Appendix
RAFT + Height Calculator + Stitching Technical Overview

1 System Architecture

This system combines aerial imagery with LiDAR measurements to create detailed cloud height fields using a deep learning approach. The process consists of four main stages:

1.1 Data Collection and Preprocessing

- HD camera images from FEGS (Fly's Eye GLM Simulator)
- LiDAR measurements from Cloud Physics LiDAR (CPL)
- Aircraft flight metadata (altitude, speed, orientation)
- Image synchronization and alignment

1.2 Motion Analysis and Height Estimation

- Fine-tuned RAFT (Recurrent All-Pairs Field Transforms) model for optical flow
- Parallax-based height calculation using motion fields
- Integration with aircraft metadata for scale calibration
- Confidence mapping for measurement reliability

1.3 Height Field Generation and Validation

- Height field calculation from motion vectors
- LiDAR measurement integration for calibration
- Uncertainty estimation and quality control
- Individual sequence height field generation

1.4 Height Field Stitching and Global Integration

- Global coordinate system creation using GPS data
- Weighted height field merging
- Multi-sequence confidence integration
- Large-scale cloud topology reconstruction

2 Key Processing Steps

2.1 Data Input Preparation

- Temporal alignment of image sequences (5-second intervals)
- Image preprocessing and augmentation
- Metadata normalization and feature engineering
- Sequence packaging for batch processing

2.2 Motion Field Generation

- RAFT model application to image pairs
- Bidirectional flow calculation
- Motion field refinement and consistency checking
- Confidence map generation

2.3 Height Field Calculation

- Parallax principle application
- Aircraft motion compensation
- Scale factor calibration using LiDAR
- Height field uncertainty estimation

2.4 Global Field Stitching

- GPS-based sequence positioning
- Distance-weighted height field merging
- Confidence-based blending
- Banding artifact removal
- Seamless transition handling

3 System Advantages

3.1 Wide Field Coverage

- 90-degree field of view compared to single-point LiDAR
- Continuous spatial coverage
- Higher spatial resolution
- Large-scale cloud field reconstruction through stitching

3.2 Cost Efficiency

- Uses existing camera hardware
- Reduces dependency on expensive LiDAR systems
- Simplified instrument package
- Maximizes value from existing flight data

3.3 Measurement Quality

- LiDAR-calibrated accuracy
- Uncertainty quantification
- Real-time quality assessment
- Multi-sequence validation
- Robust to measurement gaps

3.4 Comprehensive Cloud Mapping

- Seamless integration of multiple sequences
- Preservation of local detail
- Global context maintenance
- Confidence-weighted merging
- Artifact-free reconstruction

This system bridges the gap between limited single-point LiDAR measurements and the need for cloud height fields, enabling more accurate atmospheric observations and improved weather predictions. The stitching capability allows for reconstruction of large-scale cloud fields, providing coverage and detail in cloud height mapping from aerial platforms.

4 RAFT Architecture and Memory Usage

RAFT (Recurrent All-Pairs Field Transforms) is an optical flow model that works by building dense correlations between all pixels in a pair of images and iteratively refining flow predictions. It uses a significant amount of memory. In our implementation, we create a number of helper functions to monitor and report memory usage.

5 Core Components

5.1 Feature Extraction

The first step in RAFT is feature extraction, where each input image is processed through a convolutional encoder network. This encoder transforms each pixel into a rich feature representation containing hundreds of channels of information. Rather than producing a single feature map, RAFT creates a feature pyramid with multiple resolution levels. This pyramid structure allows the model to capture both fine details and broader image context, but it means storing multiple feature maps of varying sizes for each image.

5.2 All-Pairs Correlation

The correlation computation is where RAFT's memory usage truly explodes. For every single pixel in the first image, RAFT computes correlation scores with every possible pixel in the second image. This creates an enormous 4D correlation volume of size $H \times W \times H \times W$, where H and W are the image height and width. To put this in perspective, for our 384×384 pixel images, the correlation volume alone requires 384^4 elements. Even using 4-byte floats, that's approximately 55GB of memory needed just to store a single pair of images' correlation volume! This quadratic scaling with image size is the primary reason for RAFT's intense memory requirements.

5.3 Iterative Refinement

The final major component is RAFT's iterative refinement mechanism, which uses a Gated Recurrent Unit (GRU) to progressively update and refine flow estimates. As the GRU processes each pixel's motion, it looks up relevant correlation features based on the current estimated position. This process typically runs for 12-32 iterations, maintaining hidden states throughout. While not as memory-intensive as the correlation volume, these iterations require storing both the hidden states and intermediate results for backpropagation during training.

6 Why Sequences Are Memory-Intensive

When processing sequences, memory usage multiplies because:

- Correlation volumes are needed for each consecutive pair of frames
- Intermediate activations are stored for backpropagation
- GRU hidden states are maintained across the sequence
- Each additional frame essentially adds another full RAFT computation

For a sequence of length T:

$$\text{Memory} \propto (T-1) \times (H^2 W^2)$$

The quadratic scaling with image size (H^2W^2) makes this particularly challenging. Even with optimizations like limiting correlation range or using lower resolutions, the memory requirements grow rapidly with sequence length.

This is why RAFT, despite having a relatively modest number of parameters (5-6M), can easily consume gigabytes of GPU memory during training, especially with longer sequences. Each additional frame in the sequence means another massive correlation volume must be computed and stored for backpropagation.

7 Available Aircraft Metadata

7.1 Temporal and Position Data

- **DateTime_UTC**: ISO-8601 formatted date and time in UTC (yyyy-mm-ddThh:mm:ss)
- **Lat**: Platform Latitude (degrees N, -90 to 90)
- **Lon**: Platform Longitude (degrees E, -180 to 179.9999)

7.2 Altitude Measurements

- **GPS_MSL_Alt**: GPS Altitude above Mean Sea Level (meters)
- **WGS_84_Alt**: WGS 84 Geoid Altitude (meters)
- **Press_Alt**: Pressure Altitude (feet)

7.3 Velocity Parameters

- **Grnd_Spd**: Ground Speed (m/s)
- **True_Airspeed**: True Airspeed (m/s)
- **Mach_Number**: Aircraft Mach Number (dimensionless)
- **Vert_Velocity**: Aircraft Vertical Velocity (m/s, negative=downward, positive=upward)

7.4 Aircraft Orientation

- **True_Hdg**: True Heading (degrees true, 0 to 359.9999)
- **Track**: Track Angle (degrees true, 0 to 359.9999)
- **Drift**: Drift Angle (degrees)
- **Pitch**: Pitch (degrees, -90 to 90, negative=nose down, positive=nose up)
- **Roll**: Roll (degrees, -90 to 90, negative=left wing down, positive=right wing down)

7.5 Environmental Conditions

- **Ambient_Temp**: Ambient Temperature (°C)
- **Total_Temp**: Total Temperature (°C)
- **Static_Press**: Static Pressure (millibars)
- **Dynamic_Press**: Dynamic Pressure - total minus static (millibars)
- **Cabin_Pressure**: Cabin Pressure/Altitude (millibars)

7.6 Wind Data

- **Wind_Speed**: Wind Speed (m/s, ≥ 0)
- **Wind_Dir**: Wind Direction (degrees true, 0 to 359.9999)

7.7 Solar Position

- **Solar_Zenith**: Solar Zenith Angle (degrees)
- **Sun_Elev_AC**: Sun Elevation from Aircraft (degrees)
- **Sun_Az_Grd**: Sun Azimuth from Ground (degrees true, 0 to 359.9999)
- **Sun_Az_AC**: Sun Azimuth from Aircraft (degrees true, 0 to 359.9999)

8 Random Scaling and Cropping Strategy

8.1 Motivation

The goal behind this data augmentation strategy is to help the model become more robust at estimating cloud heights when the LiDAR measurement point isn't always in the center of the image. This is important because:

1. **Original Data Limitation**
    - LiDAR measurements are always taken at the exact center of each image.
    - This could cause the model to overfit to center-focused features.
    - Real-world applications may need height estimates from different parts of the image.
2. **Spatial Understanding**
    - The model needs to understand cloud height features regardless of their position.
    - It should learn relationships between cloud features across the entire image.
    - This is important for generalizing to different viewing angles and positions.

8.2 Implementation Strategy

8.2.1 1. Random Cropping (random_crop_sequence)

Algorithm: Random Crop Sequence
Input: image_sequence, min_crop_size = 384
Output: cropped_sequence, new_center_coords
—
orig_height, orig_width ← image_size
crop_size ← random_int(min_crop_size, min(orig_height, orig_width))
top ← random_int(0, orig_height - crop_size)
left ← random_int(0, orig_width - crop_size)
orig_center ← (orig_width/2, orig_height/2)
new_center ← (orig_center.x - left, orig_center.y - top)
cropped_sequence ← crop_images(sequence, top, left, crop_size)

Key Features:

- Random crop size between minimum (384) and image dimension.
- Random position for the crop window.
- Maintains aspect ratio (square crop).
- Original center point is transformed to new coordinates.

8.2.2 2. Resizing (resize_sequence_and_adjust_center)

Algorithm: Resize and Adjust Center
Input: cropped_sequence, center_coords, target_size = 384
Output: resized_sequence, adjusted_center_coords
—
scale_factor ← target_size/crop_size
new_center.x ← center_coords.x × scale_factor
new_center.y ← center_coords.y × scale_factor
resized_sequence ← resize_images(cropped_sequence, target_size)

8.3 Benefits of This Approach

1. **Data Augmentation**
   - Effectively multiplies training data.
   - Creates variations in LiDAR measurement position.
   - Introduces scale variation while preserving aspect ratios.
2. **Model Robustness**
   - Forces the model to look at cloud features everywhere in the image.
   - Prevents overfitting to center-specific patterns.
   - Better generalization to different viewing conditions.
3. **Training Stability**
   - Consistent final image size (384x384).
   - Maintains original aspect ratios.

- Preserves relative spatial relationships.

4. **Validation Strategy**
   - During validation, no random cropping is applied.
   - Center point remains fixed for validation.
   - Allows fair comparison with ground truth.

8.4 Random Scaling and Cropping Example

Original Image:

- Size: 800 x 800.
- LiDAR measurement point: (400, 400) [exact center].

Random Crop:

- Crop size: 500 x 500 (randomly chosen between 384 and 800).
- Top-left position: (220, 150) [randomly chosen to fit the 500x500 crop].

Step 1: Crop Transformation

Original center: (400, 400)
Crop offset: (220, 150)
New center = Original center - Crop offset
New center = (400-220, 400-150) = (180, 250)

In the 500x500 cropped image, the LiDAR point is now at (180, 250) - notably off-center.

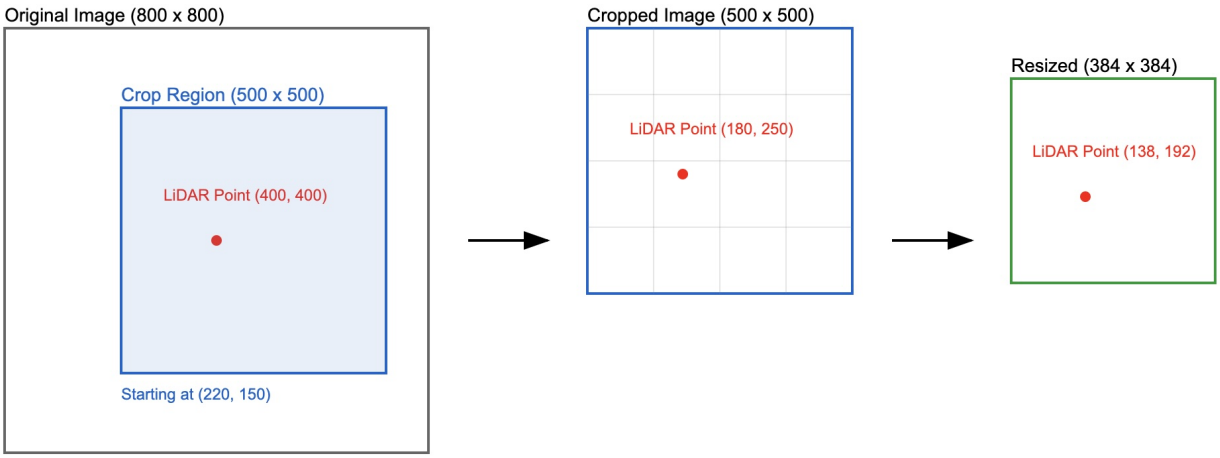

Figure 1: Random scaling and cropping example

Step 2: Resize to 384x384

Scale factor: $384/500 = 0.768$
Final center: $(180 \times 0.768, 250 \times 0.768)$
Final center coordinates: $(138, 192)$

Final Result:

- Image size: 384 x 384.
- LiDAR point: (138, 192).
- Original center point has moved significantly off-center:
  - Horizontally: shifted from center by 54 pixels (384/2 - 138 = 54).
  - Vertically: shifted from center by 0 pixels (384/2 - 192 = 0).

This asymmetric transformation illustrates how the cropping strategy creates variations in the LiDAR measurement position.

9 FlightMetadataManager Design

The FlightMetadataManager is designed to handle the complex array of sensor and flight data collected during NASA's ER-2 missions. Here's why different types of metadata are handled in specific ways:

9.1 Metadata Categorization

The metadata is categorized into distinct groups based on their physical meaning and units:

9.1.1 Geographic Parameters

- `Lat`, `Lon`: Geographic coordinates
- `GPS_MSL_Alt`, `WGS_84_Alt`, `Press_Alt`: Different altitude measurements

Motivation: These parameters require consistent handling as they're used for spatial positioning and often need to be used together for accurate location determination.

9.1.2 Aircraft Motion Parameters

- `Grnd_Spd`, `True_Airspeed`, `Mach_Number`: Speed measurements
- `Vert_Velocity`: Vertical movement
- `True_Hdg`, `Track`, `Drift`, `Pitch`, `Roll`: Aircraft orientation

Motivation: These parameters describe the aircraft's motion and orientation, which affect image capture conditions and need to be considered when analyzing cloud heights.

9.1.3 Environmental Parameters

- `Ambient_Temp`, `Total_Temp`: Temperature readings
- `Static_Press`, `Dynamic_Press`, `Cabin_Pressure`: Pressure measurements
- `Wind_Speed`, `Wind_Dir`: Wind conditions

Motivation: Environmental conditions can affect both the aircraft's performance and the cloud formation being studied.

9.1.4 Solar Parameters

- `Solar_Zenith`, `Sun_Elev_AC`, `Sun_Az_Grd`, `Sun_Az_AC`: Sun position relative to aircraft

Motivation: Helps in understanding lighting conditions which affect image quality and interpretation.

9.2 Special Handling Features

9.2.1 Temporal Feature Engineering

The system employs cyclic encoding for temporal features through trigonometric transformations:

$$\begin{aligned}
\text{hour}_{\text{sin}} &= \sin(2\pi \cdot \text{hour}/24) \\
\text{hour}_{\text{cos}} &= \cos(2\pi \cdot \text{hour}/24) \\
\text{day}_{\text{sin}} &= \sin(2\pi \cdot \text{day}/365) \\
\text{day}_{\text{cos}} &= \cos(2\pi \cdot \text{day}/365)
\end{aligned}$$

Motivation:

- Preserves cyclic nature of time (e.g., hour 23 is close to hour 0).
- Natural range of [-1, 1] eliminates the need for additional normalization.
- Smooth transitions between time periods.
- Better representation for neural network processing.

9.2.2 Normalization Strategy

The system applies different normalization strategies based on parameter type:

For standard parameters:

$$x_{\text{normalized}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

For angular parameters:

$$\theta_{\text{normalized}} = \begin{cases} \sin(\theta) & \text{for sine component} \\ \cos(\theta) & \text{for cosine component} \end{cases}$$

For pre-normalized parameters:

$$x_{\text{final}} = x_{\text{original}}$$

9.3 Parameter Organization

9.3.1 Feature Type Grouping

The system organizes parameters into hierarchical categories:

1. **Primary Features**
$$\mathbf{F}_{\text{primary}} = \{\mathbf{F}_{\text{flight}}, \mathbf{F}_{\text{temporal}}, \mathbf{F}_{\text{target}}\}$$
2. **Flight Data Subset**
$$\mathbf{F}_{\text{flight}} = \{\mathbf{P}_{\text{geographic}}, \mathbf{P}_{\text{motion}}, \mathbf{P}_{\text{environmental}}, \mathbf{P}_{\text{solar}}\}$$
3. **Parameter Vectors**
$$\mathbf{P}_{\text{geographic}} = [\text{Lat}, \text{Lon}, \text{Alt}_{\text{GPS}}, \text{Alt}_{\text{WGS}}, \text{Alt}_{\text{Press}}]$$
$$\mathbf{P}_{\text{motion}} = [\text{Speed}_{\text{ground}}, \text{Speed}_{\text{air}}, \text{Mach}, \text{Velocity}_{\text{vert}}, \dots]$$

9.3.2 Index Management

The system maintains a strict ordering system:

- **Base Index Structure:**
$$I(p) : \mathbf{F}_{\text{all}} \to \mathbb{N}_0$$
- **Parameter Access:**
$$p_i = I^{-1}(i) \text{ where } i \in [0, |\mathbf{F}_{\text{all}}| - 1]$$

This organization ensures:

- Consistent parameter ordering across batches.
- $\mathcal{O}(1)$ parameter lookup time.
- Maintainable structure for feature additions.
- Clear separation of concerns for different parameter types.

9.4 Key Benefits

1. **Data Integrity**
   - Consistent handling of related parameters.
   - Proper normalization based on parameter type.
   - Preservation of meaningful relationships between parameters.
2. **Model Performance**
   - Well-organized features for neural network input.
   - Properly scaled values for better training.
   - Efficient batch processing.
3. **Maintenance and Extensibility**
   - Clear structure for adding new parameters.
   - Easy modification of normalization strategies.
   - Simple parameter grouping updates.
4. **Error Prevention**
   - Type checking for parameters.
   - Validation of normalization requirements.
   - Consistent parameter ordering.

10 Training and Validation Split Design

10.1 Initial Dataset Creation

First, we create a full dataset without normalization:

$$D_{\text{full}} = \{S_1, S_2, ..., S_n\}$$

where each S_i is a sequence of frames.

10.2 Sequence-Based Splitting

Rather than splitting individual frames randomly, we split entire sequences:

$$
\begin{aligned}
&I_{\text{sequences}} = \{1, 2, ..., n\} \text{ where } n = |D_{\text{full}}| \\
&\text{Split } I_{\text{sequences}} \text{ into:} \\
&\quad I_{\text{train}} \text{ (80\% of sequences)} \\
&\quad I_{\text{val}} \text{ (20\% of sequences)}
\end{aligned}
$$

Motivation for Sequence-Based Splitting:

- Preserves temporal continuity within sequences.
- Prevents data leakage between train/validation.
- Maintains sequence integrity.

10.3 Row Extraction Process

For each set (train and validation):

$$
\begin{aligned}
&\text{For each sequence index } i: \\
&\quad L_i = \text{sequence\_length}(i) \\
&\quad R_i = \{i, i+1, ..., i+L_i-1\} \\
&R_{\text{train}} = \bigcup_{i \in I_{\text{train}}} R_i \\
&R_{\text{val}} = \bigcup_{i \in I_{\text{val}}} R_i
\end{aligned}
$$

10.4 Dataset Creation

10.4.1 Training Dataset

$D_{\text{train}} = \text{Cloud2CloudDataset}(R_{\text{train}})$ with:

- apply_normalization = True
- apply_crop_and_scale = True
- Calculate normalization parameters:

  For each feature f:

  $\mu_f = \text{mean}(f)$

  $\sigma_f = \text{std}(f)$

10.4.2 Validation Dataset

$D_{\text{val}} = \text{Cloud2CloudDataset}(R_{\text{val}})$ with:

- apply_normalization = True
- apply_crop_and_scale = False
- Use training normalization parameters:

$$f_{\text{normalized}} = \frac{f - \mu_f}{\sigma_f}$$

10.5 Key Differences Between Train and Validation

Training Set Features:

- Random cropping and scaling.
- Data augmentation.
- Shuffled sequences.
- Calculates normalization parameters.

Validation Set Features:

- No random cropping (center crop only).
- Limited augmentation.
- Sequential processing.
- Uses training set's normalization parameters.

Note: We use training-only normalization parameters

This is an important machine learning practice for several reasons:

1. **Preventing Data Leakage**
   - If we calculated normalization parameters using all data (including validation), we'd let information from the validation set influence our training data.
   - This would create a subtle form of data leakage where our model "peeks" at validation data statistics.
2. **Real-world Scenario Simulation**
   - In production, we'll need to normalize new, unseen data.
   - We won't be able to calculate new normalization parameters for this data.
   - Using training-only parameters better simulates this real-world scenario.
3. **Validation Integrity**
   - Validation should measure how well our model generalizes.
   - Using training-derived normalization parameters tests if our normalization approach itself generalizes.
   - If validation performance is good, it suggests our normalization is robust.

Mathematically, for any feature f:

Training:

$$\mu_{\text{train}} = \text{mean}(f_{\text{train}})$$

$$\sigma_{\text{train}} = \text{std}(f_{\text{train}})$$

$$f_{\text{train\_normalized}} = \frac{f_{\text{train}} - \mu_{\text{train}}}{\sigma_{\text{train}}}$$

Validation:

$$f_{\text{val\_normalized}} = \frac{f_{\text{val}} - \mu_{\text{train}}}{\sigma_{\text{train}}} \quad \text{(Using training parameters)}$$

This ensures that our validation metrics genuinely reflect how well our model will perform on unseen data.

10.6 DataLoader Configuration

$\text{Loader}_{\text{train}} = \text{DataLoader}(D_{\text{train}})$ with:

- batch_size = 1
- shuffle = True
- custom collate function

$\text{Loader}_{\text{val}} = \text{DataLoader}(D_{\text{val}})$ with:

- batch_size = 1
- shuffle = False
- same collate function

At this stage, the dataloader using only a batch size of 1 is useful for:

- Visualizing individual sequences.
- Debugging data transformations.
- Checking center point calculations.
- Verifying normalization.

11 CloudMotionModel Class

The `CloudMotionModel` is a dual-RAFT architecture designed for cloud motion estimation that combines a trainable RAFT model with a frozen reference model.

11.1 Model Components

11.1.1 1. Dual RAFT Models

The architecture employs two RAFT instances:

- **Trainable RAFT**: Fine-tuned for cloud-specific motion.
- **Frozen Reference RAFT**: Maintains baseline motion estimates.

```
self.raft = raft_large(weights=Raft_Large_Weights.DEFAULT if use_pretrained else None)
self.reference_raft = raft_large(weights=Raft_Large_Weights.DEFAULT)  # Frozen
```

11.1.2 2. Bidirectional Anomaly Detection and Preservation

The `preserve_bidirectional_anomalies` method addresses a key challenge in cloud motion sequences: temporal inconsistency in feature detection. Cloud features, particularly at high altitudes, may be detected strongly in some frame pairs but weakly or not at all in others.

The model identifies motion anomalies using statistical thresholds:

$$\text{strong\_anomalies} = \text{motion} > (\mu + \sigma \cdot \text{threshold})$$
$$\text{weak\_anomalies} = \text{motion} < (\mu - \sigma \cdot \text{threshold})$$

where:

- μ is the mean motion across the sequence (calculated per pixel).
- σ is the standard deviation of motion.
- `threshold` is the anomaly detection parameter (default 2.0).

For each pixel location (x, y), the final motion is determined by:

$$\text{final\_motion}(x, y) = \begin{cases} \max(\text{motion}(x, y)) & \text{if strong anomaly detected at any time} \\ \min(\text{motion}(x, y)) & \text{if weak anomaly detected at any time} \\ \mu(\text{motion}(x, y)) & \text{otherwise} \end{cases}$$

This preservation mechanism is helpful because cloud motion features in a sequence $\{I_1, I_2, ..., I_T\}$ may only be distinctly visible in specific frame pairs. For example, a high-altitude cloud feature might show significant motion between frames I_1 and I_2, but become less distinct in subsequent pairs due to:

- Changes in viewing angle.
- Varying illumination conditions.
- Temporary occlusions by other cloud layers.
- Varying contrast against the background.

By preserving both strong and weak anomalies across the entire sequence, we ensure that important motion features are retained even if they're only detected in a subset of frame pairs. This approach improves the robustness of height estimation by:

- Preserving rare but significant motion features.
- Reducing the impact of temporary feature occlusions.
- Maintaining consistency in height estimates for high-altitude clouds.
- Combining evidence of cloud motion across multiple temporal samples.

11.1.3 3. Confidence Estimation

Confidence scores are computed using motion magnitude:

$$\text{confidence} = \sigma(\|\text{refined\_motion}\|_2)$$

where σ is the sigmoid function to normalize confidence to $[0, 1]$.

11.2 Forward Pass Flow

1. Sequence Processing

- **Input**: Sequence of images $\{I_1, I_2, ..., I_T\}$.
- For each consecutive pair (I_t, I_{t+1}):
  - Scale images to $[0, 255]$ range.
  - Process through both RAFT models.
  - Store motion fields.

2. Motion Stack Creation

- Stack motion fields temporally.
- Handle packed sequences if present.
- Motion dimensions: $(B, T-1, 2, H, W)$, where B is batch size, T is sequence length, H is height, and W is width.

3. Motion Refinement

- Apply bidirectional anomaly preservation.
- Generate confidence maps.
- Handle packed sequence conversions.

4. Output Generation

```
return {
    'motion_fields': Original motion sequence
    'refined_motion': Anomaly-preserved motion
    'reference_motion_fields': Reference RAFT sequence
    'reference_motion': Reference refined motion
    'confidence': Confidence maps
}
```

12 ParallaxHeightCalculator Class

The `ParallaxHeightCalculator` implements the parallax principle to convert motion fields into cloud heights using a combination of LiDAR calibration and statistical methods.

12.1 Scale Management

The calculator maintains three critical scales that transform raw pixel motion into meaningful physical cloud heights. Each scale solves a specific challenge in the height estimation process:

Motion Scale Maps pixel motion to physical distance:

- Raw optical flow gives motion in pixels.
- Converts to real-world distances (meters) using LiDAR calibration.
- High-altitude clouds show less pixel motion, biasing the scale towards larger values to detect high clouds.

Height Scale Converts relative motion to absolute height:

- Parallax effect creates relative motion proportional to the height ratio.
- Converts ($h_{\text{cloud}}/h_{\text{aircraft}}$) ratio to absolute height.
- Calibrated using LiDAR measurements, updates slowly for stability.

Vertical Extent Scale Controls the vertical range of cloud features:

- Prevents compression/stretching of vertical structure.
- Scales cloud thickness to 30% of cloud-top height or 3000m.
- Preserves realistic cloud vertical development.

Each scale uses asymmetric smoothing with bias towards larger values:

$$\text{scale}_{\text{new}} = (1 - \alpha)\text{scale}_{\text{old}} + \alpha(\text{scale}_{\text{update}})$$

where α is:

$$\begin{cases} 0.05 & \text{if scale}_{\text{update}} > \text{scale}_{\text{old}} \\ 0.01 & \text{if scale}_{\text{update}} \leq \text{scale}_{\text{old}} \end{cases}$$

This asymmetric smoothing ensures stability in height calculations while preserving sensitivity to high-altitude clouds.

12.2 Scale Calculation

For each frame with LiDAR data:

1. **Height Ratio:**
$$\text{height\_ratio} = \frac{\text{lidar\_height}}{\text{aircraft\_altitude}}$$
2. **Expected Motion:**
$$\text{expected\_motion} = \text{aircraft\_speed} \times \text{height\_ratio}$$
3. **Motion Scale:**
$$\text{motion\_scale} = \frac{\text{expected\_motion}}{\text{center\_motion}}$$
4. **Vertical Extent Scale:**
$$\text{vertical\_extent} = \min\left(\frac{0.3 \times \text{lidar\_height}}{\text{current\_range}}, 3000\right)$$

12.3 Height Calculation

For each pixel (x, y) with valid motion:

1. **Relative Motion:**
$$\text{relative\_motion}_{x,y} = \frac{\text{motion\_magnitude}_{x,y} \times \text{motion\_scale}}{\text{aircraft\_speed}}$$
2. **Raw Height:**
$$\text{height}_{x,y} = \text{aircraft\_altitude} \times \text{relative\_motion}_{x,y} \times \text{height\_scale}$$
3. **Confidence-weighted Smoothing:**
$$\text{height}_{x,y} = c \times \text{height}_{x,y} + (1 - c) \times \text{smooth}(\text{height}_{x,y})$$
where c is the confidence value.

12.4 Height Field Adjustment

When LiDAR calibration is used:

1. Center height subtraction:
$$h' = h - h_{\text{center}}$$
2. Vertical extent scaling:
$$h'' = h' \times \text{vertical\_extent\_scale}$$
3. LiDAR height addition:
$$h_{\text{final}} = h'' + h_{\text{lidar}}$$

12.5 Confidence Calculation

Combines motion magnitude and local consistency:

1. **Base Confidence:**
$$c_1 = 1 - e^{-\text{motion}/\text{threshold}}$$
2. **Local Consistency:**
$$c_2 = 1 - \frac{|\text{motion} - \text{smooth}(\text{motion})|}{\text{smooth}(\text{motion})}$$
3. **Final Confidence:**
$$c = c_1 \times c_2$$

12.6 Robust Scale Updates

Uses Median Absolute Deviation (MAD) for outlier rejection:

$$\text{MAD} = \text{median}(|\text{scales} - \text{median}(\text{scales})|)$$

Inliers are defined as:

$$|\text{scale} - \text{median}(\text{scales})| < 5.0 \times \text{MAD}$$

This ensures stable height estimation even with noisy motion fields and varying aircraft conditions, while the asymmetric smoothing preserves sensitivity to high-altitude cloud features.

13 CombinedModel Class

The `CombinedModel` integrates the `CloudMotionModel` with the `ParallaxHeightCalculator` to create a complete cloud-height measurement pipeline.

13.1 Motion Estimation

Employs dual RAFT instances from `CloudMotionModel`:

- Trainable RAFT for cloud-specific motion.
- Frozen reference RAFT for baseline comparison.
- Both produce motion fields $M \in \mathbb{R}^{B\times T\times 2\times H\times W}$.

13.2 Height Calculation

Maintains two parallel height calculators:

```
self.finetuned_height_calculator = ParallaxHeightCalculator(
    smoothing_factor=0.1,
    history_size=1000,
    use_lidar_to_calculate_heights=True
)
self.reference_height_calculator = ParallaxHeightCalculator(...)
```

This allows direct comparison between fine-tuned and baseline performance.

13.3 Forward Pass Pipeline

1. Motion Field Generation

- **Input sequence:** $I \in \mathbb{R}^{B\times T\times 3\times H\times W}$.
- Generate motion fields from both RAFT models.
- Preserve bidirectional anomalies.
- Calculate motion confidence.

2. Height Field Generation When metadata and LiDAR are available:

$$h_{\text{ft}} = \text{finetuned\_calculator}(M_{\text{ft}}, \text{metadata}, h_{\text{lidar}})$$
$$h_{\text{ref}} = \text{reference\_calculator}(M_{\text{ref}}, \text{metadata}, h_{\text{lidar}})$$

13.4 Output Structure

```
{
    'motion_fields': Sequence of motion fields
    'refined_motion': Final motion with preserved anomalies
    'reference_motion': Reference model motion
    'height_field_finetuned': Height estimates from fine-tuned model
    'height_uncertainty_finetuned': Uncertainty estimates
    'height_field_reference': Reference height estimates
    'height_uncertainty_reference': Reference uncertainty
    'calibration_params_finetuned': Current scale parameters
    'calibration_params_reference': Reference scale parameters
}
```

13.5 Scale Management

Both height calculators maintain independent running scales:

- Motion scale: s_m for pixel-to-meter conversion.
- Height scale: s_h for relative-to-absolute height.
- Vertical extent scale: s_v for cloud thickness.

These scales are updated and saved in checkpoints for model continuity.

13.6 State Management

```
def get_height_calculator_states(self):
    """Returns current states of both calculators."""
    return {
        'finetuned': {
            'running_motion_scale',
            'running_height_scale',
            'running_vertical_extent_scale',
            'scale_histories'
        },
        'reference': {...}
    }
```

13.7 Model Hierarchy

```
CombinedModel
├── CloudMotionModel
│   ├── Trainable RAFT
│   └── Reference RAFT
├── Fine-tuned Height Calculator
│   └── Scale Management
└── Reference Height Calculator
    └── Scale Management
```

13.8 Features and Capabilities

This architecture enables:

- Direct comparison of fine-tuned vs baseline performance.
- Height estimation with uncertainty quantification.

- Scale preservation across training sessions.
- Comprehensive motion and height field visualization.

14 SequenceConsistencyLoss Class

The `SequenceConsistencyLoss` enforces temporal coherence and motion consistency in cloud height estimation through three complementary loss components.

14.1 Photometric Loss

Photometric loss ensures accurate frame reconstruction by comparing how well the predicted motion fields can reconstruct subsequent frames. This enforces temporal consistency in the motion estimates.

$$L_{\text{photo}} = \sum_{t=1}^{T-1} \|I_{t+1} - \text{warp}(I_t, F_t)\| \cdot V_t$$

where:

- I_t is frame t in the sequence.
- F_t is the estimated motion field.
- V_t is a visibility mask.
- `warp()` applies the motion field to warp frame t.

Why this is important for cloud motion:

- Clouds undergo continuous, gradual changes in appearance and shape.
- At high altitudes, cloud features must maintain temporal consistency to enable accurate tracking.
- The visibility mask component handles complex scenarios like overlapping or occluding clouds.
- This loss helps distinguish between actual cloud movement and changes caused by varying illumination conditions or aircraft motion.

14.2 Smoothness Loss

Smoothness loss penalizes rapid spatial variations in the motion field, promoting locally coherent motion predictions. This helps avoid spurious motion estimates and encourages physically plausible cloud motion patterns.

$$L_{\text{smooth}} = \sum_{t=1}^{T-1} \|\nabla F_t\|$$

where ∇F_t represents spatial gradients of the motion field.

Why this is important for cloud motion:

- Cloud formations exhibit fluid-like behavior with coherent motion patterns.
- At high altitudes, clouds tend to move more uniformly.
- This loss ensures motion fields reflect the continuous flow of cloud systems.
- It prevents physically impossible motion discontinuities that could lead to height estimation errors.

14.3 Flow Consistency Loss

Flow consistency loss ensures bidirectional consistency between forward and backward motion estimates. This helps identify and penalize invalid motion predictions that are not consistent when applied in both temporal directions.

$$L_{\text{consist}} = \sum_{t=1}^{T-1} \|F_t + \text{warp}(F'_t, F_t)\|$$

where F'_t is the backward flow.

Why this is important for cloud motion:

- True cloud movement should be temporally reversible over short time scales.
- Differentiates between actual cloud motion and apparent motion caused by changes in cloud shape or illumination.
- Bidirectional verification ensures reliable height estimates across varying viewing conditions.

14.4 Combined Loss

The final loss combines these components with configurable weights:

$$L_{\text{total}} = L_{\text{photo}} + \lambda_{\text{smooth}} L_{\text{smooth}} + \lambda_{\text{consist}} L_{\text{consist}}$$

where λ_{smooth} and λ_{consist} are weighting factors.

14.5 Implementation Features

- Uses PyTorch's packed sequence functionality to efficiently handle variable-length sequences.
- Implements visibility masking to exclude occluded regions from loss computation.
- Provides bidirectional consistency checking through forward and backward flow comparison.
- Applies spatial regularization through smoothness constraints.
- Handles batch processing for efficient training.

14.6 Guided Learning Goals

The combined loss terms guide the model to learn motion fields that:

1. Accurately reconstruct subsequent frames.
2. Maintain spatial smoothness.
3. Exhibit bidirectional consistency.
4. Account for occlusions through visibility masking.

15 CombinedLoss Architecture

The `CombinedLoss` combines motion consistency with LiDAR supervision to enable accurate height field estimation.

15.1 Motion Loss Component

Uses `SequenceConsistencyLoss` to ensure temporal coherence in predicted motion fields:

$$L_{\text{motion}} = L_{\text{sequence}}(F_{\text{pred}}, I, l)$$

where:

- F_{pred} are predicted motion fields.
- I are input image sequences.
- l are sequence lengths.

15.2 LiDAR Supervision Component

While the model can predict motion patterns, converting these to absolute heights requires calibration. A single LiDAR measurement at the image center provides this reference point.

1. **Calculate motion at image center:**

$$m_{\text{center}} = \|F_{\text{pred}}(x_c, y_c)\|$$

2. **Convert to relative motion:**

$$r_{\text{actual}} = \frac{m_{\text{center}}}{v_{\text{aircraft}}}$$

3. **Calculate expected relative motion:**

$$r_{\text{expected}} = \frac{h_{\text{lidar}}}{h_{\text{aircraft}}}$$

4. **Determine scaling factor:**

$$s = \frac{r_{\text{expected}}}{r_{\text{actual}}}$$

5. **Apply to motion field:**

$$F_{\text{scaled}} = F_{\text{pred}} \cdot s$$

LiDAR Loss: The loss is:

$$L_{\text{lidar}} = L_{\text{smooth\_l1}}(F_{\text{scaled}}, F_{\text{pred}})$$

15.3 Total Loss

The total loss combines the motion and LiDAR components:

$$L_{\text{total}} = L_{\text{motion}} + w_{\text{lidar}} \cdot L_{\text{lidar}} \cdot \alpha$$

where:

- w_{lidar} is the LiDAR weight (default 0.01).
- $\alpha = \text{clamp}\left(\frac{L_{\text{motion}}}{L_{\text{lidar}}}, 0.01, 10\right)$ dynamically scales the LiDAR loss.

15.4 Height Error Monitoring

Tracks error between predicted and LiDAR heights:

$$E_{\text{height}} = \frac{1}{N} \sum_{i=1}^{N} |h_i - h_{\text{ref},i}|$$

For valid predictions where:

- h_i are predicted center heights.
- $h_{\text{ref},i}$ are LiDAR measurements.
- N is the number of valid samples.

16 Collation and Packing

16.1 Purpose of Collation

When batching sequences for training, the following challenges must be addressed:

1. Variable-length sequences (2-5 frames).
2. Missing/invalid data.
3. Multiple aligned data streams (images, flight data, timestamps).
4. Memory efficiency.
5. RNN processing requirements.

16.2 Initial Data Unpacking

```
Input Batch Structure:
- image_sequences: (B × T × C × H × W)
- flight_data_list: (B × T × D_flight)
- validation_heights: (B × 1)
- resized_center_coords: (B × 2)
- image_paths: (B × T)
- timestamp_sequences: (B × T)
```

16.3 Sequence Validation and Truncation

For each sequence:

$$\begin{aligned}
&\text{Valid Index Set} = \{i : \text{GPS_MSL_Alt}_i \neq \text{NaN}\} \\
&\text{last_valid_idx} = \max(\text{Valid Index Set}) \\
&\text{start_idx} = \max(0, \text{last_valid_idx} - \text{max_sequence_length} + 1) \\
&\text{sequence_length} = \min(\text{end_idx} - \text{start_idx}, \text{max_sequence_length})
\end{aligned}$$

16.4 Sequence Length Requirements

Constraints:

$$2 \leq \text{sequence\_length} \leq 5$$

- A minimum of 2 frames is required for temporal patterns.
- A maximum of 5 frames ensures computational efficiency.
- Valid GPS altitude data is essential.

16.5 Sequence Sorting and Packing

Why Pack Sequences?

1. **Memory Efficiency:** Avoids padding all sequences to the maximum length, storing only actual data points.
2. **Computational Efficiency:** RNN computations focus only on valid timesteps, avoiding padding tokens.
3. **Better Gradient Flow:** Prevents gradients from propagating through padding, improving training stability.

Packing Process:

1. Sort sequences by length (descending):

$$\text{sequences} = \text{sort}(\text{sequences}, \text{key} = \text{len}, \text{reverse} = \text{True})$$

2. Pack sequences:

```
PackedSequence Structure:
- data: All sequence elements concatenated
- batch_sizes: Number of sequences at each timestep
- sorted_indices: Original sorting order
- unsorted_indices: Restore original order
```

Example:

```
Original Sequences:
Seq1: [A1, A2, A3]
Seq2: [B1, B2]
Seq3: [C1, C2, C3, C4]

Sorted:
C1, C2, C3, C4
A1, A2, A3, --
B1, B2, --, --

Packed:
data: [C1, A1, B1, C2, A2, B2, C3, A3, C4]
batch_sizes: [3, 3, 2, 1]
```

16.6 Final Output Structure

```
{
    'images': PackedSequence,      # Packed image sequences
    'metadata': PackedSequence,    # Packed flight data
    'validation_height': Tensor,   # Single height per sequence
    'center_coords': Tensor,       # Coordinate pairs
    'sequence_lengths': Tensor,    # Length of each sequence
    'image_paths': List,           # For debugging/visualization
    'timestamps': List             # Temporal alignment
}
```

16.7 Sequence Truncation Strategy

Two main issues with sequence lengths:

1. **Memory Issues:**
   - Long sequences (5+ frames) cause Out-of-Memory (OOM) errors.
   - Each image is $384 \times 384 \times 3$, consuming significant GPU memory when batched.
2. **Data Quality Issues:**
   - Abnormally long sequences (1000+ frames) indicate data collection or synchronization problems.
   - Causes include:
     - Missing LiDAR measurements.
     - GPS data gaps.
     - Timestamp misalignment.

16.7.1 Truncation Implementation

1. Valid Data Detection:

```
For each sequence:
1. Check GPS_MSL_Alt for NaN values
2. Find last valid index where data exists
3. If no valid data found, skip sequence
```

2. Sequence Windowing:

```
start_idx = max(0, last_valid_idx - max_sequence_length + 1)
end_idx = last_valid_idx + 1
```

Key Points:

- Always keeps the most recent valid frames.
- Works backward from the last valid measurement.
- Ensures LiDAR measurement aligns with the final frame.

3. Length Constraints:

```
min_sequence_length = 2  # Minimum for temporal patterns
max_sequence_length = 5  # Maximum for memory management
```

Benefits:

- Prevents OOM errors.
- Ensures consistent batch sizes.
- Maintains temporal coherence.
- Filters out problematic sequences.

16.8 Example Scenarios

1. Normal Case:

```
Original: [F1, F2, F3, F4, F5, F6, F7, F8]  // 8 frames
Last valid: F8
Truncated: [F4, F5, F6, F7, F8]  // 5 frames
```

2. Bad Data Case:

```
Original: [F1, F2, ..., F1000]  // 1000 frames
Last valid: F1000
Truncated: [F996, F997, F998, F999, F1000]  // 5 frames
```

3. Short Sequence:

```
Original: [F1, F2, F3]  // 3 frames
Last valid: F3
Result: [F1, F2, F3]  // Kept as is
```

4. Invalid Sequence:

```
Original: [F1(NaN), F2(NaN), F3(NaN)]  // All invalid
Result: Sequence discarded
```

Effectiveness: This truncation strategy effectively:

- Manages memory usage.
- Maintains data quality.
- Preserves temporal relationships.
- Handles edge cases gracefully.

17 Training Loop Architecture

The training loop operates in epochs, with each epoch processing the full dataset. Here's what happens in each epoch:

17.1 Batch Processing

- Loads a batch of image sequences with metadata.
- Skips invalid batches.
- Moves data to GPU if available.

17.2 Model Forward Pass

- Processes images through the RAFT model.
- Computes motion fields and height estimates.
- Uses automatic mixed precision for efficiency.

17.3 Loss Calculation

- Computes combined loss, including:
  - Motion consistency.
  - LiDAR supervision.
  - Height estimation error.
- Checks for NaN/Inf values.

17.4 Optimization Step

- Computes gradients.
- Clips gradients at 0.5 norm.
- Updates model parameters.
- Cleans GPU memory.

17.5 Monitoring

- Tracks multiple loss components.
- Prints progress every 20 batches.
- Visualizes first batch results.
- Logs metrics and learning rate.

17.6 End of Epoch

- Computes average losses.
- Updates learning rate based on performance.
- Creates visualization plots.
- Saves a checkpoint if the loss improved.

The loop includes error handling for out-of-memory issues and careful GPU memory management throughout due to RAFT's voracious memory appetite.

17.7 Model Setup

$$\begin{aligned} \text{Model} &= \text{CombinedModel}(\text{RAFT}) \\ \text{Loss} &= \text{CombinedLoss}(\lambda_{\text{smooth}}) \\ \text{Optimizer} &= \text{AdamW}(\text{lr} = \alpha) \\ \text{Scheduler} &= \text{ReduceLROnPlateau}(\text{factor} = 0.5, \text{patience} = 5) \end{aligned}$$

17.8 Per-Epoch Training

For each epoch e:

1. **Forward Pass:**

$$\begin{aligned} \text{outputs} &= \text{Model}(I, M, h_{\text{lidar}}, l) \\ L, L_{\text{dict}} &= \text{Loss}(\text{outputs}, I, l, h_{\text{lidar}}, M) \end{aligned}$$

   where:

   - I: image sequence.
   - M: metadata.
   - h_{lidar}: validation height.
   - l: sequence lengths.

2. **Backward Pass:**

$$\begin{aligned} \nabla L &= \text{backward}(L) \\ \|\nabla\| &\leq 0.5 \text{ (gradient clipping)} \end{aligned}$$

3. **Loss Tracking:**

$$\bar{L}_e = \frac{1}{N}\sum_{i=1}^{N} L_i$$

4. **Learning Rate Update:**

$$\alpha_{e+1} = \begin{cases} 0.5\alpha_e & \text{if no improvement for 5 epochs} \\ \alpha_e & \text{otherwise} \end{cases}$$

17.9 Checkpointing

Saves the model when:

$$L_e < \min_{i<e} L_i \text{ or } e \bmod f = 0$$

where f is the save frequency.

17.10 Features

- Automatic mixed precision training.
- GPU memory management.
- Gradient anomaly detection.
- Visualization every epoch.
- Progress logging.

18 Training Progress Visualization

This function creates visualizations during model training to monitor both optical flow estimation and height field generation.

18.1 Motion Field Visualization

$$\begin{aligned} &\text{Input : } F_{\text{refined}}, F_{\text{reference}} \in \mathbb{R}^{2\times H\times W} \\ &\text{Output : 3-panel comparison showing:} \\ &\quad \text{– Original image sequence} \\ &\quad \text{– Frame-by-frame motion fields} \\ &\quad \text{– Refined vs Original RAFT comparison} \end{aligned}$$

18.2 Height Field Generation

For valid LiDAR measurements:

$$\begin{aligned} h_{\text{center}} &= \text{denormalize}(h_{\text{lidar}}) \\ h_{\text{refined}} &= \text{calculate\_heights}(F_{\text{refined}}, \text{metadata}, h_{\text{center}}) \\ h_{\text{reference}} &= \text{calculate\_heights}(F_{\text{reference}}, \text{metadata}, h_{\text{center}}) \end{aligned}$$

18.3 Calibration Analysis

$$\begin{aligned} &\text{Parameters tracked:} \\ &\qquad s_{\text{motion}} = \text{motion scale factor} \\ &\qquad s_{\text{height}} = \text{height scale factor} \\ &\qquad s_{\text{vertical}} = \text{vertical extent scale} \end{aligned}$$

18.4 Diagnostic Metrics

For each model (refined and reference):

- Height field range.
- Valid data percentage.
- Motion magnitude range.
- Center height accuracy.
- Calibration parameters.

18.5 Additional Features

The function handles:

- PackedSequence unpacking.
- Tensor normalization.
- GPU memory management.
- Exception handling.

19 Stitching Process

19.1 Data Collection and Preprocessing

The `collect_height_fields` function processes sequences to obtain:

- Height fields
- Confidence maps
- Flight metadata

A constraint of `collect_height_fields` is that it requires temporal ordering. To stitch height fields together, sequences must maintain their chronological order.

By setting `batch_size=1`:

- Each sequence stays in chronological order.
- Motion evolution can be tracked over time.
- Height fields align with their timestamps.

Tradeoffs:

- **Pros:** Simpler code, preserved temporal order.
- **Cons:** Must process sequences one at a time, slower execution.

```
# Creates batch_size=1 loader to maintain temporal order
stitch_loader = DataLoader(
    dataset,
    batch_size=1,  # Needed for temporal ordering
    shuffle=False,
    collate_fn=filter_collate_fn,
    num_workers=1,
    pin_memory=True
)
```

19.2 Global Coordinate System Creation

The `create_global_grid` function creates a unified coordinate system to stitch together multiple cloud height fields.

19.2.1 Haversine Distance

The haversine formula calculates great-circle distances between latitude/longitude points on a sphere (Earth):

$$a = \sin^2\left(\frac{\Delta\phi}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\Delta\lambda}{2}\right)$$
$$d = 2R\arcsin(\sqrt{a})$$

where:

- ϕ is latitude
- λ is longitude
- R is Earth's radius (6371km)

19.2.2 Grid Creation Process

1. Find Spatial Bounds:

For each height field H_i :

$$dx_i = \text{haversine}(\text{ref}_{\text{lat}}, \text{ref}_{\text{lon}}, \text{ref}_{\text{lat}}, \text{lon}_i)$$
$$dy_i = \text{haversine}(\text{ref}_{\text{lat}}, \text{ref}_{\text{lon}}, \text{lat}_i, \text{ref}_{\text{lon}})$$
$$\text{pos}_i = (dx_i \cdot \text{scale}, dy_i \cdot \text{scale})$$

2. Calculate Grid Dimensions:

$$x_{\min} = \min_i(\text{pos}_i^x)$$
$$x_{\max} = \max_i(\text{pos}_i^x + w_i)$$
$$y_{\min} = \min_i(\text{pos}_i^y)$$
$$y_{\max} = \max_i(\text{pos}_i^y + h_i)$$

where w_i, h_i are height field dimensions.

3. Adjust Measurement Points: For each point p:

$$p_{\text{adj}}^x = p^x - x_{\min}$$
$$p_{\text{adj}}^y = p^y - y_{\min}$$

The `create_global_grid` function returns:

- Grid dimensions
- Adjusted positions of all height fields
- Measurement point locations
- Reference coordinates and scale
- Relative positions for stitching

19.3 Field Stitching

The `stitch_height_fields` function combines fields using weighted averaging.

19.3.1 Create Output Arrays

The process creates arrays to track weighted height sums and sum of weights:

$$H_{\text{sum}} \in \mathbb{R}^{h \times w}$$
$$W_{\text{sum}} \in \mathbb{R}^{h \times w}$$

19.3.2 Generate Confidence Mask

Confidence masks use a Gaussian falloff from measurement points:

$$C(x, y) = 0.2 + 0.6 \exp\left(-\frac{d^2}{2\sigma^2}\right)$$

where d is the distance from measurement points.

Design Choice:

- Minimum confidence: 0.2 far from measurements.
- Maximum confidence: 0.8 at measurement points.
- Dynamic range: 0.6 allows smooth blending.

19.3.3 Add Weighted Contributions

Each height field contributes to the global grid:

$$H_{\text{sum}}(x, y) += h(x, y) \cdot w(x, y) \cdot c(x, y)$$
$$W_{\text{sum}}(x, y) += w(x, y) \cdot c(x, y)$$

19.3.4 Calculate Final Heights

$$H_{\text{final}}(x, y) = \frac{H_{\text{sum}}(x, y)}{W_{\text{sum}}(x, y)}$$

19.4 Post-Processing

After stitching, the system:

- Detects and smooths discontinuities with Gaussian filters.
- Preserves strong gradients in high-confidence regions.
- Creates an uncertainty visualization overlay.

The `stitch_height_fields` function produces:

- A stitched height field.
- A comprehensive confidence map.
- Metadata including sequences, measurement points, bounds, and reference coordinates.

19.5 Visualization

Two complementary visualizations:

1. `visualize_stitching_steps`:
    - 2D overview.
    - LiDAR measurement points.
    - Data validity percentages.
    - Height ranges.
2. `visualize_stitched_height_field_3d`:
    - Interactive 3D surface.
    - Uncertainty overlay.
    - Synchronized views.
    - Diagnostic statistics.