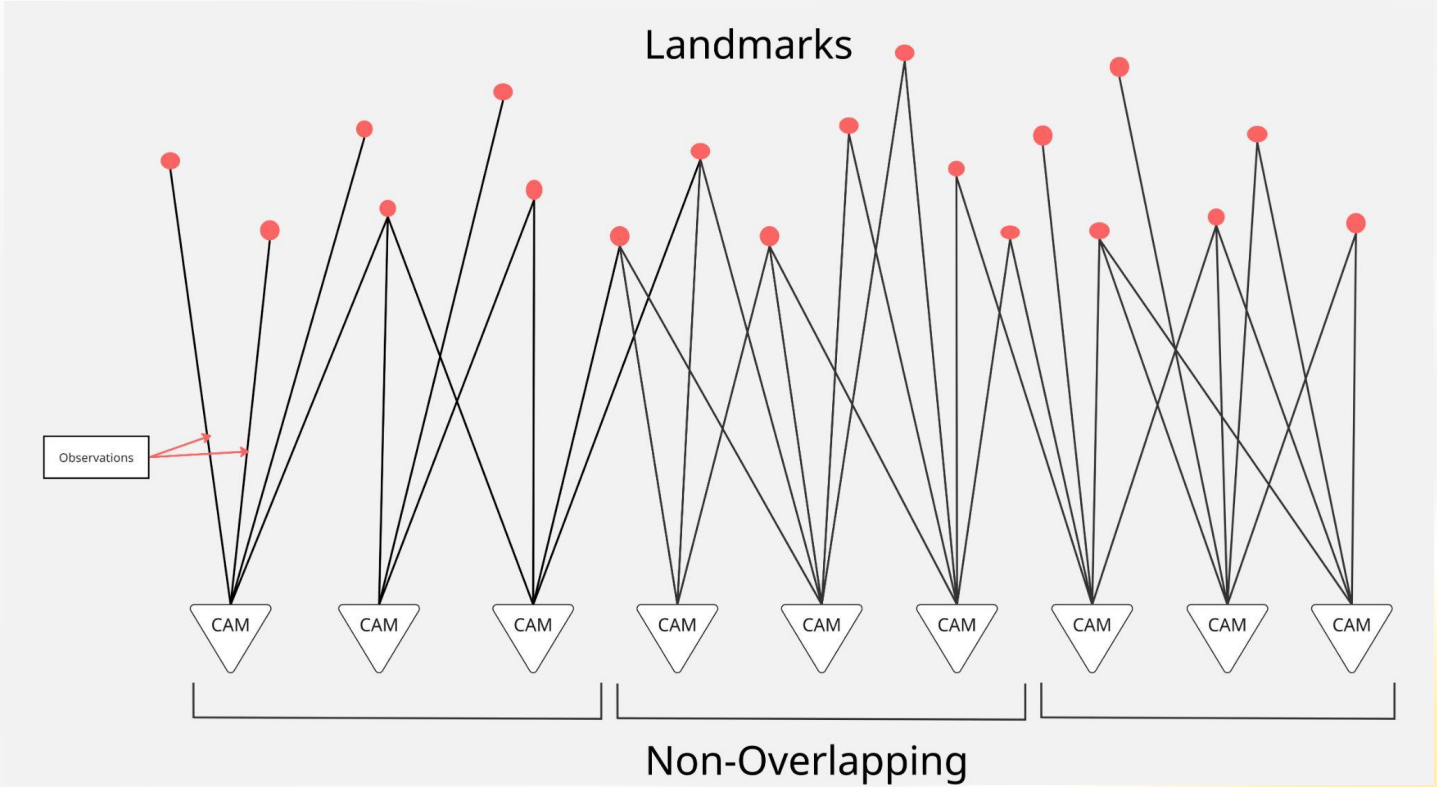


```cpp
enum class LogLevel : std::uint8_t
{
  DEBUG = 0,
  INFO  = 1,
  WARN  = 2,
  ERROR = 3
};

template<typename Policy>
concept LoggingPolicy = requires(Policy policy, LogLevel level, std::string_view message)
{
  { policy.log(level, message) } -> std::same_as<void>;
};
```

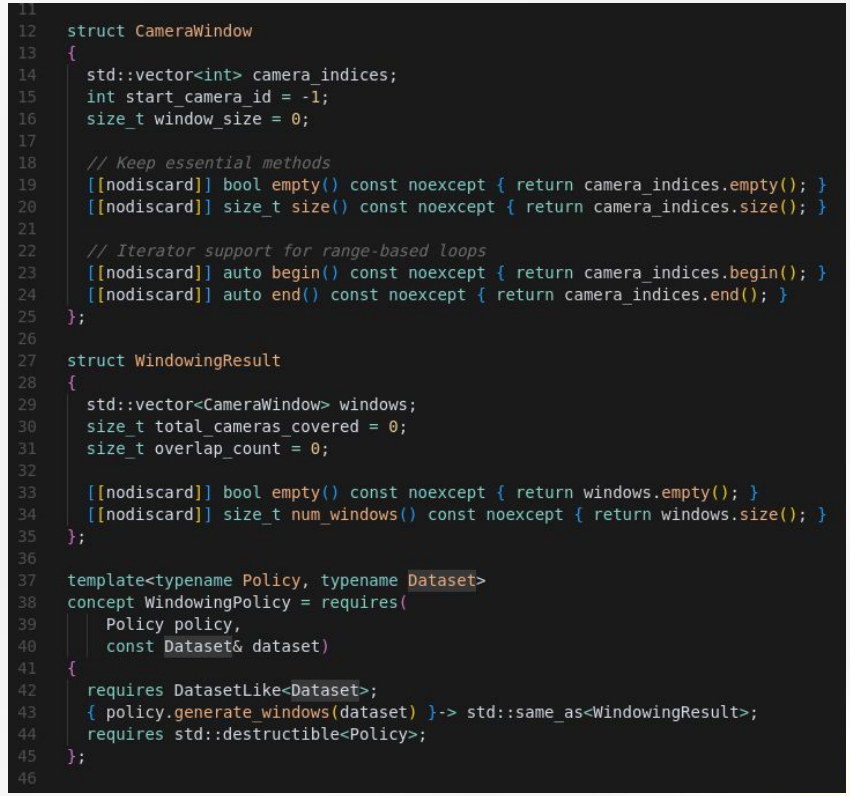
```cpp
// ROS Logging Policy
class ROSLoggingPolicy
{
public:
    explicit ROSLoggingPolicy(std::shared_ptr<rclcpp::Node> node) noexcept
        : node_(std::move(node)) {}

    // Satisfy LoggingPolicy concept
    void log(LogLevel level, std::string_view message) noexcept
    {
        if (!node_)return;

        switch (level) {
            case LogLevel::DEBUG:
                RCLCPP_DEBUG(node_->get_logger(), "%s", message.data());
                break;
            case LogLevel::INFO:
                RCLCPP_INFO(node_->get_logger(), "%s", message.data());
                break;
            case LogLevel::WARN:
                RCLCPP_WARN(node_->get_logger(), "%s", message.data());
                break;
            case LogLevel::ERROR:
                RCLCPP_ERROR(node_->get_logger(), "%s", message.data());
                break;
        }
    }
private:
  std::shared_ptr<rclcpp::Node> node_;
};

// Verify that ROSLoggingPolicy satisfies the concept
static_assert(LoggingPolicy<ROSLoggingPolicy>);
```

# Logging and window policy





# Logging and window policy



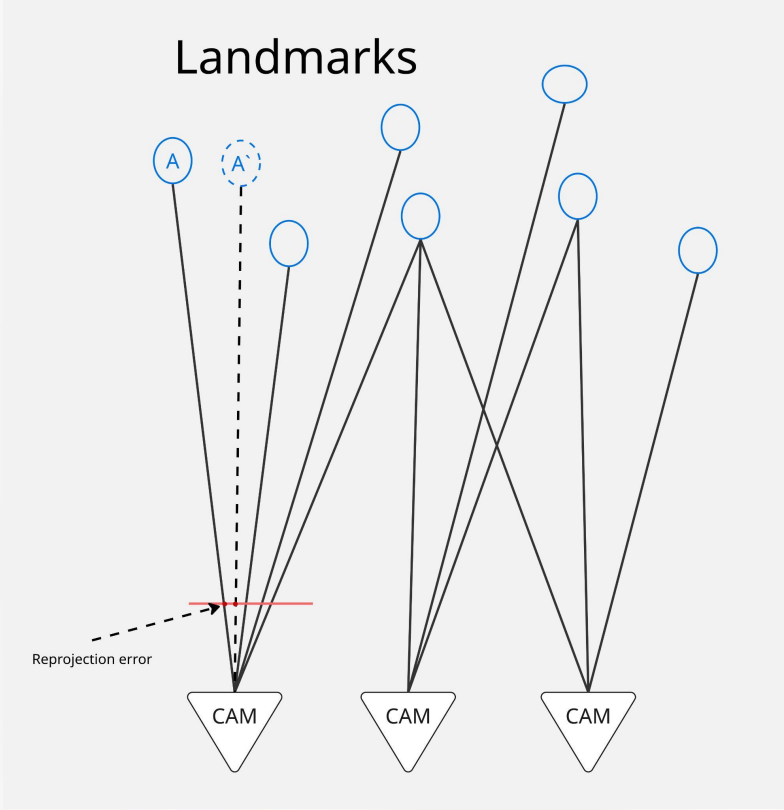
```cpp
struct CameraWindow
{
  std::vector<int> camera_indices;
  int start_camera_id = -1;
  size_t window_size = 0;

  // Keep essential methods
  [[nodiscard]] bool empty() const noexcept { return camera_indices.empty(); }
  [[nodiscard]] size_t size() const noexcept { return camera_indices.size(); }

  // Iterator support for range-based loops
  [[nodiscard]] auto begin() const noexcept { return camera_indices.begin(); }
  [[nodiscard]] auto end() const noexcept { return camera_indices.end(); }
};

struct WindowingResult
{
  std::vector<CameraWindow> windows;
  size_t total_cameras_covered = 0;
  size_t overlap_count = 0;

  [[nodiscard]] bool empty() const noexcept { return windows.empty(); }
  [[nodiscard]] size_t num_windows() const noexcept { return windows.size(); }
};

template<typename Policy, typename Dataset>
concept WindowingPolicy = requires(
    Policy policy,
    const Dataset& dataset)
{
  requires DatasetLike<Dataset>;
  { policy.generate_windows(dataset) }-> std::same_as<WindowingResult>;
  requires std::destructible<Policy>;
};
```

# Pipeline - outlier filtering model



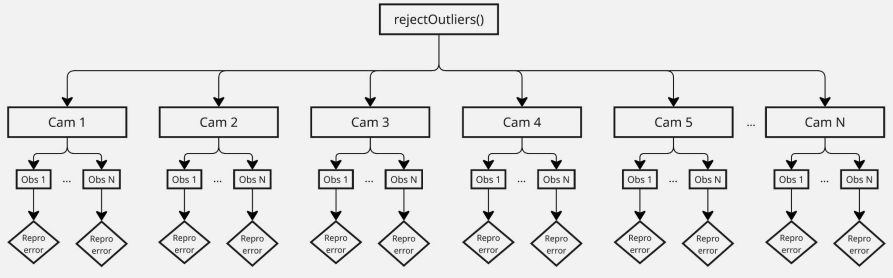
```
14
15  struct OutlierRejectionResult
16  {
17    size_t total_observations_processed{0};
18    size_t total_inliers_found{0};
19
20    // Default constructor
21    OutlierRejectionResult() = default;
22
23    // Statistics
24    [[nodiscard]] double inlier_ratio() const noexcept {
25        return total_observations_processed > 0 ?
26                    static_cast<double>(total_inliers_found) / total_observations_processed : 0.0;
27    }
28  };
29
30  // outlier rejection policy
31  template<typename Policy, typename Dataset>
32  concept OutlierRejectionPolicy = requires(
33      Policy policy,
34      Dataset& dataset,
35      std::span<const int> camera_window)
36  {
37    requires DatasetLike<Dataset>;
38    // Main outlier rejection method
39    { policy.rejectOutliers(dataset, camera_window) } -> std::same_as<OutlierRejectionResult>;
40
41    // Stats
42    { policy.get_algorithm_name() } -> std::convertible_to<std::string_view>;
43    { policy.get_statistics() } -> std::convertible_to<std::string>;
44
45      requires std::destructible<Policy>;
46  };
47
```

# Pipeline - Simple outlier filtering



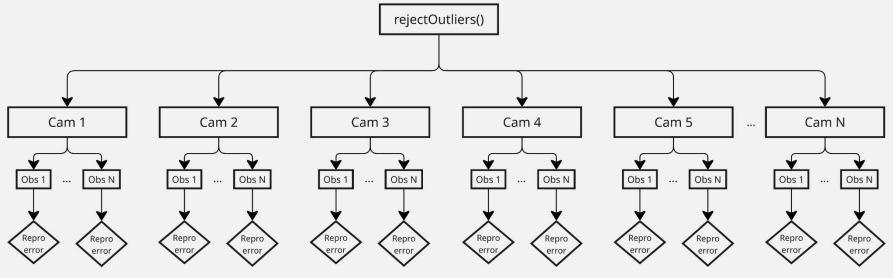
# Pipeline - Simple outlier filtering

```cpp
// OutlierRejectionPolicy interface
OutlierRejectionResult rejectOutliers(DatasetLike auto& dataset,
                                      std::span<const int> camera_window)
{
  bal_cppcon::policies::concepts::OutlierRejectionResultInternal result_internal;

  // Process each camera in the window
  processCamerasParallel(dataset, camera_window, result_internal);
  OutlierRejectionResult result = result_internal.to_result();
  update_statistics(result);
  return result;
}

void processCamerasParallel(
    DatasetLike auto& dataset,
    std::span<const int> camera_window,
    bal_cppcon::policies::concepts::OutlierRejectionResultInternal& result)
{
  auto window_cameras = camera_window |
                        std::views::transform([&dataset](int idx) {
                          return dataset.cameras()[idx];});

  // Process each camera in parallel
  std::for_each(std::execution::par, window_cameras.begin(),
                window_cameras.end(),
                [this, &dataset, &result](const auto& camera) {
                  processSingleCamera(dataset, camera, result);});
}
```



# Pipeline - Simple outlier filtering

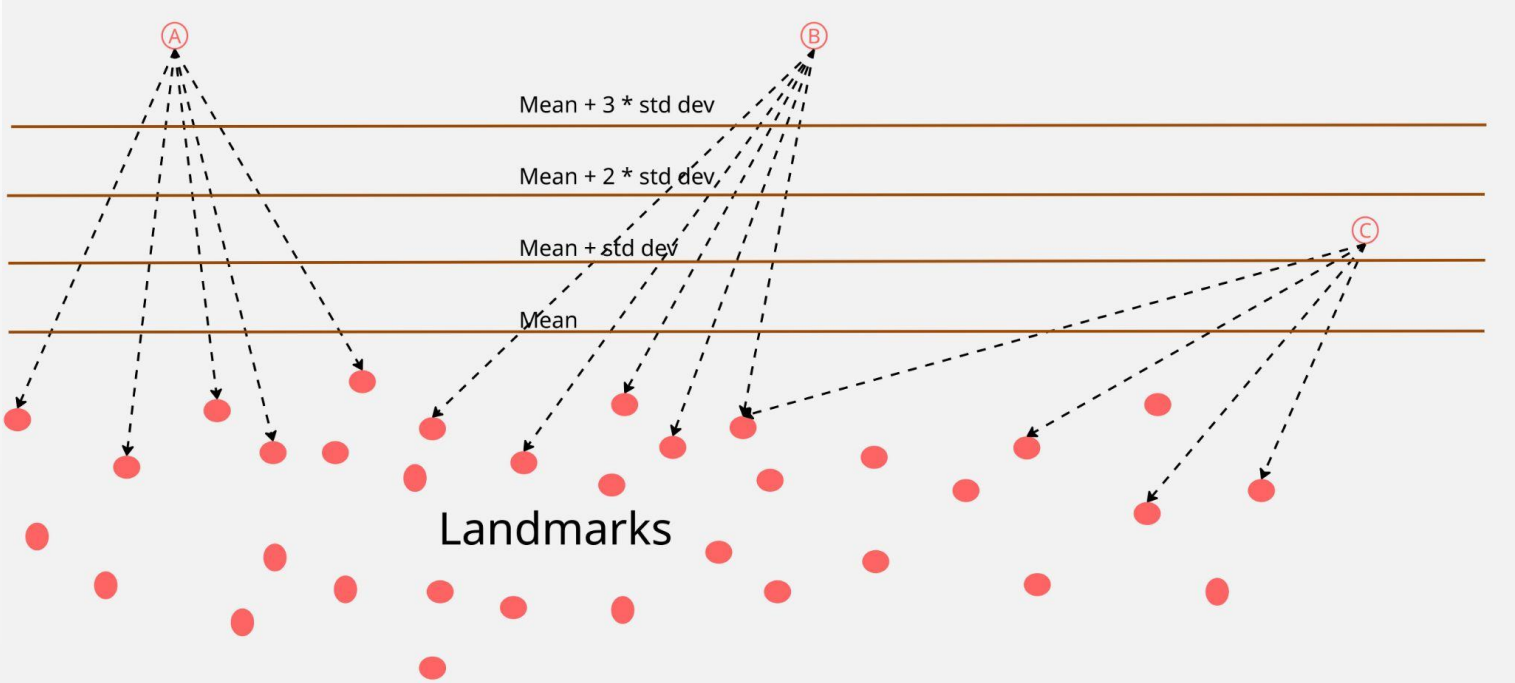
```cpp
// Get mutable access to observations to update outlier flags
auto& observations = dataset.observations();

// Compute reprojection errors for all observations
std::vector<std::pair<int, double>> error_pairs(obs_indices_set.size());

std::transform(std::execution::par, obs_indices_set.begin(),
               obs_indices_set.end(), error_pairs.begin(),
               [this, &dataset, &camera](int obs_idx) {
                 return compute_reprojection_error(dataset, camera, obs_idx);});

// Update outlier flags in observations and count inliers
std::atomic<size_t> inlier_count{0};

std::for_each(std::execution::par, error_pairs.begin(), error_pairs.end(),
              [this, &observations, &inlier_count](const auto& pair) {
                const auto& [obs_idx, error] = pair;
                if (obs_idx >= 0 &&
                    obs_idx < static_cast<int>(observations.size())) {
                  auto& obs = observations[obs_idx];

                  // Determine if observation is an outlier based on reprojection error
                  bool is_outlier = error >= config_.reprojection_threshold;

                  // Update the observation's outlier status
                  obs.outlier = is_outlier;
                  if (is_outlier) {
                    obs.outlier_reason = "simple_outlier_rejection";
                  } else {
                    obs.outlier_reason = "";
                    inlier_count.fetch_add(1);
                  }
                }
              });
```



# Pipeline - Spatial Filtering

```cpp

//  spatial filter result
struct SpatialFilterResult {
    size_t total_observations_processed{0};
    size_t total_observations_kept{0};

    SpatialFilterResult() = default;

    // Statistics
    [[nodiscard]] double retention_ratio() const noexcept {
        return total_observations_processed > 0 ?
               static_cast<double>(total_observations_kept) / total_observations_processed : 0.0;
    }
};

//spatial filter policy concept
template<typename Policy, typename Dataset>
concept SpatialFilterPolicy = requires(
    Policy policy,
    Dataset& dataset,
    std::span<const int> camera_window)
{
    requires DatasetLike<Dataset>;

    // Main spatial filtering method
    { policy.filterSpatially(dataset, camera_window) }
        -> std::same_as<SpatialFilterResult>;

    // stats
    { policy.getLastStatistics() } -> std::convertible_to<std::string>;

    requires std::destructible<Policy>;
};
```

# Pipeline - Spatial Filtering



# Pipeline - Spatial Filtering

```cpp
template <typename Dataset>
requires DatasetLike<Dataset>
void buildSpatialIndex(Dataset& dataset)
{
  std::lock_guard<std::mutex> lock(mutex_);

  if (config_.verbose_logging)
  {
    logging_policy_.log(LogLevel::INFO, "Building R-tree spatial index");
  }

  rtree_.clear();

  // build rtree with points as spatial index
  size_t count = 0;
  const auto& points = dataset.points();
  for (const auto& point : points)
  {
    if (point.size() >= 3)
    {
      rtree_.insert(std::make_pair(Point3D(point[0], point[1], point[2]), count++));
    }
  }

  rtree_built_ = true;

  if (config_.verbose_logging)
  {
    logging_policy_.log(LogLevel::INFO,
      "R-tree built with " + std::to_string(count) + " points");
  }
}
```

- Rtree used to store 3d points for quick nearest neighbour query.
- Groups nearby points into bounding boxes, forming a tree structure.
- Complexity for querying is O(logn).

# Pipeline - Spatial Filtering

```
223         for (int obs_idx : obs_indices_set)
224         {
225           if (obs_idx >= 0 && obs_idx < static_cast<int>(observations.size()))
226           {
227             const auto& obs = observations[obs_idx];
228             if (!obs.outlier) {  // Only process inliers
229                 inlier_obs_indices.push_back(obs_idx);
230             }
231           }
232         }
233
234         // Process each inlier observation for this camera in parallel and update outlier flags
235         std::atomic<size_t> kept_count{0};
236         std::for_each(std::execution::par, inlier_obs_indices.begin(), inlier_obs_indices.end(),
237                       [this, &dataset, &kept_count](int obs_idx)
238         {
239           bool is_valid = processSingleObservation(dataset, obs_idx);
240           if (is_valid)
241           {
242             kept_count.fetch_add(1);
243           }
244         });
245
```

- Only process inliers. Reject points that were detected as outliers by an earlier filter.
- std::for_each with parallel execution is used to speed up the filtering

# Pipeline - Spatial Filtering

```
307     // Query R-tree for k-nearest neighbors
308     std::vector<Value> neighbors;
309     rtree_.query(boost::geometry::index::nearest(query_pt, config_.k_neighbors + 1),
310                  std::back_inserter(neighbors));
311
312     // Filter out the query point itself and calculate distances
313     auto distances = neighbors
314       | std::views::filter([point_idx = obs.point_index](const auto& neighbor) {
315           return static_cast<int>(neighbor.second) != point_idx;
316         })
317       | std::views::transform([&query_pt](const auto& neighbor) {
318           return boost::geometry::distance(query_pt, neighbor.first);
319         })
320       | std::ranges::to<std::vector>();
321
322     // Calculate mean and standard deviation
323     auto [distance_sum, squared_distance_sum, num_distances] = std::ranges::fold_left(
324       distances,
325       std::tuple{0.0, 0.0, 0u},
326       [](auto previous_result, double distance_value)
327       {
328         auto& [total_distance_sum, total_squared_distance_sum, distance_count] = previous_result;
329         total_distance_sum              += distance_value;
330         total_squared_distance_sum      += distance_value * distance_value;
331         ++distance_count;
332         return previous_result;
333       }
334     );
335
336     double mean = distance_sum / num_distances;
337     double variance = (squared_distance_sum - num_distances * mean * mean) / num_distances;
338     double stddev = std::sqrt(variance);
339
340     // Check if the nearest neighbor distance is reasonable (not an outlier)
341     double nearest_distance = std::ranges::min(distances);
342     bool is_valid = stddev > 0.0 &&
343                     std::abs(nearest_distance - mean) <= config_.std_dev_multiplier * stddev;
344
```

- Query *k* neighbors nearest to the query point and find its distance from them.
- Fold::left is used to calculate sum of distances, squared sum and count.
- Calculate mean and variance using those values.
- Outlier check is a variant of z-score. Checks if the nearest distance to a neighbour is within a multiple of the standard deviation.

# Pipeline - Filtering



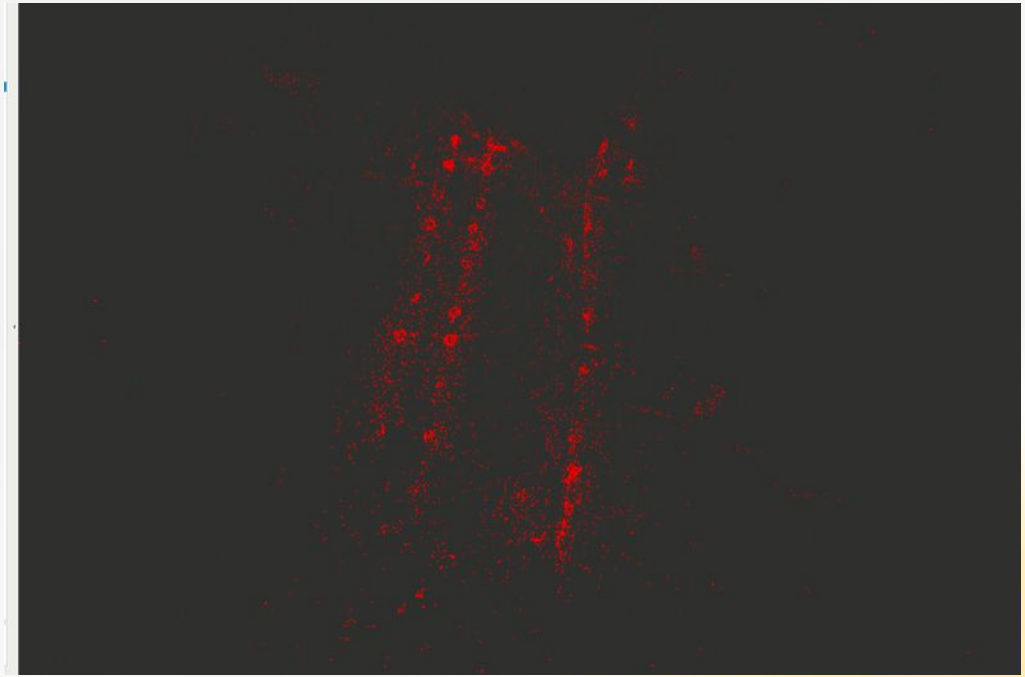
**Ladybug dataset**

**Thresholds**

- Reprojection error threshold : 25.0pix
- Neighbours for spatial filter : 100
- std_dev multiplier : 4.5

**Outliers**

- Count=22866/678155 (3.371796%),
- ReprojError:
  - Mean=21.980313px,
  - StdDev=36.999753px,
  - Median=19.146612px



# Pipeline - Filtering

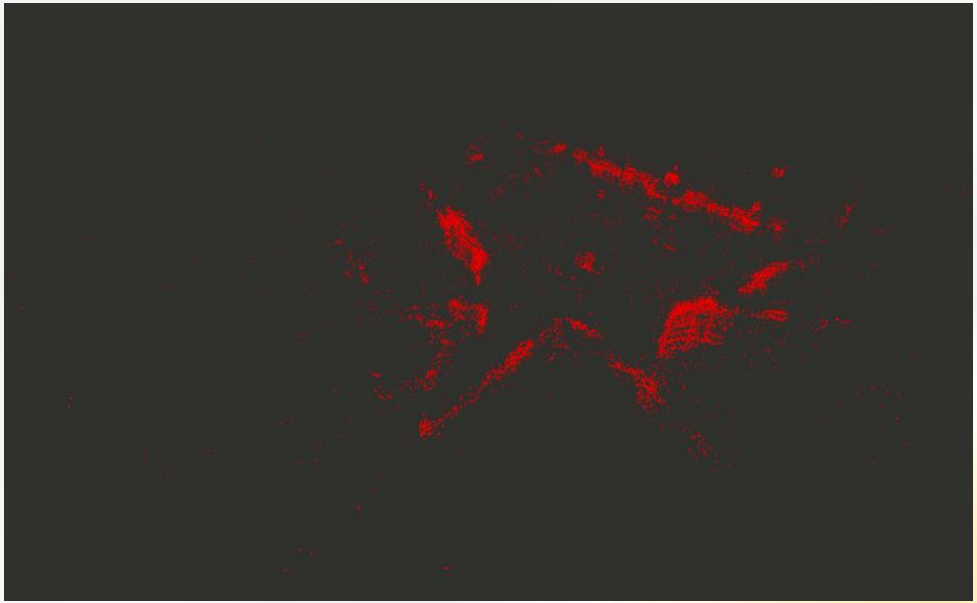
**Trafalgar Square dataset**

**Thresholds**

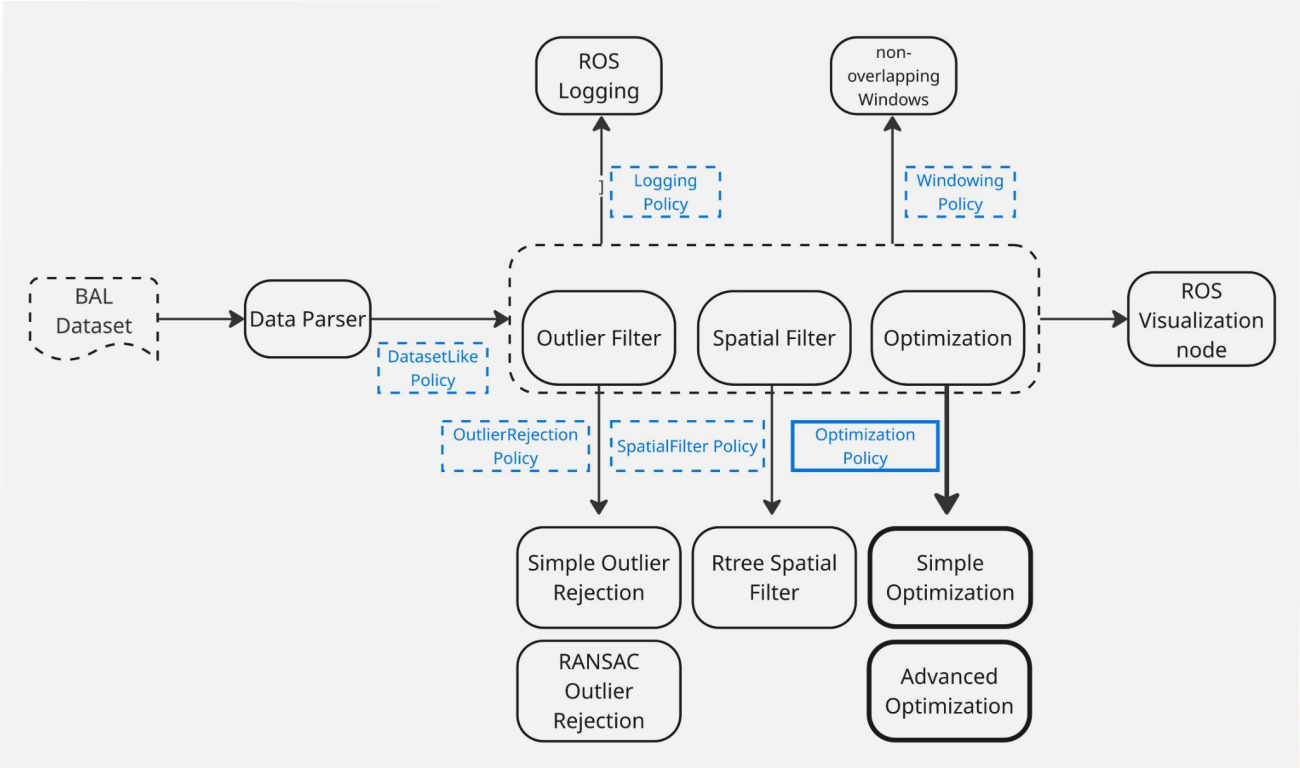
- Reprojection error threshold : 25.0pix
- Neighbours for spatial filter : 100
- std_dev multiplier : 4.5

**Outliers**

- Count=20114/225267 (8.928960%),
- ReprojError:
  - Mean=37.676941px,
  - StdDev=19.108898px,
  - Median=34.209722px



# Pipeline - Optimization



# Optimization policy


```cpp
struct IterationSummary
{
  double cost{0.0};
  double cost_change{0.0};
  double cost_reduction_percentage{0.0};
  bool step_is_successful{false};
};

  bool success{false};
  double initial_cost{0.0};
  double final_cost{0.0};
  double cost_reduction_percentage{0.0};

  std::vector<IterationSummary> iteration_summaries;
  std::string solver_summary;
};


template<typename Policy, typename Dataset>
concept OptimizationPolicy = requires(
    Policy policy,
    Dataset& dataset,
    std::span<const int> camera_window)
{
  requires DatasetLike<Dataset>;
  { policy.optimize(dataset, camera_window) }-> std::same_as<OptimizationResult>;

  { policy.getLastStatistics() } -> std::convertible_to<std::string>;
  { policy.reset() } -> std::same_as<void>;

  requires std::destructible<Policy>;
};
```

# Pipeline - Optimization



**Problem:** Refine camera poses and 3D points from noisy observations.

**Solution:** Minimize some cost functions that describe the reprojection error of the features

**Given** (Input Data)

- Observations: 2D points ($u_{ij}$, $v_{ij}$) where camera *i* sees 3D point *j*.
- Initial estimates: Camera parameters - ( R, T, f, $k_1$, $k_2$) and 3d point (x, y, z).

**Optimize**

- 9 Camera parameters - Extrinsics(R, t) and intrinsics(f, $k_1$, $k_2$).
- 3 Landmark parameters - Pose (x, y, z)

**Optimization Problem** $\min_{\{c_i\},\{X_j\}} \sum_{(i,j)\in\mathcal{O}} \rho\left(\| p_{ij} - \pi(c_i, X_j) \|^2\right)$

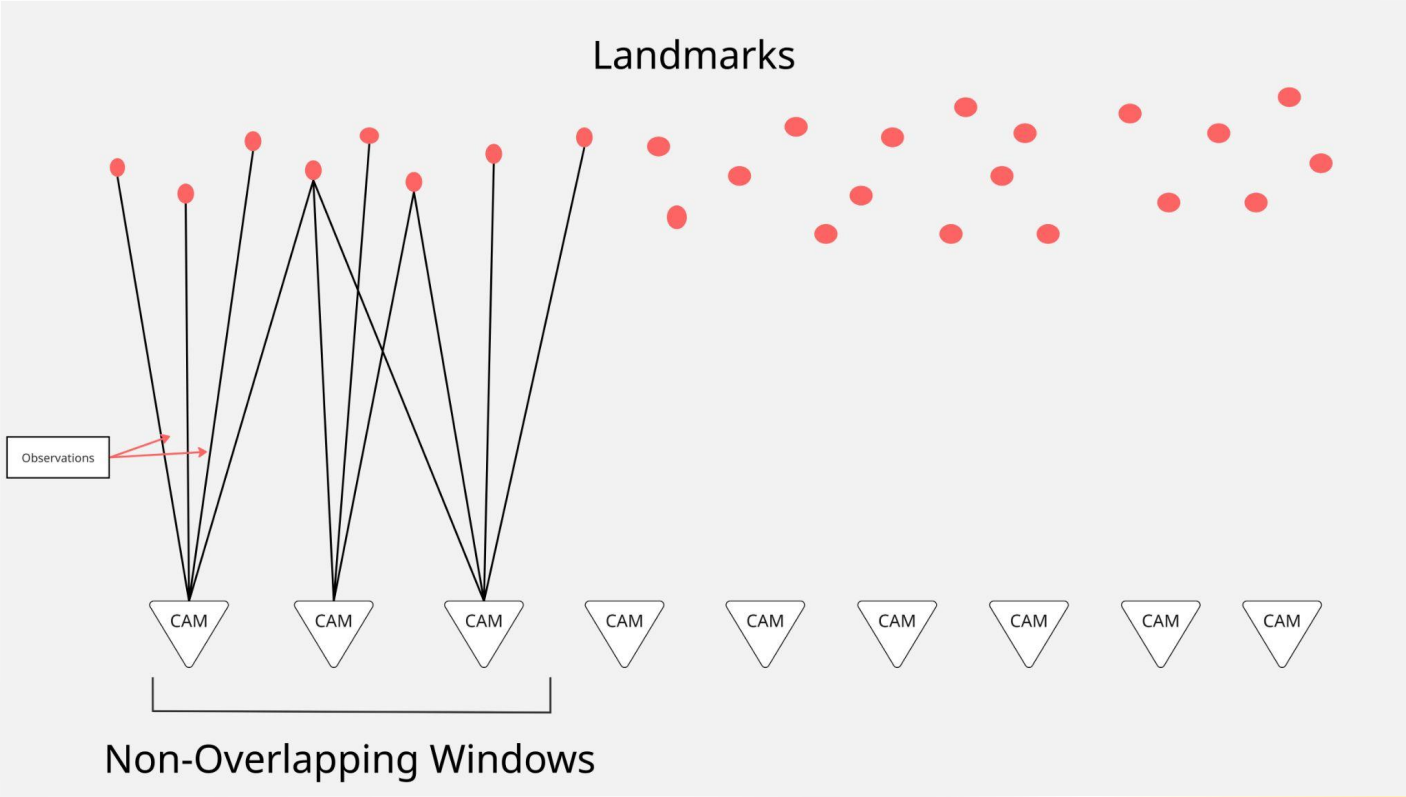
- For each observation, solve
- This equation can be expressed as $min||r(x)||^2$

# Pipeline - Optimization



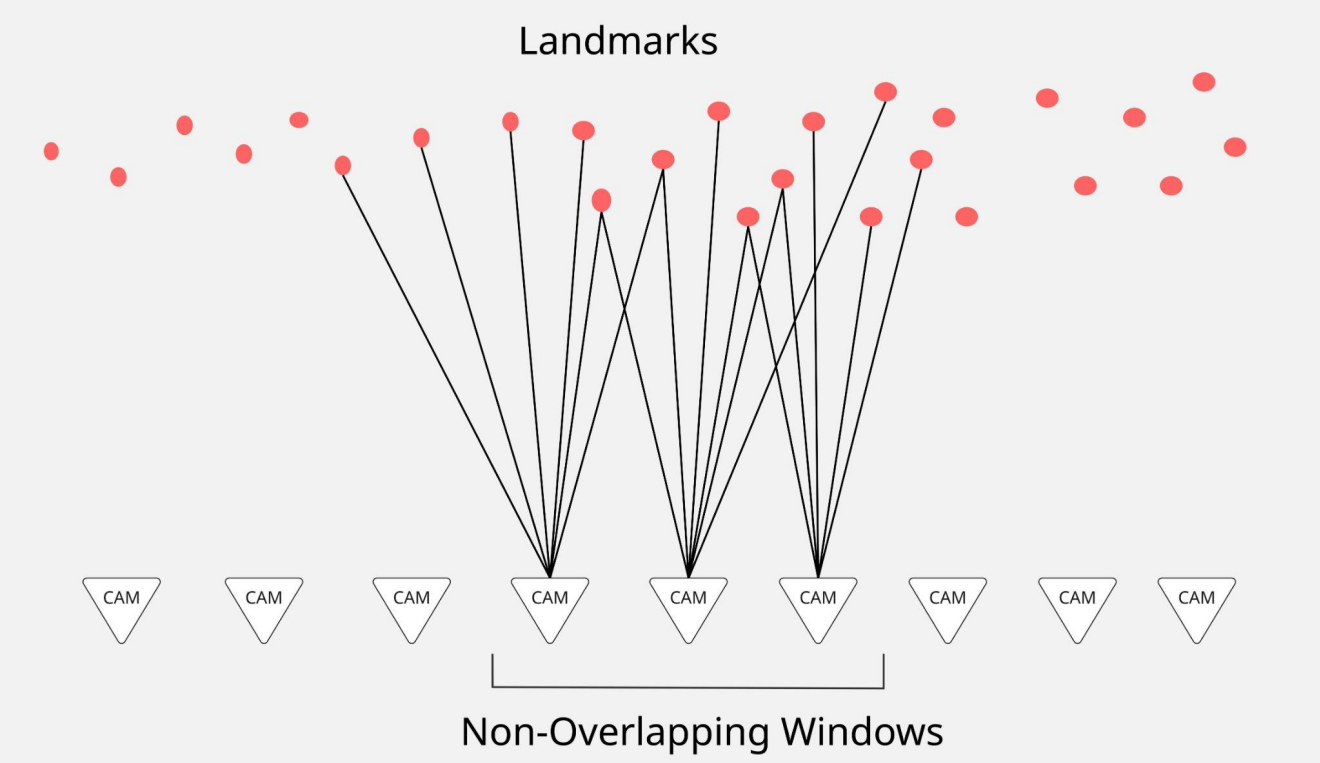
- **Linearization**
  - At the current estimate, compute Jacobians of residuals wrt parameters.
    - Using a first-order Taylor expansion, we can rephrase the problem as $f(x + \delta) \sim f(x) + J\delta$
    - So the reprojection error becomes this Normal Equation :
      - $min||r(x)||^2 \rightarrow min||r(x + \delta)||^2 \rightarrow min||f(x) + J\delta)||^2 \rightarrow (J^TJ)\delta = -J^Tr$
- **Solve Linear Subproblem**
  - Use **Schur complement** for a BA problem. Exploit sparsity in the jacobian.
    - Schur complement eliminates point variables, turning the large system into a smaller one in camera parameters.
    - Point variables are solved for by back-substituting the solved camera poses.
- **Trust-Region Strategy**
  - Use **Levenberg–Marquardt** or **Dogleg** to decide step size.
    - Prevents overshooting of estimates in cases of poorly conditioned or highly nonlinear problems.
    - Balances speed of Gauss–Newton with stability of gradient descent in flat or narrow directions.
- **Update Parameters and re-iterate**

# Pipeline - Optimization

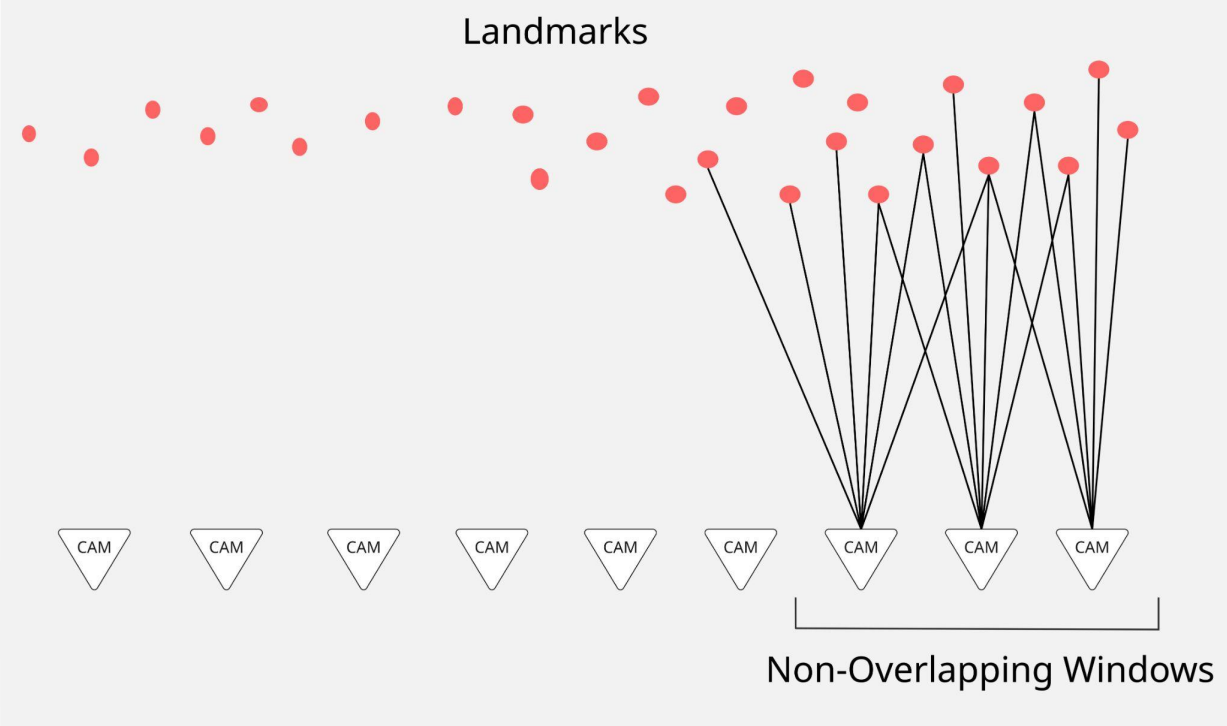


# Pipeline - Optimization





# Pipeline - Optimization



# Pipeline - Optimization



## Optimizer settings

```
if (config_.fix_first_camera && storage_info.cam_count > 0
    && !camera_params_storage_.empty())
{
  problem.SetParameterBlockConstant(camera_params_storage_[0].data());
}

ceres::Solver::Options options;
options.minimizer_type = ceres::TRUST_REGION;
options.trust_region_strategy_type = ceres::LEVENBERG_MARQUARDT;
options.linear_solver_type = ceres::SPARSE_SCHUR;
options.preconditioner_type = ceres::SCHUR_JACOBI;
options.num_threads = config_.num_threads;
options.minimizer_progress_to_stdout = config_.minimizer_progress_to_stdout;
options.max_num_iterations = config_.max_num_iterations;
options.function_tolerance = config_.function_tolerance;
options.gradient_tolerance = config_.gradient_tolerance;
options.parameter_tolerance = config_.parameter_tolerance;
options.logging_type = ceres::PER_MINIMIZER_ITERATION;
```

# Pipeline - Optimization

```cpp
auto cameras_with_observations = camera_window_span
  | std::views::filter([&](int camera_index) {return camera_index >= 0 && camera_index < static_cast<int>(in_dataset.cameras().size());})
  | std::views::transform([&](int camera_index) {return in_dataset.cameras()[camera_index].id;})
  | std::views::filter([&](int camera_id) {
      auto obs_it = camera_to_observations.find(camera_id);
      return obs_it != camera_to_observations.end() && !obs_it->second.empty();});

for (int camera_id : cameras_with_observations)
{
  // Find camera in input dataset by ID with bounds checking
  auto cam_it = std::ranges::find_if(in_dataset.cameras(),
      [camera_id](const auto& cam) { return cam.id == camera_id; });
  if (cam_it == in_dataset.cameras().end()) {continue;}

  // Extract camera parameters
  const auto& cam = *cam_it;
  auto q = cam.rotation().unit_quaternion();
  auto t = cam.translation();
  auto k = cam.intrinsics.as_vec3();

  // Create camera parameter vector
  std::vector<double> cam_params =
  {
    static_cast<double>(q.x()),
    static_cast<double>(q.y()),
    static_cast<double>(q.z()),
    static_cast<double>(q.w()),
    static_cast<double>(t.x()),
    static_cast<double>(t.y()),
    static_cast<double>(t.z()),
    static_cast<double>(k[0]),
    static_cast<double>(k[1]),
    static_cast<double>(k[2])
  };

  // Validate parameters are finite
  bool valid = std::ranges::all_of(cam_params, [](double p) { return std::isfinite(p); });
  if (!valid) continue;

  // Store vector and get pointer
  camera_id_to_storage_idx[camera_id] = camera_params_storage_.size();
  camera_params_storage_.push_back(std::move(cam_params));
  double* cam_ptr = camera_params_storage_.back().data();
  problem.AddParameterBlock(cam_ptr, 10);
}
```

- Filter and transform used to reject invalid camera indices and extract camera ids
- Camera broken down into 10 parameters - extrinsic + intrinsic
- Only add valid(non Nan) blocks
- Use quaternion parameterization for rotation

# Pipeline - Optimization

```
// Process each observation for this camera
auto valid_observations = obs_indices
  | std::views::filter([&](int idx) {return idx >= 0 && idx < static_cast<int>(observations.size());})
  | std::views::filter([&](int idx) {const auto& obs = observations[idx];
      if (obs.outlier) {
          outlier_filtered++;
          return false;
      }
      return true;
  })
  | std::views::filter([&](int idx) {return observations[idx].camera_index == camera_id;})
  | std::views::filter([&](int idx) {const auto& obs = observations[idx];
      return obs.point_index >= 0 && obs.point_index < static_cast<int>(points.size());});

size_t camera_obs_count = 0;
for (int obs_idx : valid_observations)
{
  total_observations_processed++;
  const auto& obs = observations[obs_idx];

  // Create or reuse point parameter block
  size_t pt_storage_idx;
  auto pit = point_index_to_storage_idx.find(obs.point_index);
  if (pit == point_index_to_storage_idx.end())
  {
    // Create new point parameter block
    const auto& p = points[obs.point_index];
    // Validate point coordinates
    if (!std::ranges::all_of(p, [](double coord) { return std::isfinite(coord); })) {continue;}

      // Store vector and get pointer
      pt_storage_idx = point_params_storage_.size();
      point_index_to_storage_idx[obs.point_index] = pt_storage_idx;
      point_params_storage_.push_back({
        static_cast<double>(p[0]),
        static_cast<double>(p[1]),
        static_cast<double>(p[2])});

      double* pt_ptr = point_params_storage_.back().data();
      problem.AddParameterBlock(pt_ptr, 3);
    } else
    {
      pt_storage_idx = pit->second;
    }
```

- Select only valid observations using views::filter
- Add the 3 dimensions of the point as a parameter block.

# Pipeline - Optimization

```cpp
for (const auto& [camera_id, storage_idx] : valid_camera_updates)
{
  auto cam_it = std::ranges::find_if(dataset.cameras(), [camera_id](const auto& cam) { return cam.id == camera_id; });
  if (cam_it == dataset.cameras().end()) continue;

  std::span<const double, 10> params{camera_params_storage_[storage_idx]};
  Eigen::Quaterniond q(params[3], params[0], params[1], params[2]);
  q.normalize();
  Eigen::Vector3d t(params[4], params[5], params[6]);
    cam_it->T_c_w = typename Dataset::Traits::SE3(typename Dataset::Traits::SO3(q), t);

  // Update intrinsics
  cam_it->intrinsics.focal_length = params[7];
  cam_it->intrinsics.k1 = params[8];
  cam_it->intrinsics.k2 = params[9];
}

auto valid_point_updates = point_index_to_storage_idx
  | std::views::filter([&](const auto& pair) {
      const auto& [point_index, storage_idx] = pair;
      return storage_idx < point_params_storage_.size() &&
             point_index >= 0 &&
             point_index < static_cast<int>(dataset.points().size());
  });

for (const auto& [point_index, storage_idx] : valid_point_updates)
{
  std::span<const double, 3> point_params{point_params_storage_[storage_idx]};
  dataset.points()[point_index] = typename Dataset::Traits::Vec3(
    point_params[0], point_params[1], point_params[2]);
}
}
```

- For each camera in the window, find its optimized block and update its values.
- For each point that was updated, access the value and update it.

# Pipeline - Results



**Optimization results in balanced mode**

| **Dataset** | Avg Repro Error before | Avg Repro Error after |
|---|---|---|
| Ladybug | 3.859 pixels | 202.43 pixels |
| Trafalgar Square | 8.309 pixels | 294.173 pixels |

# Pipeline - Optimization



```
// cameras out of the current window become constant
auto prev_window_cameras = prev_window_cameras_ | std::views::filter([&](int cam_id){ return !window_cameras.contains(cam_id); });
std::ranges::for_each(prev_window_cameras, [&](int cam_id)
{
  if (auto it = camera_id_to_index_.find(cam_id); it != camera_id_to_index_.end())
  {
    auto& camera = cameras_[it->second];
    problem_->SetParameterBlockConstant(camera.data());
    camera.is_variable = false;
  }
});

// Entering cameras become variable
auto current_window_cameras = window_cameras | std::views::filter([&](int cam_id){ return !prev_window_cameras_.contains(cam_id); });
std::ranges::for_each(current_window_cameras, [&](int cam_id)
{
  if (auto it = camera_id_to_index_.find(cam_id); it != camera_id_to_index_.end())
  {
    auto& camera = cameras_[it->second];
    problem_->SetParameterBlockVariable(camera.data());
    camera.is_variable = true;
  }
});
```

```
    // camera.parameters = [qx,qy,qz,qw, tx,ty,tz, fx, k1,k2]
    for (int i = 4; i <= 6; ++i)
    {
      problem_->SetParameterLowerBound(camera.data(), i, config_.cam_translation_min);
      problem_->SetParameterUpperBound(camera.data(), i, config_.cam_translation_max);
    }
    problem_->SetParameterLowerBound(camera.data(), 7, config_.focal_min);
    problem_->SetParameterUpperBound(camera.data(), 7, config_.focal_max);
    problem_->SetParameterLowerBound(camera.data(), 8, config_.k_min);
    problem_->SetParameterUpperBound(camera.data(), 8, config_.k_max);
    problem_->SetParameterLowerBound(camera.data(), 9, config_.k_min);
    problem_->SetParameterUpperBound(camera.data(), 9, config_.k_max);
  }
}
```

- Add all cameras to the optimization problem
- Only set cameras that in the window as variable, set all other cameras as constants.
- User parameterLowerBound and parameterUpperBound to restrict the optimization space.

# Pipeline - Optimization

```cpp
// Adjust in-window support counts based on entering/leaving cameras
std::unordered_set<int> visited_points;  // points whose support may have changed
std::ranges::for_each(curr_camera, [&](int cam_id)
{
  auto cam_obs_it = camera_id_to_obs_indices_.find(cam_id);
  std::ranges::for_each(cam_obs_it->second, [&](int obs_idx)
  {
    const auto& obs = observations_[static_cast<size_t>(obs_idx)];
    if (auto pt_it = point_id_to_index_.find(obs.point_id); pt_it != point_id_to_index_.end())
    {
      points_[pt_it->second].in_window_support++;
      visited_points.insert(obs.point_id);
    }
  });
});
std::ranges::for_each(prev_camera, [&](int cam_id)
{
  auto cam_obs_it = camera_id_to_obs_indices_.find(cam_id);
  std::ranges::for_each(cam_obs_it->second, [&](int obs_idx)
  {
    const auto& obs = observations_[static_cast<size_t>(obs_idx)];
    if (auto pt_it = point_id_to_index_.find(obs.point_id); pt_it != point_id_to_index_.end())
    {
      points_[pt_it->second].in_window_support--;
      visited_points.insert(obs.point_id);
    }
  });
});
```

- Keep points observed by any in-window cameras, remove those with zero in-window support
- Freeze points with exactly 1 in-window camera. Optimize those with 2+ in-window cameras.
- Add residuals for ALL observations of kept points, including cross-window observations

# Pipeline - Optimization

```
if (!point.is_in_problem)
{
  problem_->AddParameterBlock(point.data(), 3);
  point.is_in_problem = true;
  if (config_.use_parameter_bounds) {
    for (int i = 0; i < 3; ++i)
    {
        problem_->SetParameterLowerBound(point.data(), i, config_.point_min);
        problem_->SetParameterUpperBound(point.data(), i, config_.point_max);
    }
  }
}

if (point.in_window_support == 1 && config_.freeze_weak_points)
{
  problem_->SetParameterBlockConstant(point.data());
  point.is_variable = false;
} else
{
  problem_->SetParameterBlockVariable(point.data());
  point.is_variable = true;
}
```

- Keep points observed by any in-window cameras, remove those with zero in-window support
- Freeze points with exactly 1 in-window camera. Optimize those with 2+ in-window cameras.
- Add residuals for ALL observations of kept points, including cross-window observations

# Pipeline - Optimization



```
if (!config_.use_schur_ordering) return;

parameter_ordering_ = std::make_shared<ceres::ParameterBlockOrdering>();

// Group 0: Points (to be eliminated via Schur complement)
for (auto& point : points_)
{
  if (point.is_variable && point.is_in_problem)
  {
    parameter_ordering_->AddElementToGroup(point.data(), 0);
  }
}

// Group 1: Cameras (retained in the reduced system)
for (auto& camera : cameras_)
{
  if (camera.is_variable && camera.is_in_problem)
  {
    parameter_ordering_->AddElementToGroup(camera.data(), 1);
  }
}
solver_options_.linear_solver_ordering = parameter_ordering_;
```

Parameter ordering

- For schur complement, ceres wants to eliminate the points block and solve for the camera poses.
- Groups parameters into two categories to explicitly tag the blocks associated with cameras and points.
- Elements added to group 0 are eliminated first while group 1 is retained.
- Ceres does automatically group terms but this guarantees desired performance .

# Pipeline - Optimization

```cpp
if (!config_.use_inner_iterations) return;

inner_iteration_ordering_ = std::make_shared<ceres::ParameterBlockOrdering>();

// Group 0: Points (optimize first in inner iterations)
// This allows points to adjust while cameras are held fixed
for (auto& point : points_)
{
  if (point.is_variable && problem_->HasParameterBlock(point.data()))
  {
    inner_iteration_ordering_->AddElementToGroup(point.data(), 0);
  }
}

// Group 1: Cameras (optimize second in inner iterations)
// This allows cameras to adjust while points are held fixed
for (auto& camera : cameras_)
{
  if (camera.is_variable && problem_->HasParameterBlock(camera.data()))
  {
    inner_iteration_ordering_->AddElementToGroup(camera.data(), 1);
  }
}
solver_options_.inner_iteration_ordering = inner_iteration_ordering_;
```

Inner iterations

- Instead of solving the whole problem in one go, inner iterations solves it one group at a time. Decouples the problem.
- This solve is done in addition to the main solve.
- Group 0 - Fix all cameras, optimize only points
- Group 1 - Fix all points, optimize only the cameras
- Each subproblem is simpler and more focused
- Drawback : Adds more iterations and not guaranteed to give u better solution than joint optimization

# Pipeline - Optimization



```
10  class AdaptiveHuberLoss : public ceres::LossFunction {
11  public:
12      explicit AdaptiveHuberLoss(double initial_delta = 1.0)
13          : delta_(initial_delta)
14      {}
15
16      void Evaluate(double s, double rho[3]) const override
17      {}
18
```

Loss functions

- Cauchy and huber loss functions
- Huber : Quadratic loss till threshold and liner beyond.
- Cauchy : Quadratic loss till threshold and log beyond.
- Cauchy is more tolerant to large outlier but can slow down optimization if not tuned.
- Custom loss function usually an overkill.

# Pipeline - Results

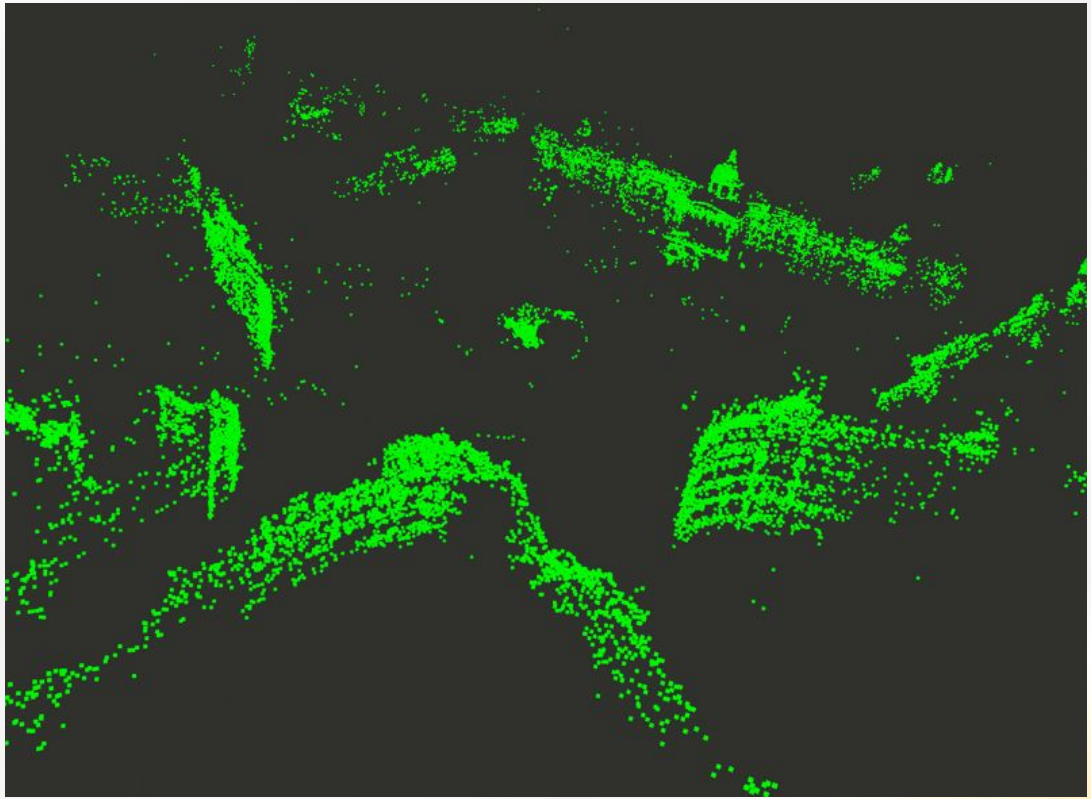


**Optimization results**

| **Dataset** | Avg Repro Error before | Avg Repro Error after |
|---|---|---|
| Ladybug | 3.859 pixels | 1.060 pixels |
| Trafalgar Square | 8.309 pixels | 0.748 pixels |

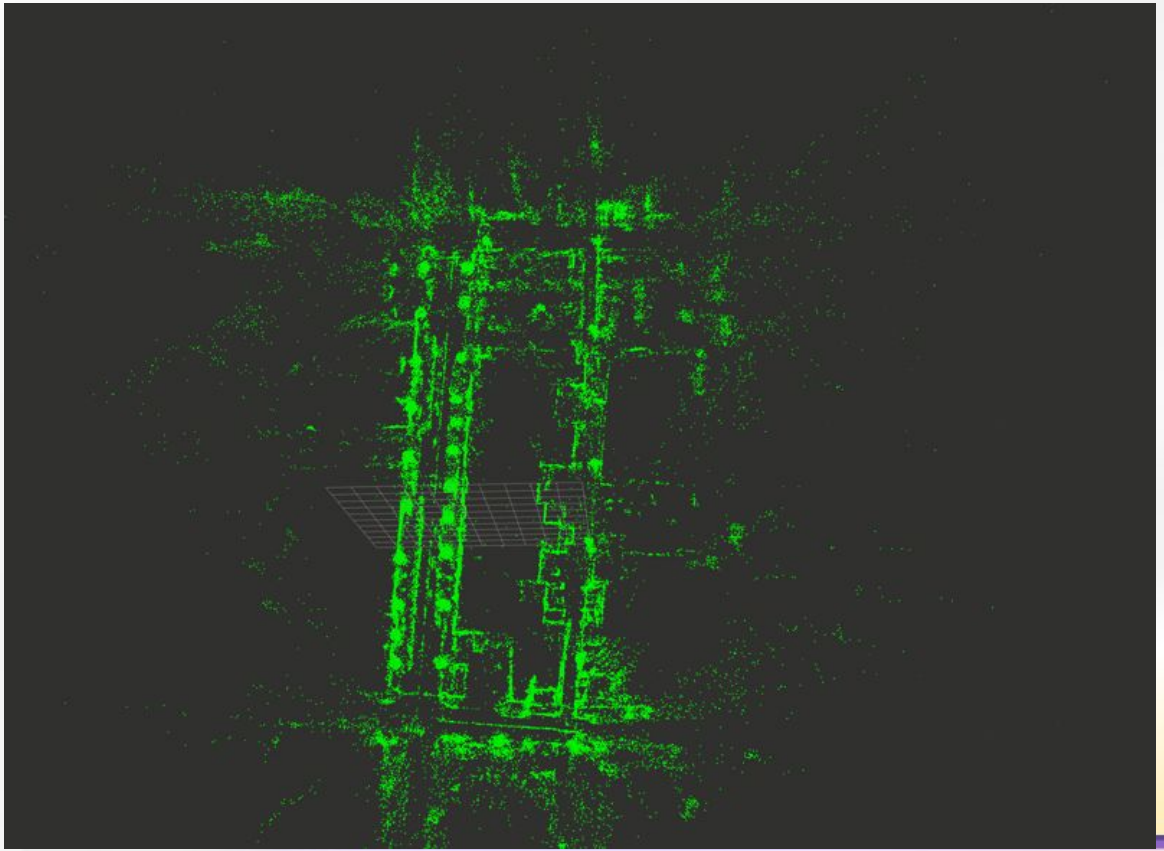
# Pipeline - Results





# Pipeline - Results





# References


Back to basics - concepts by Nicolai Josuttis (link)

Implementing particle filter with C++ ranges - Nahuel Espinosa (link)

Linear algebra and optimization Seminar - Sameer Agarwal(link)

Ceres solver - documentation(link)


# Pipeline - Questions