Surfel-LIO

Implementation Details

A Technical Companion Document

Seungwon Choi[1], Dong-Gyu Park[2], Seo-Yeon Hwang[2], Tae-Wan Kim[1]

[1]Seoul National University [2]Clobot Co., Ltd.

https://github.com/93won/lidar_inertial_odometry

December 3, 2025

Abstract

This document provides detailed implementation information for Surfel-LIO, a LiDAR-inertial odometry system that achieves O(1) correspondence retrieval through pre-computed surfels and hierarchical voxel hashing. We present a comprehensive mapping between the theoretical formulations in the paper and their corresponding code implementations, including data structures, algorithms, and configuration parameters. This companion document is intended to facilitate reproducibility and help researchers understand, modify, and extend our system.

Contents

1 Introduction

1.1 Purpose of This Document

This document serves as a technical companion to the paper *"Surfel-LIO: Fast LiDAR-Inertial Odometry with Pre-computed Surfels and Hierarchical Z-order Voxel Hashing."*. While the paper presents the theoretical foundations and experimental results, this document bridges the gap between theory and implementation by providing:

- **Theory-to-Code Mapping**: Direct correspondence between equations in the paper and their C++ implementations
- **Data Structure Details**: Memory layouts, design decisions, and practical considerations
- **Algorithm Walkthroughs**: Step-by-step explanations with code snippets
- **Configuration Guide**: Parameter descriptions and tuning recommendations
- **Build Instructions**: Dependencies, compilation, and execution steps

1.2 System Overview

Surfel-LIO is a tightly-coupled LiDAR-inertial odometry system designed for real-time state estimation. The key innovation lies in the map representation: instead of performing nearest neighbor search and plane fitting at query time, we pre-compute surfel parameters (centroid, normal, planarity) during map updates. This shifts computational burden from the time-critical IEKF iteration phase to the less frequent map update phase.

Figure 1 illustrates the overall system architecture. The three main components are:

1. **IMU Propagation**: High-frequency state prediction using inertial measurements
2. **LiDAR Update**: Point-to-plane registration with pre-computed surfels via IEKF
3. **Map Management**: Hierarchical voxel structure with lazy surfel updates

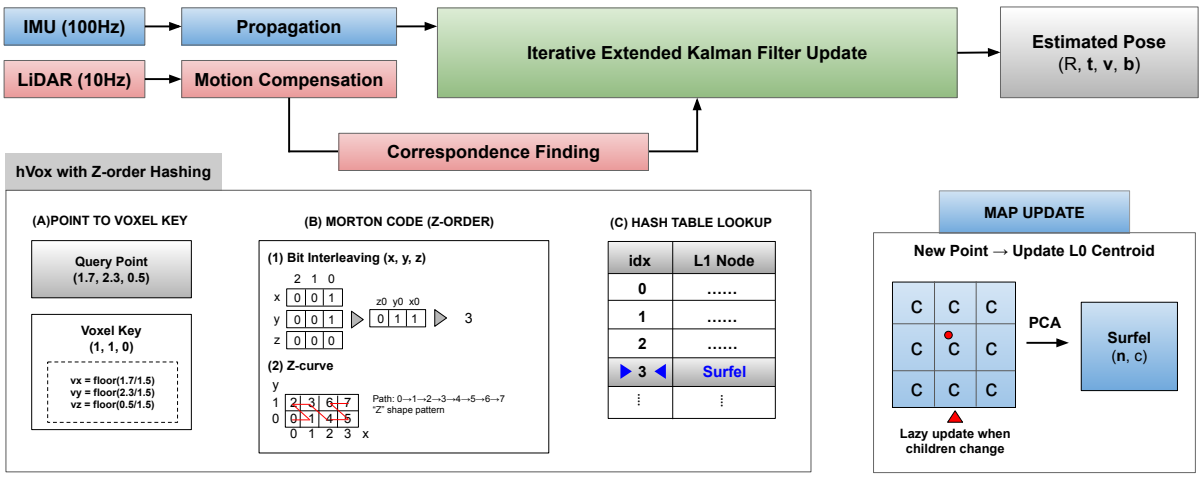

Figure 1: System architecture of Surfel-LIO. Top: the IEKF-based estimation pipeline. Bottom-left: O(1) correspondence finding via Z-order hashing. Bottom-right: incremental map update with lazy surfel recomputation.

1.3 Code Repository Structure

The codebase is organized as follows:

```
lidar_inertial_odometry/
|-- app/
|   +-- lio_player.cpp          # Main executable
|-- src/
|   |-- core/
|   |   |-- Estimator.cpp/h     # IEKF state estimation
|   |   |-- State.cpp/h         # State representation (SO3, position, etc.)
|   |   |-- VoxelMap.cpp/h      # hVox: hierarchical voxel map
|   |   +-- ProbabilisticKernelOptimizer.cpp/h
|   |-- util/
|   |   |-- ConfigUtils.cpp/h   # YAML configuration parser
|   |   |-- LieUtils.cpp/h      # SO3/SE3 operations
|   |   +-- PointCloudUtils.cpp/h
|   +-- viewer/
|       +-- LIOViewer.cpp/h     # Visualization
|-- config/
|   |-- avia.yaml               # Livox AVIA configuration
|   +-- mid360.yaml             # Livox Mid-360 configuration
|-- thirdparty/
|   |-- pangolin/               # Visualization library
|   |-- spdlog/                 # Logging library
|   +-- unordered_dense/        # Robin Hood hash table
+-- CMakeLists.txt
```

1.4 Key Design Decisions

Several design decisions distinguish our implementation:

1. **Two-Level Hierarchy (L0/L1)**: L0 voxels store point centroids; L1 voxels store surfels computed from L0 children. This separation allows fine-grained point aggregation while providing robust plane estimation from sufficient samples.

2. **Morton Code Hashing**: Z-order curve encoding preserves spatial locality, improving cache efficiency when processing spatially coherent LiDAR scans.

3. **Robin Hood Hashing**: We use `ankerl::unordered_dense` for hash table implementation, which provides better cache locality than standard `std::unordered_map` through open addressing with Robin Hood collision resolution.

4. **Lazy Surfel Updates**: Surfels are marked "dirty" when underlying L0 voxels change, but recomputation is deferred until the surfel is actually queried. This avoids unnecessary PCA computations for voxels that are never accessed.

5. **Incremental Centroid Updates**: L0 centroids are updated incrementally using Welford's algorithm, requiring only O(1) operations per point insertion regardless of accumulated point count.

1.5 Document Organization

The remainder of this document is organized as follows:

- **Section 2**: Data structures for L0 voxels, L1 surfels, and hash tables
- **Section 3**: Z-order spatial hashing and Morton code implementation
- **Section 4**: Map update and correspondence finding algorithms
- **Section 5**: IEKF formulation and implementation
- **Section 6**: Configuration parameters and tuning guide

Each section presents the relevant equations from the paper alongside the corresponding code implementation, with explanatory notes on practical considerations and design choices.

2 Data Structures

This section describes the core data structures used in Surfel-LIO. Figure 2 illustrates the hierarchical relationship between L0 and L1 voxels.

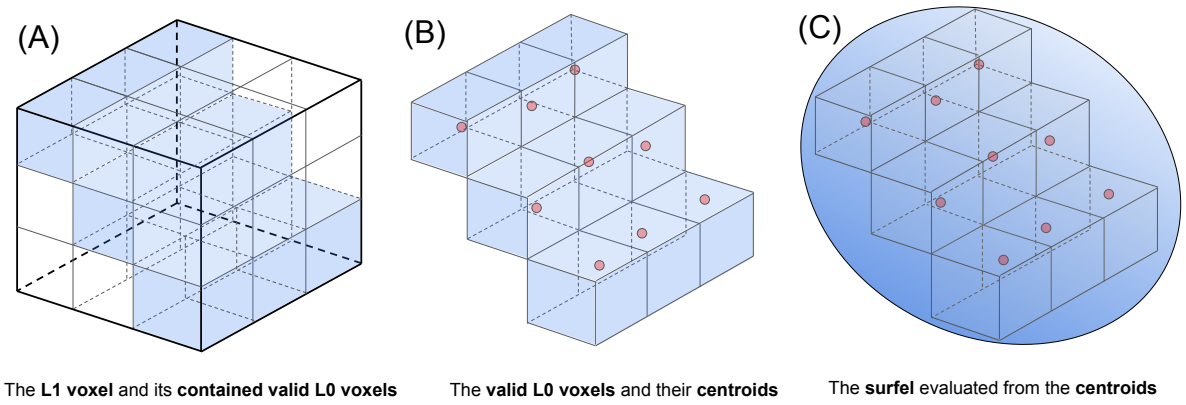

Figure 2: Hierarchical voxel structure and surfel representation. (A) L1 voxel containing $3\times3\times3$ L0 children. (B) L0 voxels store incrementally updated centroids. (C) L1 surfels are derived via PCA on L0 centroids.

2.1 L0 Voxel: Point Aggregation

Paper Reference: Equation (1), Figure 3(B)

L0 voxels are the finest level of our hierarchy. Each L0 voxel aggregates all points falling within its spatial extent into a single centroid, updated incrementally as new points arrive.

2.1.1 Theory

The incremental centroid update formula avoids storing raw points:

$$\mathbf{c}^{(n+1)} = \frac{n \cdot \mathbf{c}^{(n)} + \mathbf{p}_{\text{new}}}{n+1} \qquad \text{(Paper Eq. 1)}$$

where n is the accumulated point count, $\mathbf{c}^{(n)}$ is the current centroid, and $\mathbf{p}_{\text{new}}$ is the newly inserted point.

2.1.2 Implementation

Listing 1: L0 Voxel structure (src/core/VoxelMap.h)

```
struct L0Voxel {
    Eigen::Vector3d centroid;  // Running centroid
    int count;                 // Number of accumulated points

    L0Voxel() : centroid(Eigen::Vector3d::Zero()), count(0) {}

    // Incremental centroid update - O(1) per point
    void addPoint(const Eigen::Vector3d& point) {
        centroid = (count * centroid + point) / (count + 1);
```

```
        count++;
    }
};
```

2.1.3 Key Properties

- **Memory**: O(1) per voxel (fixed 32 bytes: 24 for Vector3d + 4 for count + padding)
- **Update**: O(1) per point insertion
- **No raw point storage**: Memory usage independent of point density

2.2 L1 Voxel: Surfel Representation

Paper Reference: Equations (2)–(5), Figure 3(C)

L1 voxels serve as parent nodes, each encompassing a $3 \times 3 \times 3$ grid of L0 children. Instead of storing points, each L1 voxel maintains a pre-computed surfel—a compact planar primitive.

2.2.1 Theory

Given centroids $\{\mathbf{c}_i\}_{i=1}^{m}$ from m occupied L0 children:

Mean centroid:

$$\bar{\mathbf{c}} = \frac{1}{m} \sum_{i=1}^{m} \mathbf{c}_i \qquad \text{(Paper Eq. 2)}$$

Covariance matrix:

$$\mathbf{C} = \frac{1}{m} \sum_{i=1}^{m} (\mathbf{c}_i - \bar{\mathbf{c}})(\mathbf{c}_i - \bar{\mathbf{c}})^\top \qquad \text{(Paper Eq. 3)}$$

Surfel normal (eigenvector of smallest eigenvalue):

$$\mathbf{n} = \mathbf{v}_3 \quad \text{where} \quad \lambda_1 \geq \lambda_2 \geq \lambda_3 \qquad \text{(Paper Eq. 4)}$$

Planarity score:

$$\rho = \frac{\lambda_2 - \lambda_3}{\lambda_1 + \epsilon} \qquad \text{(Paper Eq. 5)}$$

2.2.2 Implementation

Listing 2: Surfel structure (src/core/VoxelMap.h)

```
struct Surfel {
    Eigen::Vector3d normal;     // Plane normal (unit vector)
    Eigen::Vector3d centroid;   // Plane centroid
    double planarity;           // Confidence score [0, 1]
    bool valid;                 // Sufficient points for estimation?

    Surfel() : normal(Eigen::Vector3d::Zero()),
               centroid(Eigen::Vector3d::Zero()),
               planarity(0.0), valid(false) {}
};

struct L1Voxel {
    Surfel surfel;
    std::vector<uint64_t> children;  // Morton codes of L0 children
    bool dirty;                       // Needs recomputation?
```

```
    L1Voxel() : dirty(true) {}

    // Compute surfel from L0 children centroids
    void computeSurfel(const std::vector<Eigen::Vector3d>& centroids) {
        int m = centroids.size();
        if (m < 3) {  // Need at least 3 points for plane
            surfel.valid = false;
            return;
        }

        // Compute mean centroid (Eq. 2)
        Eigen::Vector3d mean = Eigen::Vector3d::Zero();
        for (const auto& c : centroids) {
            mean += c;
        }
        mean /= m;

        // Compute covariance matrix (Eq. 3)
        Eigen::Matrix3d cov = Eigen::Matrix3d::Zero();
        for (const auto& c : centroids) {
            Eigen::Vector3d diff = c - mean;
            cov += diff * diff.transpose();
        }
        cov /= m;

        // Eigendecomposition
        Eigen::SelfAdjointEigenSolver<Eigen::Matrix3d> solver(cov);
        Eigen::Vector3d eigenvalues = solver.eigenvalues();   // Ascending
            order
        Eigen::Matrix3d eigenvectors = solver.eigenvectors();

        // Normal is eigenvector of smallest eigenvalue (Eq. 4)
        surfel.normal = eigenvectors.col(0);  // First column = smallest
        surfel.centroid = mean;

        // Planarity score (Eq. 5)
        double lambda1 = eigenvalues(2);  // Largest
        double lambda2 = eigenvalues(1);  // Middle
        double lambda3 = eigenvalues(0);  // Smallest
        constexpr double epsilon = 1e-6;
        surfel.planarity = (lambda2 - lambda3) / (lambda1 + epsilon);

        surfel.valid = true;
        dirty = false;
    }
};
```

2.2.3 Planarity Interpretation

The planarity score ρ indicates how well the L0 centroids fit a plane:

- $\rho \approx 1$: Points lie on a plane (ideal for registration)
- $\rho \approx 0$: Points are collinear or isotropic (unreliable normal)

Surfels with $\rho < \rho_{\text{min}}$ are rejected during correspondence search.

2.3 Hash Table: Robin Hood Hashing

Paper Reference: Section III-C

Both L0 and L1 voxels are stored in hash tables for O(1) average-case access. We use `ankerl::unordered_dense`, a Robin Hood hashing implementation.

2.3.1 Why Robin Hood Hashing?

Standard hash tables (e.g., `std::unordered_map`) use separate chaining, where collisions are resolved by linked lists. This causes:

- Poor cache locality (pointer chasing)
- Memory fragmentation
- Unpredictable access times

Robin Hood hashing uses open addressing with a key insight: when inserting, if the new element has traveled farther from its ideal position than the current occupant, they swap places. This:

- Minimizes variance in probe sequence lengths
- Keeps data contiguous in memory
- Improves cache efficiency

2.3.2 Implementation

Listing 3: Hash table declarations (src/core/VoxelMap.h)

```
#include "thirdparty/unordered_dense/unordered_dense.h"

class VoxelMap {
private:
    // Morton code -> L0 voxel
    ankerl::unordered_dense::map<uint64_t, L0Voxel> l0_map_;

    // Morton code -> L1 voxel
    ankerl::unordered_dense::map<uint64_t, L1Voxel> l1_map_;

    double voxel_size_;         // L0 voxel edge length (s0)
    double l1_voxel_size_;      // L1 voxel edge length (s1 = 3 * s0)
    double planarity_thresh_;  // Minimum planarity for valid surfel

public:
    VoxelMap(double voxel_size, double planarity_thresh)
        : voxel_size_(voxel_size),
          l1_voxel_size_(3.0 * voxel_size),
          planarity_thresh_(planarity_thresh) {}

    // O(1) average lookup
    L0Voxel* getL0Voxel(uint64_t morton_code);
    L1Voxel* getL1Voxel(uint64_t morton_code);

    // O(1) average insertion
    void insertPoint(const Eigen::Vector3d& point);

    // O(1) correspondence retrieval
    Surfel* findCorrespondence(const Eigen::Vector3d& point);
};
```

Table 1: Hash table implementation comparison

Property	**std::unordered_map**	**ankerl::unordered_dense**
Collision resolution	Separate chaining	Open addressing (Robin Hood)
Memory layout	Scattered (pointers)	Contiguous
Cache efficiency	Poor	Good
Average lookup	O(1)	O(1)
Worst-case lookup	O(n)	O(log n) expected

2.3.3 Memory Layout Comparison

2.4 Voxel Size Relationship

The L0 and L1 voxel sizes are related by a factor of 3:

$$s_1 = 3 \cdot s_0 \tag{1}$$

This means each L1 voxel can contain up to $3^3 = 27$ L0 children. In practice, occupancy is typically 30–70% depending on scene geometry.

Listing 4: Voxel size configuration (config/avia.yaml)

```
voxel:
  size: 0.5          # L0 voxel size (s0) in meters
  # L1 voxel size is automatically 3 * 0.5 = 1.5 meters

surfel:
  min_planarity: 0.1  # Minimum planarity threshold
  min_points: 3       # Minimum L0 children for valid surfel
```

Design Rationale:

- **L0 size (0.5m)**: Small enough for fine-grained point aggregation, large enough to contain multiple points per scan
- **L1 size (1.5m)**: Large enough to accumulate sufficient L0 centroids for robust plane estimation
- **Factor of 3**: Enables efficient parent-child lookup via integer division of Morton codes

3 Z-order Spatial Hashing

This section describes how we convert 3D coordinates to hash keys using Morton codes, enabling cache-friendly spatial indexing.

3.1 Motivation: Cache Locality

Paper Reference: Section III-C, Figure 2

Standard hash functions scatter spatially adjacent voxels across memory. This causes frequent cache misses when processing LiDAR scans, where consecutive points are spatially coherent due to the sensor's scanning pattern.

The Problem:

- Modern CPUs access RAM in cache lines (typically 64 bytes)

- Loading from random memory: ∼100 cycles latency
- Loading from L1 cache: ∼4 cycles latency
- LiDAR points are spatially adjacent → their voxels should be memory-adjacent

The Solution: Z-order (Morton) encoding maps spatially adjacent 3D coordinates to numerically adjacent hash keys, improving cache hit rates.

3.2 Morton Code: Bit Interleaving

Paper Reference: Equation (6), Figure 2(B)

The Morton code interleaves the bits of 3D coordinates into a single 64-bit integer:

$$M(k_x, k_y, k_z) = \sum_{i=0}^{20} \left(x_i \cdot 4^i + y_i \cdot 2 \cdot 4^i + z_i \cdot 4 \cdot 4^i \right) \qquad \text{(Paper Eq. 6)}$$

where x_i, y_i, z_i are the i-th bits of each coordinate.

3.2.1 Visual Example

For coordinates $(x, y, z) = (1, 1, 0)$ with 3-bit representation:

```
x = 1 = 001 (binary)
y = 1 = 001 (binary)
z = 0 = 000 (binary)

Interleaving (z, y, x order per bit position):
Bit 0: z0=0, y0=1, x0=1 -> 011
Bit 1: z1=0, y1=0, x1=0 -> 000
Bit 2: z2=0, y2=0, x2=0 -> 000

Morton code = 000 000 011 = 3 (decimal)
```

3.2.2 Implementation

Listing 5: Morton code encoding (src/core/VoxelMap.cpp)

```cpp
// Spread bits: insert two 0s between each bit
// Example: 0b101 -> 0b001000001
inline uint64_t spreadBits(uint64_t v) {
    // Mask to keep only lower 21 bits
    v &= 0x1fffff;

    // Magic bit manipulation for parallel spreading
    v = (v | (v << 32)) & 0x1f00000000ffff;
    v = (v | (v << 16)) & 0x1f0000ff0000ff;
    v = (v | (v <<  8)) & 0x100f00f00f00f00f;
    v = (v | (v <<  4)) & 0x10c30c30c30c30c3;
    v = (v | (v <<  2)) & 0x1249249249249249;

    return v;
}

// Compute Morton code from 3D integer coordinates
uint64_t computeMortonCode(int x, int y, int z) {
    // Offset to handle negative coordinates
    // 21 bits support range [-2^20, 2^20] = [-1048576, 1048576]
```

```
    constexpr int OFFSET = 1 << 20;  // 2^20

    uint64_t ux = static_cast<uint64_t>(x + OFFSET);
    uint64_t uy = static_cast<uint64_t>(y + OFFSET);
    uint64_t uz = static_cast<uint64_t>(z + OFFSET);

    // Interleave: x takes bit positions 0,3,6,...
    //             y takes bit positions 1,4,7,...
    //             z takes bit positions 2,5,8,...
    return spreadBits(ux) | (spreadBits(uy) << 1) | (spreadBits(uz) << 2);
}
```

3.2.3 Bit Spreading Explained

The magic numbers perform parallel bit spreading. For a 21-bit input, we insert two zeros between each bit:

```
Input:  ...  b2  b1  b0
Output: ... b2 00 b1 00 b0

Step-by-step for input 0b101 (decimal 5):
Original:      0b00000101
After spread:  0b001000001

Magic number derivation:
0x1f00000000ffff   = mask for 32-bit split
0x1f0000ff0000ff   = mask for 16-bit split
0x100f00f00f00f00f = mask for 8-bit split
0x10c30c30c30c30c3 = mask for 4-bit split
0x1249249249249249 = mask for 2-bit split (final)
```

3.3 Coordinate Quantization

Paper Reference: Algorithm 2, Line 1

Before computing the Morton code, we convert floating-point coordinates to integer voxel keys:

$$\mathbf{k} = \lfloor \mathbf{p}^W / s \rfloor \tag{2}$$

where $\mathbf{p}^W$ is the point in world frame and s is the voxel size.

3.3.1 Implementation

Listing 6: Coordinate quantization (src/core/VoxelMap.cpp)

```
// Convert world coordinate to voxel key
Eigen::Vector3i pointToVoxelKey(const Eigen::Vector3d& point,
                                double voxel_size) {
    return Eigen::Vector3i(
        static_cast<int>(std::floor(point.x() / voxel_size)),
        static_cast<int>(std::floor(point.y() / voxel_size)),
        static_cast<int>(std::floor(point.z() / voxel_size))
    );
}

// Full pipeline: point -> voxel key -> Morton code
uint64_t pointToMortonCode(const Eigen::Vector3d& point,
                           double voxel_size) {
    Eigen::Vector3i key = pointToVoxelKey(point, voxel_size);
    return computeMortonCode(key.x(), key.y(), key.z());
```

```
}
```

3.3.2 Example

```
Point: (1.7, 2.3, 0.5) meters
Voxel size: 0.5 meters

Voxel key: floor([1.7, 2.3, 0.5] / 0.5) = (3, 4, 1)

Morton code: interleave(3, 4, 1)
  3 = 011, 4 = 100, 1 = 001
  Interleaved: 010 011 101 = 157 (decimal)
```

3.4 Hierarchical Key Relationship

A key advantage of our design is that L0 and L1 voxel keys have a simple mathematical relationship:

$$\mathbf{k}_1 = \lfloor \mathbf{k}_0 / 3 \rfloor \tag{3}$$

Since $s_1 = 3 \cdot s_0$, the parent L1 voxel key is obtained by integer division.

3.4.1 Implementation

Listing 7: Hierarchical key conversion (src/core/VoxelMap.cpp)

```cpp
// Get parent L1 key from L0 key
Eigen::Vector3i getParentKey(const Eigen::Vector3i& l0_key) {
    return Eigen::Vector3i(
        static_cast<int>(std::floor(l0_key.x() / 3.0)),
        static_cast<int>(std::floor(l0_key.y() / 3.0)),
        static_cast<int>(std::floor(l0_key.z() / 3.0))
    );
}

// Alternative: direct Morton code manipulation
// (More efficient but requires careful bit handling)
uint64_t getParentMortonCode(uint64_t l0_morton) {
    // Extract x, y, z from Morton code
    int x = extractX(l0_morton);
    int y = extractY(l0_morton);
    int z = extractZ(l0_morton);

    // Integer division by 3
    int px = x / 3;
    int py = y / 3;
    int pz = z / 3;

    return computeMortonCode(px, py, pz);
}
```

3.5 Cache Efficiency Analysis

The Z-order curve traces a recursive Z-shaped path through 3D space, ensuring that:

1. Spatially adjacent voxels have similar Morton codes

2. Similar Morton codes map to nearby hash table entries
3. Nearby entries likely reside in the same cache line

Quantitative Impact:

Consider processing a LiDAR scan with 20,000 points. With standard hashing:

- Each point may cause a cache miss: $20,000 \times 100$ cycles $= 2,000,000$ cycles

With Morton code hashing (assuming 80% cache hit rate):

- Cache hits: $16,000 \times 4$ cycles $= 64,000$ cycles
- Cache misses: $4,000 \times 100$ cycles $= 400,000$ cycles
- Total: $464,000$ cycles ($\sim$4.3$\times$ faster)

Table 2: Hash function comparison for spatial data

Method	Computation	Locality	Overhead
std::hash	O(1)	Poor	None
Morton (Z-order)	O(1)	Good	Bit ops
Hilbert curve	O(log n)	Best	Lookup table

We choose Morton codes as they provide good locality with minimal computational overhead—just bit interleaving operations that modern CPUs execute in a few cycles.

4 Voxel Map Operations

This section describes the core operations of the hierarchical voxel map (hVox), including map updates and correspondence finding.

4.1 Map Update Pipeline

When a new LiDAR scan arrives, each point is inserted into the hierarchical voxel structure:

Listing 8: VoxelMap::updateMap (VoxelMap.cpp)

```
void VoxelMap::updateMap(
    const std::vector<Eigen::Vector3d> &points,
    const Eigen::Matrix3d &R,
    const Eigen::Vector3d &t)
{
    for (const auto &pt_L : points)
    {
        // Transform point from LiDAR frame to world frame
        Eigen::Vector3d pt_W = R * pt_L + t;

        // Compute L0 voxel key using Morton code
        int ix = static_cast<int>(std::floor(pt_W.x() / l0VoxelSize_));
        int iy = static_cast<int>(std::floor(pt_W.y() / l0VoxelSize_));
        int iz = static_cast<int>(std::floor(pt_W.z() / l0VoxelSize_));
        uint64_t l0Key = computeMortonCode(ix, iy, iz);

        // Find or create L0 voxel
        auto &l0Voxel = l0Map_[l0Key];

        // Compute local index within L0 voxel for L1 voxel
        int localX = static_cast<int>(std::floor(
```

```
            (pt_W.x() - ix * l0VoxelSize_) / l1VoxelSize_));
        int localY = static_cast<int>(std::floor(
            (pt_W.y() - iy * l0VoxelSize_) / l1VoxelSize_));
        int localZ = static_cast<int>(std::floor(
            (pt_W.z() - iz * l0VoxelSize_) / l1VoxelSize_));

        // L1 voxel index (0-26 for 3x3x3 subdivision)
        int l1Idx = localX + localY * l0L1Ratio_
                    + localZ * l0L1Ratio_ * l0L1Ratio_;

        // Update L1 voxel (surfel)
        l0Voxel.l1Voxels[l1Idx].addPoint(pt_W);
    }
}
```

4.2 Incremental Surfel Update

Each L1 voxel maintains a surfel that is updated incrementally as points are added. This corresponds to Equations 1–5 in the paper:

Listing 9: L1Voxel::addPoint - Incremental statistics (VoxelMap.cpp)

```
void L1Voxel::addPoint(const Eigen::Vector3d &pt)
{
    numPoints_++;

    // Eq. 1: Incremental centroid update
    // mu_k = mu_{k-1} + (p_k - mu_{k-1}) / k
    Eigen::Vector3d delta = pt - centroid_;
    centroid_ += delta / static_cast<double>(numPoints_);

    // Eq. 2: Incremental covariance update
    // Uses Welford's online algorithm for numerical stability
    if (numPoints_ > 1)
    {
        Eigen::Vector3d delta2 = pt - centroid_;
        // M_k = M_{k-1} + (p_k - mu_{k-1})(p_k - mu_k)^T
        covMatrix_ += delta * delta2.transpose();
    }

    // Mark surfel as needing recomputation
    surfelValid_ = false;
}
```

4.3 Lazy Surfel Computation

Surfel normal and planarity are computed lazily (only when needed) via eigendecomposition:

Listing 10: L1Voxel::computeSurfel - PCA via eigendecomposition (VoxelMap.cpp)

```
void L1Voxel::computeSurfel()
{
    if (surfelValid_ || numPoints_ < minPointsForSurfel_)
        return;

    // Eq. 3: Covariance matrix C = M / (n-1)
    Eigen::Matrix3d C = covMatrix_ / (numPoints_ - 1);

    // Eq. 4: Eigendecomposition C = V * Lambda * V^T
    Eigen::SelfAdjointEigenSolver<Eigen::Matrix3d> solver(C);
    eigenvalues_ = solver.eigenvalues();    // lambda1 <= lambda2 <= lambda3
    eigenvectors_ = solver.eigenvectors();
```

```
13
14      // Normal vector = eigenvector of smallest eigenvalue
15      normal_ = eigenvectors_.col(0);
16
17      // Eq. 5: Planarity check (lambda1 << lambda2)
18      planarity_ = (eigenvalues_(1) - eigenvalues_(0))
19                   / (eigenvalues_(2) + 1e-6);
20
21      surfelValid_ = true;
22  }
```

4.4 Correspondence Finding

The key innovation of Surfel-LIO is O(1) correspondence retrieval. Given a query point, we directly compute the voxel key and retrieve the pre-computed surfel:

Listing 11: VoxelMap::findCorrespondence (VoxelMap.cpp)

```
1   bool VoxelMap::findCorrespondence(
2       const Eigen::Vector3d &queryPt,
3       Eigen::Vector3d &closestPoint,
4       Eigen::Vector3d &normal) const
5   {
6       // Step 1: Compute L0 voxel key (O(1) hash computation)
7       int ix = static_cast<int>(std::floor(queryPt.x() / l0VoxelSize_));
8       int iy = static_cast<int>(std::floor(queryPt.y() / l0VoxelSize_));
9       int iz = static_cast<int>(std::floor(queryPt.z() / l0VoxelSize_));
10      uint64_t l0Key = computeMortonCode(ix, iy, iz);
11
12      // Step 2: O(1) hash table lookup
13      auto it = l0Map_.find(l0Key);
14      if (it == l0Map_.end())
15          return false;
16
17      // Step 3: Compute L1 voxel index within L0
18      const auto &l0Voxel = it->second;
19      int localX = static_cast<int>(std::floor(
20          (queryPt.x() - ix * l0VoxelSize_) / l1VoxelSize_));
21      int localY = static_cast<int>(std::floor(
22          (queryPt.y() - iy * l0VoxelSize_) / l1VoxelSize_));
23      int localZ = static_cast<int>(std::floor(
24          (queryPt.z() - iz * l0VoxelSize_) / l1VoxelSize_));
25      int l1Idx = localX + localY * l0L1Ratio_
26                  + localZ * l0L1Ratio_ * l0L1Ratio_;
27
28      // Step 4: Retrieve pre-computed surfel
29      const auto &l1Voxel = l0Voxel.l1Voxels[l1Idx];
30      if (!l1Voxel.isValid())
31          return false;
32
33      // Step 5: Return surfel data (no nearest-neighbor search!)
34      closestPoint = l1Voxel.getCentroid();
35      normal = l1Voxel.getNormal();
36
37      return true;
38  }
```

4.5 Complexity Analysis

The correspondence finding achieves O(1) complexity through:

1. **Morton code computation**: Simple bit operations (Eq. 6)

2. **Hash table lookup**: Robin Hood hashing provides O(1) average case
3. **Pre-computed surfels**: No runtime PCA or nearest-neighbor search

Listing 12: Batch correspondence finding for scan registration

```cpp
void Estimator::findAllCorrespondences(
    const std::vector<Eigen::Vector3d> &points,
    const Eigen::Matrix3d &R,
    const Eigen::Vector3d &t,
    std::vector<Correspondence> &correspondences)
{
    correspondences.clear();
    correspondences.reserve(points.size());

    // OpenMP parallelization for batch processing
    #pragma omp parallel for
    for (size_t i = 0; i < points.size(); i++)
    {
        Eigen::Vector3d pt_W = R * points[i] + t;
        Eigen::Vector3d closestPt, normal;

        // O(1) correspondence lookup per point
        if (voxelMap_.findCorrespondence(pt_W, closestPt, normal))
        {
            #pragma omp critical
            {
                correspondences.emplace_back(
                    points[i], closestPt, normal);
            }
        }
    }
}
```

This design enables processing speeds of 500+ FPS on standard hardware, as each of the 20,000 points per scan requires only O(1) operations for correspondence finding.

5 State Estimation (IEKF)

This section describes the Iterated Extended Kalman Filter (IEKF) used to fuse IMU measurements with LiDAR observations.

5.1 State Vector Definition

Paper Reference: Equation (7)

The state vector contains the robot's pose, velocity, and IMU biases:

$$\mathbf{x} = \begin{bmatrix} \mathbf{R}^\top & \mathbf{p}^\top & \mathbf{v}^\top & \mathbf{b}_g^\top & \mathbf{b}_a^\top \end{bmatrix}^\top \in SO(3) \times \mathbb{R}^{12} \qquad \text{(Paper Eq. 7)}$$

5.1.1 Implementation

Listing 13: State representation (src/core/State.h)

```cpp
struct State {
    // Rotation (SO3)
    Eigen::Matrix3d R;      // 3x3 rotation matrix, body to world

    // Position and velocity (R^3 each)
    Eigen::Vector3d p;      // Position in world frame [m]
```

Table 3: State vector components

Symbol	Dimension	Description
$\mathbf{R}$	$SO(3)$	Rotation from body to world frame
$\mathbf{p}$	$\mathbb{R}^3$	Position in world frame
$\mathbf{v}$	$\mathbb{R}^3$	Velocity in world frame
$\mathbf{b}_g$	$\mathbb{R}^3$	Gyroscope bias
$\mathbf{b}_a$	$\mathbb{R}^3$	Accelerometer bias

```
    Eigen::Vector3d v;      // Velocity in world frame [m/s]

    // IMU biases (R^3 each)
    Eigen::Vector3d b_g;    // Gyroscope bias [rad/s]
    Eigen::Vector3d b_a;    // Accelerometer bias [m/s^2]

    // Gravity vector (constant, estimated during initialization)
    Eigen::Vector3d g;      // Gravity in world frame [m/s^2]

    // Covariance matrix (15x15 for error state)
    Eigen::Matrix<double, 15, 15> P;

    State() {
        R = Eigen::Matrix3d::Identity();
        p = Eigen::Vector3d::Zero();
        v = Eigen::Vector3d::Zero();
        b_g = Eigen::Vector3d::Zero();
        b_a = Eigen::Vector3d::Zero();
        g = Eigen::Vector3d(0, 0, -9.81);
        P = Eigen::Matrix<double, 15, 15>::Identity() * 0.001;
    }
};
```

5.1.2 Error State Parameterization

For covariance propagation and update, we use the error state:

$$\delta\mathbf{x} = \begin{bmatrix}\delta\boldsymbol{\theta}^\top & \delta\mathbf{p}^\top & \delta\mathbf{v}^\top & \delta\mathbf{b}_g^\top & \delta\mathbf{b}_a^\top\end{bmatrix}^\top \in \mathbb{R}^{15} \tag{4}$$

where $\delta\boldsymbol{\theta} \in \mathbb{R}^3$ parameterizes rotation error via:

$$\mathbf{R}_{\text{true}} = \mathbf{R} \cdot \text{Exp}(\delta\boldsymbol{\theta}) \tag{5}$$

5.2 IMU Propagation

Paper Reference: Equations (8)–(10)

Upon receiving an IMU measurement $(\boldsymbol{\omega}_k, \mathbf{a}_k)$, we propagate the state using discrete-time kinematics.

5.2.1 Theory

Rotation update:

$$\hat{\mathbf{R}}_{k+1} = \hat{\mathbf{R}}_k \cdot \text{Exp}((\boldsymbol{\omega}_k - \mathbf{b}_g)\Delta t) \qquad \text{(Paper Eq. 8)}$$

Velocity update:

$$\hat{\mathbf{v}}_{k+1} = \hat{\mathbf{v}}_k + (\hat{\mathbf{R}}_k(\mathbf{a}_k - \mathbf{b}_a) + \mathbf{g})\Delta t \qquad \text{(Paper Eq. 9)}$$

Position update:

$$\hat{\mathbf{p}}_{k+1} = \hat{\mathbf{p}}_k + \hat{\mathbf{v}}_k \Delta t + \frac{1}{2}(\hat{\mathbf{R}}_k(\mathbf{a}_k - \mathbf{b}_a) + \mathbf{g})\Delta t^2 \quad \text{(Paper Eq. 10)}$$

5.2.2 Implementation

Listing 14: IMU propagation (src/core/Estimator.cpp)

```cpp
void Estimator::propagateIMU(const IMUData& imu, double dt) {
    // Bias-corrected measurements
    Eigen::Vector3d omega = imu.gyro - state_.b_g;  // Angular velocity
    Eigen::Vector3d acc = imu.accel - state_.b_a;   // Linear acceleration

    // Rotation update (Eq. 8)
    Eigen::Matrix3d dR = expSO3(omega * dt);  // Exponential map
    Eigen::Matrix3d R_new = state_.R * dR;

    // Acceleration in world frame
    Eigen::Vector3d acc_world = state_.R * acc + state_.g;

    // Position update (Eq. 10) - use current velocity
    Eigen::Vector3d p_new = state_.p + state_.v * dt
                          + 0.5 * acc_world * dt * dt;

    // Velocity update (Eq. 9)
    Eigen::Vector3d v_new = state_.v + acc_world * dt;

    // Update state
    state_.R = R_new;
    state_.p = p_new;
    state_.v = v_new;

    // Propagate covariance
    propagateCovariance(omega, acc, dt);
}
```

5.2.3 SO(3) Exponential Map

Listing 15: SO(3) exponential map (src/util/LieUtils.cpp)

```cpp
// Convert axis-angle to rotation matrix
Eigen::Matrix3d expSO3(const Eigen::Vector3d& omega) {
    double theta = omega.norm();

    if (theta < 1e-10) {
        // Small angle approximation: exp(omega)     I + [omega]_x
        return Eigen::Matrix3d::Identity() + skewSymmetric(omega);
    }

    // Rodrigues' formula
    Eigen::Vector3d axis = omega / theta;
    Eigen::Matrix3d K = skewSymmetric(axis);

    return Eigen::Matrix3d::Identity()
         + std::sin(theta) * K
         + (1 - std::cos(theta)) * K * K;
}

// Skew-symmetric matrix from vector
Eigen::Matrix3d skewSymmetric(const Eigen::Vector3d& v) {
    Eigen::Matrix3d S;
    S <<     0, -v.z(),  v.y(),
```

```
        v.z(),       0, -v.x(),
       -v.y(),  v.x(),       0;
    return S;
}
```

5.3 Point-to-Plane Residual

Paper Reference: Equation (11)

Each LiDAR point with a valid surfel correspondence contributes a point-to-plane residual.

5.3.1 Theory

Given a point $\mathbf{p}_i^L$ in LiDAR frame and its corresponding surfel with normal $\mathbf{n}_i$ and centroid $\mathbf{c}_i$:

$$r_i = \mathbf{n}_i^\top \left(\mathbf{R}\mathbf{p}_i^L + \mathbf{p} - \mathbf{c}_i \right) \qquad \text{(Paper Eq. 11)}$$

This measures the signed distance from the transformed point to the surfel plane.

5.3.2 Implementation

Listing 16: Point-to-plane residual (src/core/Estimator.cpp)

```
double Estimator::computeResidual(const Eigen::Vector3d& p_lidar,
                                  const Surfel& surfel) {
    // Transform point to world frame
    Eigen::Vector3d p_world = state_.R * p_lidar + state_.p;

    // Point-to-plane distance (Eq. 11)
    double residual = surfel.normal.dot(p_world - surfel.centroid);

    return residual;
}
```

5.4 Jacobian Computation

Paper Reference: Equation (12)

The IEKF requires the Jacobian of the residual with respect to the error state.

5.4.1 Theory

$$\mathbf{H}_i = \frac{\partial r_i}{\partial \delta \mathbf{x}} = \begin{bmatrix} -\mathbf{n}_i^\top \mathbf{R} [\mathbf{p}_i^L]_\times & \mathbf{n}_i^\top & \mathbf{0}_{1\times 9} \end{bmatrix} \qquad \text{(Paper Eq. 12)}$$

where $[\cdot]_\times$ denotes the skew-symmetric matrix.

5.4.2 Derivation

The residual depends on rotation and position:

$$r_i = \mathbf{n}_i^\top (\mathbf{R}\mathbf{p}_i^L + \mathbf{p} - \mathbf{c}_i) \tag{6}$$

Derivative w.r.t. position error $\delta\mathbf{p}$:

$$\frac{\partial r_i}{\partial \delta \mathbf{p}} = \mathbf{n}_i^\top \tag{7}$$

Derivative w.r.t. rotation error $\delta\boldsymbol{\theta}$:

Using $\mathbf{R}_{\text{true}} = \mathbf{R} \cdot \text{Exp}(\delta\boldsymbol{\theta}) \approx \mathbf{R}(\mathbf{I} + [\delta\boldsymbol{\theta}]_\times)$:

$$\frac{\partial r_i}{\partial \delta\boldsymbol{\theta}} = \mathbf{n}_i^\top \frac{\partial}{\partial \delta\boldsymbol{\theta}} \left(\mathbf{R}(\mathbf{I} + [\delta\boldsymbol{\theta}]_\times)\mathbf{p}_i^L\right) \tag{8}$$

$$= \mathbf{n}_i^\top \mathbf{R} \frac{\partial}{\partial \delta\boldsymbol{\theta}} \left([\delta\boldsymbol{\theta}]_\times \mathbf{p}_i^L\right) \tag{9}$$

$$= -\mathbf{n}_i^\top \mathbf{R} [\mathbf{p}_i^L]_\times \tag{10}$$

5.4.3 Implementation

Listing 17: Jacobian computation (src/core/Estimator.cpp)

```
Eigen::Matrix<double, 1, 15> Estimator::computeJacobian(
        const Eigen::Vector3d& p_lidar,
        const Surfel& surfel) {

    Eigen::Matrix<double, 1, 15> H = Eigen::Matrix<double, 1, 15>::Zero();

    // Skew-symmetric matrix of point in LiDAR frame
    Eigen::Matrix3d p_skew = skewSymmetric(p_lidar);

    // Jacobian w.r.t. rotation error (columns 0-2)
    H.block<1, 3>(0, 0) = -surfel.normal.transpose() * state_.R * p_skew;

    // Jacobian w.r.t. position error (columns 3-5)
    H.block<1, 3>(0, 3) = surfel.normal.transpose();

    // Jacobian w.r.t. velocity, bias_g, bias_a (columns 6-14)
    // These are zero for point-to-plane residual

    return H;
}
```

5.5 IEKF Update Loop

The IEKF iteratively refines the state estimate until convergence.

Listing 18: IEKF update loop (src/core/Estimator.cpp)

```
void Estimator::updateLiDAR(const std::vector<Eigen::Vector3d>& points_lidar) {
    // Store initial state for iteration
    State state_init = state_;

    for (int iter = 0; iter < max_iterations_; ++iter) {
        // Find correspondences with current state estimate
        std::vector<Eigen::Vector3d> points_world;
        for (const auto& p : points_lidar) {
            points_world.push_back(state_.R * p + state_.p);
        }
        auto correspondences = voxel_map_.findCorrespondences(points_world);

        // Build measurement matrices
        int num_valid = std::count_if(correspondences.begin(),
                                      correspondences.end(),
                                      [](const auto& c) { return c.valid; });

        if (num_valid < min_correspondences_) {
            std::cerr << "Insufficient correspondences: " << num_valid << std::
                endl;
            return;
```

```
        }

        Eigen::MatrixXd H(num_valid, 15);
        Eigen::VectorXd r(num_valid);
        Eigen::MatrixXd R_meas = Eigen::MatrixXd::Identity(num_valid, num_valid
            )
                                 * measurement_noise_;

        int row = 0;
        for (size_t i = 0; i < correspondences.size(); ++i) {
            if (!correspondences[i].valid) continue;

            const auto& p_lidar = points_lidar[i];
            const auto& surfel = *correspondences[i].surfel;

            H.row(row) = computeJacobian(p_lidar, surfel);
            r(row) = computeResidual(p_lidar, surfel);
            ++row;
        }

        // Kalman gain
        Eigen::MatrixXd S = H * state_.P * H.transpose() + R_meas;
        Eigen::MatrixXd K = state_.P * H.transpose() * S.inverse();

        // State correction
        Eigen::VectorXd dx = K * r;

        // Apply correction
        state_.R = state_.R * expSO3(dx.segment<3>(0));
        state_.p -= dx.segment<3>(3);
        state_.v -= dx.segment<3>(6);
        state_.b_g -= dx.segment<3>(9);
        state_.b_a -= dx.segment<3>(12);

        // Check convergence
        if (dx.norm() < convergence_thresh_) {
            break;
        }
    }

    // Update covariance
    Eigen::MatrixXd I_KH = Eigen::MatrixXd::Identity(15, 15) - K * H;
    state_.P = I_KH * state_.P * I_KH.transpose()
             + K * R_meas * K.transpose();  // Joseph form for stability
}
```

6 Configuration Parameters

This section describes all configurable parameters and their effects on system performance.

6.1 Configuration File Structure

Configuration files are in YAML format, located in the `config/` directory:

```
config/
|-- avia.yaml       # Livox AVIA configuration
+-- mid360.yaml     # Livox Mid-360 configuration
```

6.2 Voxel Parameters

Listing 19: Voxel configuration (config/avia.yaml)

```yaml
voxel:
  size: 0.5                 # L0 voxel edge length [m]
  map_extent: 200.0         # Local map radius [m]

surfel:
  min_planarity: 0.1        # Minimum planarity for valid surfel
  min_points: 3             # Minimum L0 children for surfel
  eager_update: false       # Use lazy update by default
```

Table 4: Voxel parameter descriptions

Parameter	Description
`voxel.size`	L0 voxel edge length. Smaller values give finer resolution but more memory. L1 size is automatically 3× this value.
`voxel.map_extent`	Radius of local map around robot. Points outside are removed periodically.
`surfel.min_planarity`	Minimum ρ value (Eq. 5) for valid correspondence. Higher values reject edges/-corners.
`surfel.min_points`	Minimum occupied L0 children required to compute surfel.
`surfel.eager_update`	If true, recompute surfels immediately after map update.

Tuning Guidelines:

- **Indoor**: `size: 0.3`, `min_planarity: 0.15` (finer resolution, stricter planes)
- **Outdoor**: `size: 0.5-1.0`, `min_planarity: 0.1` (coarser resolution, more tolerant)
- **Sparse environments**: Lower `min_points` to 2

6.3 IEKF Parameters

Listing 20: IEKF configuration (config/avia.yaml)

```yaml
iekf:
  max_iterations: 5         # Maximum IEKF iterations per scan
  convergence_thresh: 0.001 # Stop if ||dx|| < this value
  min_correspondences: 100  # Minimum valid correspondences
  measurement_noise: 0.01   # Observation noise variance
```

6.4 IMU Parameters

Listing 21: IMU configuration (config/avia.yaml)

```yaml
imu:
  # Noise parameters (continuous-time)
  gyro_noise: 0.01          # Gyroscope noise [rad/s/sqrt(Hz)]
  accel_noise: 0.1          # Accelerometer noise [m/s^2/sqrt(Hz)]
  gyro_bias_noise: 0.0001   # Gyroscope bias random walk
  accel_bias_noise: 0.001   # Accelerometer bias random walk

  # Initial bias estimates (optional)
  init_gyro_bias: [0.0, 0.0, 0.0]
```

Table 5: IEKF parameter descriptions

Parameter	Description
`max_iterations`	Maximum number of IEKF iterations. More iterations improve accuracy but increase latency.
`convergence_thresh`	Early termination threshold on $\|\delta\mathbf{x}\|$.
`min_correspondences`	Minimum valid point-surfel pairs required for update.
`measurement_noise`	Variance of point-to-plane residual noise (σ^2).

```
  init_accel_bias: [0.0, 0.0, 0.0]

  # Gravity magnitude
  gravity: 9.81
```

Obtaining Noise Parameters:

IMU noise parameters are typically found in the sensor datasheet or estimated via Allan variance analysis. Common values:

Table 6: Typical IMU noise values

IMU Grade	Gyro Noise	Accel Noise	Example
Consumer	0.01–0.05	0.1–0.5	BMI088, MPU6050
Industrial	0.001–0.01	0.01–0.1	ADIS16448
Tactical	<0.001	<0.01	KVH 1750

6.5 Sensor Extrinsics

The transformation from LiDAR to IMU frame:

Listing 22: Extrinsic calibration (config/avia.yaml)

```
extrinsic:
  # Rotation (quaternion: w, x, y, z)
  q_lidar_to_imu: [1.0, 0.0, 0.0, 0.0]

  # Translation [m]
  t_lidar_to_imu: [0.05, 0.0, 0.1]

  # Time offset: t_lidar = t_imu + offset
  time_offset: 0.0
```

Calibration Methods:

- **CAD model**: Measure from mechanical drawings
- **lidar_imu_calib**: Open-source calibration tool
- **Manual tuning**: Adjust until trajectory is smooth

6.6 Sensor-Specific Configurations

6.6.1 Livox AVIA (config/avia.yaml)

Listing 23: AVIA-specific settings

```yaml
lidar:
  type: "avia"
  scan_rate: 10.0           # Hz
  fov_deg: 70.0             # Circular FOV
  points_per_scan: 24000    # Approximate
  blind_range: 0.5          # Ignore points closer than this [m]

voxel:
  size: 0.5                 # Good for AVIA point density

iekf:
  max_iterations: 5         # AVIA has dense scans
```

6.6.2 Livox Mid-360 (config/mid360.yaml)

Listing 24: Mid-360-specific settings

```yaml
lidar:
  type: "mid360"
  scan_rate: 10.0           # Hz
  fov_deg: 360.0            # Horizontal FOV
  points_per_scan: 15000    # Approximate
  blind_range: 0.3          # Ignore points closer than this [m]

voxel:
  size: 0.5                 # Same voxel size works well

iekf:
  max_iterations: 4         # Fewer points, fewer iterations needed
```

6.7 Complete Configuration Example

Listing 25: Complete avia.yaml example

```yaml
# Surfel-LIO Configuration for Livox AVIA
# ========================================

lidar:
  type: "avia"
  scan_rate: 10.0
  blind_range: 0.5

imu:
  gyro_noise: 0.01
  accel_noise: 0.1
  gyro_bias_noise: 0.0001
  accel_bias_noise: 0.001
  gravity: 9.81

extrinsic:
  q_lidar_to_imu: [1.0, 0.0, 0.0, 0.0]
  t_lidar_to_imu: [0.05, 0.0, 0.1]
  time_offset: 0.0

voxel:
  size: 0.5
  map_extent: 200.0

surfel:
  min_planarity: 0.1
```

```
  min_points: 3
  eager_update: false

iekf:
  max_iterations: 5
  convergence_thresh: 0.001
  min_correspondences: 100
  measurement_noise: 0.01

visualization:
  enable: true
  point_size: 2.0
  trajectory_length: 1000
```