

École polytechnique de Louvain

Deep learning based Scene Flow estimation with point clouds

Author: **Louis THIRIFAY**

Supervisor: **Pierre SCHAUS**

Readers: **Achille MORENVILLE, Christophe DE VLEESCHOUWER**

Academic year 2023–2024

Master [120] in Computer Science

Abstract

The Scene flow, defined as ‘the three-dimensional motion field of points in the world’, is an essential component for mobile robotics and autonomous vehicles, capturing the motion of dynamic objects in our surroundings and serving as a basis for higher-level scene understanding. With 3D sensors becoming more affordable, such as LiDARs and RGB-D cameras, and with the recent progress in deep learning, deep neural networks directly processing raw point clouds have been the subject of growing interest for several years now for scene understanding tasks.

However, such methods usually do not take into account the real-time nature of LiDAR point clouds, scanned at a frequency of 10Hz. They remove the ground points naïvely using a height threshold, potentially eliminating moving points, and generally do not make any distinction between the foreground and background, which are moving and static, respectively. The ego-motion of the car, i.e. the rotation and translation of the car between two time frames, is taken into account, hence not reflecting the true motion of points in the environment.

This thesis proposes a simple ground segmentation algorithm, relying on standard operations on point clouds, followed by the cancellation of the ego-motion using a registration method. A clustering-based classification of objects in static and moving classes is then presented, further eliminating static points. Finally, a network hierarchically predicting the flows of moving points is proposed, exploring a new feature learning and downsampling method.

Acknowledgement

I would like to thank Achille MORENVILLE for his supervision, valuable advice and support throughout the year. I would also like to thank my family and my friends who supported me during this journey.

Computational resources have been provided by the supercomputing facilities of the Université catholique de Louvain (CISM/UCL) and the Consortium des Équipements de Calcul Intensif en Fédération Wallonie Bruxelles (CÉCI) funded by the Fond de la Recherche Scientifique de Belgique (F.R.S.-FNRS) under convention 2.5020.11 and by the Walloon Region.

Louis THIRIFAY

Contents

Introduction	1
1 Background and Related Work	3
1 Deep hierarchical feature learning	3
2 Point information fusion and matching	6
3 Flow generation and refinement	7
4 Model complexities and inference time	10
2 KITTI Vision Benchmark Suite	11
1 General Description	11
2 KITTI Scene flow (2015)	12
3 Generating point clouds from images	12
3.1 Pinhole camera model	13
3.2 Intrinsic parameters	13
3.3 Extrinsic parameters	14
3.4 Point clouds lifting	15
4 Ego-motion compensation	17
4.1 Rigid-body transformations	17
4.2 Mercator's projection	18
3 Ground segmentation and Ego-motion refinement	20
1 Ground segmentation	20
1.1 Voxel-grid filtering	21
1.2 K-nearest neighbors search	21
1.3 Normal estimation	22
1.4 Classification	23
2 Ego-motion refinement	24
2.1 ICP	25
2.2 Point-to-Plane ICP	25
4 Moving object segmentation	27
1 DBSCAN	27
2 Cluster association	29
2.1 Point cloud descriptor	29
2.2 Comparing histograms	31
2.3 Association process	31
3 Classification	32

5	Scene flow estimation	34
1	Feature learning and downsampling	34
2	Predicting the scene flow	35
3	PSNet	35
4	FH-Net	36
5	The final network	38
6	Experiments and Results	40
1	Ground segmentation	40
1.1	Evaluation metrics	40
1.2	Results	41
2	Ego-motion refinement	44
2.1	Evaluation metrics	44
2.2	Results	44
3	Moving object segmentation	46
3.1	Clustering metrics	46
3.2	Association metrics	47
3.3	Classification metrics	48
3.4	Results	48
4	Scene Flow prediction	52
4.1	Dataset	52
4.2	Loss function	52
4.3	Training and Evaluation	53
4.4	Runtime and scalability	57
7	Discussion and Further improvements	59
	Conclusion	61
	Appendices	68
1	Background and Related Work	69
1.1	Training and loss functions	69
1.2	Model complexities and inference time	70
2	Moving object segmentation	70
3	FH-Net implementation	71

Introduction

The 3D Scene Flow is a crucial component in many computer vision tasks nowadays. It can serve as a basis for motion segmentation where the discontinuities in the scene flow can help segment the points in objects with different motion fields. The scene flow can be used in the odometry task, where the goal is to compute the ego-motion, i.e. the rotation and translation of the car, between two time instants. Another application where the scene flow is useful concerns the object tracking task, where it can be used to enhance the prediction of motion and compute the object-level movements, effectively tracking them. The term scene flow originally appeared in 1999 and is defined as ‘the three-dimensional motion field of points in the world ’ [55]. It is the 3D counterpart of the optical flow, which aims at predicting the two-dimensional motion field of pixels in an image. Understanding the ever-changing and dynamic environment is crucial for applications such as mobile robotics or autonomous vehicles. As humans, we are able to perceive the motion of objects in our surroundings and are capable to plan and react accordingly. Capturing the motion of dynamic objects is thus also desirable for computer vision tasks and essential for higher-level scene understanding, where it can provide visual cues [37].

More formally, consider a source point cloud $S = \{x_i \in \mathbb{R}^3\}_{i=1}^N$ captured at instant t and a target point cloud $T = \{y_i \in \mathbb{R}^3\}_{i=1}^M$ captured at instant $t + 1$. The goal of scene flow estimation is to predict a motion field $F = \{f_i \in \mathbb{R}^3\}_{i=1}^N$ composed of individual motion vectors describing the movement of each point in S , which is illustrated in Fig. 1. The two point clouds do not necessarily contain the same number of points or have hard correspondences.

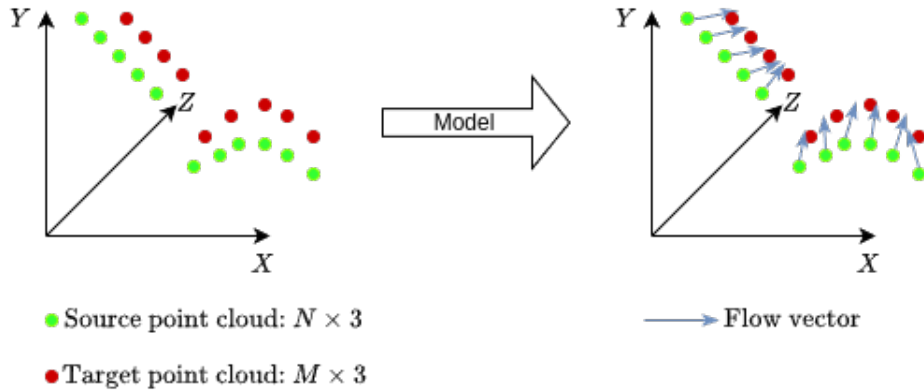


Figure 1: **Scene flow estimation** [39].

Multiple challenges can arise when solving the scene flow estimation problem [37]. First, The scene can be composed of both large and small displacements and both rigid and non-rigid objects can co-exist in the scene. Some objects can become partially or totally occluded, either entering the scene at the instant $t + 1$ or leaving it. Secondly, point clouds are by nature unstructured and unordered, and a difference in point density can be observed as the distance from the sensor increases, which can in turn hurt the feature extraction and search for correspondences. The scanning and reconstruction process can also introduce noise in the point cloud. Finally, point clouds are scanned at a frequency of 10Hz and the scene flow must be estimated at a higher frequency for real-time applications.

Many methods [61, 34, 28, 59, 57, 43, 35] do not make any distinction between the background and foreground points. However, only the foreground points that are moving give the most relevant information, as the background is always static. It is also worth noting that these methods naïvely remove the ground by discarding all points below a given height threshold, possibly eliminating moving foreground points along the way. By not making any distinction between the background and foreground points, the ego-motion of the car is taken into account in the prediction of the foreground points, hence not estimating their true motion. Generally speaking, not much accent is put on the complexity and inference time of the proposed models in the literature, rarely achieving real-time inference or most of the time not even discussing it. Currently, this is an area that is unfortunately still under-explored.

In this thesis, the ego-motion of the car is initially estimated using the car’s sensors. A ground segmentation algorithm based on classical operations on point clouds is then proposed to classify points as ground and non-ground instead of naïvely removing points below a certain threshold. A registration method refines the initially computed ego-motion, which is then canceled so that the car’s motion is not taken into account in the predicted flows. Next, the remaining non-ground points are clustered in objects and the clusters at both time instants are associated together and further classified as static or moving. In the end, only the moving points are fed to the network to predict their motion. Finally, to improve the efficiency and inference time of the network, another set abstraction for feature learning and downsampling is explored, not relying on commonly used and expensive operations. The set abstraction is plugged into a hierarchical network predicting flows at different resolutions.

Experiments were conducted on each part of the pipeline separately, with a focus on both the quantitative performance and the runtime. The proposed ground segmentation and ego-motion refinement achieve good performance while being suitable for real-time applications. The clustering followed by the association and classification of clusters further eliminates a consequent number of static points but is a bottleneck in terms of runtime. Finally, the network, although interesting in theory, fails to meet performance expectations due to his inability to train properly. However, Experiments showed significant improvement in terms of inference time.

Chapter 1

Background and Related Work

The concept of scene flow was first introduced in [55] as the three-dimensional field of motion vectors in the world, where they proposed a linear algorithm to compute a 3D scene flow from a 2D optical flow. Following this work, conventional methods using RGB and stereo images to predict optical flow and disparity/depth maps separately have been developed [37]. These image-based methods usually rely on standard variational formulations and energy minimization, with regularization for the smoothness of motion and assumption of rigidity of local objects. However, these methods are not directly optimized for 3D scene flow, yielding limited accuracy. With 3D sensors becoming more affordable, such as LiDARs and RGB-D cameras, and with the recent progress in deep learning, deep neural networks directly processing raw point clouds have been the subject of growing interest for several years now for scene understanding tasks [37].

1 Deep hierarchical feature learning

One of the foundation building blocks in deep learning and scene flow estimation is feature extraction and traditional convolutions cannot be used directly due to the irregular nature of point clouds. Pioneering works such as PointNet [44] and PointNet++ [45] set new standards for extracting deep hierarchical features on point sets, allowing new deep learning methods based on these works to be developed. As point clouds are unordered, the model needs to be invariant to permutation but also transformations such as rotation and translation. Multiple strategies exist among which a simple symmetric function to aggregate information from all the points can be used, taking n vectors as input and outputting a new vector invariant to the order [44]. PointNet [44] can be defined with the function $f : \mathbb{R}^{n \times d} \rightarrow \mathbb{R}^{d'}$, that maps a set of points $\{x_1, x_2, \dots, x_n\}$ with $x_i \in \mathbb{R}^d$ to a vector [45]:

$$f(x_1, x_2, \dots, x_n) = \gamma\left(\text{MAX}_{i=1, \dots, n}\{h(x_i)\}\right) \quad (1.1)$$

where γ and h are multi-layer perceptron (MLP) and MAX is the MAX pooling operator along the channel dimension. The function f is order invariant as the MAX function is symmetric. The idea is to learn the spatial encoding of each point through h and to aggregate the points' features through a pooling operation to give a global

descriptor for the point set. However, PointNet only extracts a global feature and fails at capturing local context at different scales as in CNNs. PointNet++ [45] resolves this limitation by introducing a hierarchical structure, with set abstraction levels, progressively learning features for larger and larger regions. A set abstraction is composed of 3 operations: sampling, grouping, and PointNet. The sampling layer uses farthest point sampling (FPS) to subsample the set of N points and pick N' centroids. The grouping layer builds a local region using ball query around each centroid, given a maximum number of neighbors K . Finally, PointNet aggregates the N' local regions of the centroids using Eq. 1.1, giving N' points with new features as the final output for a set abstraction level. As illustrated in Fig. 1.1, multiple set abstractions are chained one after another, each time decreasing the number of points while increasing the number of features and the receptive field of each point.

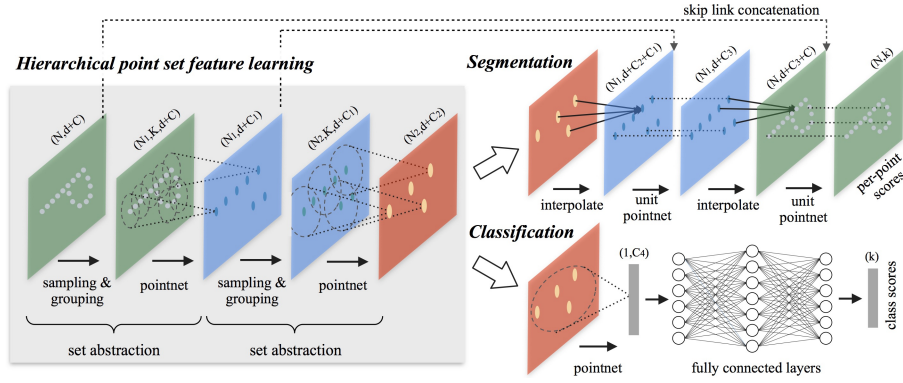


Figure 1.1: **Architecture of PointNet++ [45]**. Hierarchical feature learning with set abstractions (left) followed by task-dependent heads for segmentation and classification (right).

Following the PointNet++ [45] architecture, FlowNet3D [39], one of the first deep learning-based solutions for the scene flow estimation, proposes the `set_conv` layer, as illustrated in Fig. 1.2. It works exactly in the same manner as PointNet++ except that the MLP h of the aggregating function takes as additional input the spatial offset besides the neighboring features. The `set_conv` layer is used in many following works, including FlowStep3D [34], WhatMatters [56], LiDARSceneFlow [21]. HALFlow [57] uses `set_conv` with a random sampling at the first level to reduce resource consumption. The feature extractors of PV-RAFT [59] and FLOT [43] are based on PointNet++.

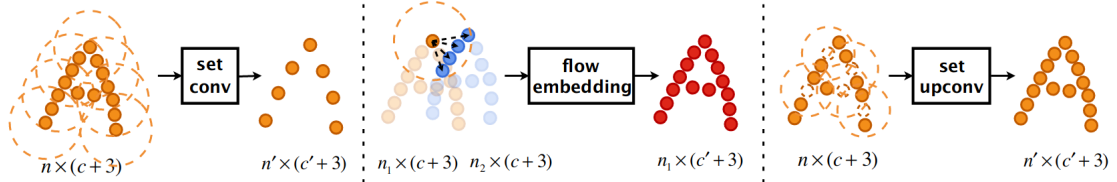


Figure 1.2: **FlowNet3D layers**. `Set_conv` layer (left) that learns deep features, `flow embedding` layer (middle) to encode point motions and `set_upconv` layer (right) to aggregate neighboring flow embeddings [39].

PointPWC-Net [61], on the other hand, employs PointConv [60] for feature learning. PointConv extends the traditional 2D convolutional operator and learns a convolution kernel to compute translation-invariant and permutation-invariant convolution on any point set. The convolutional weights are approximated by MLPs from the neighbor coordinates.

HPLFlowNet [28], inspired by BCL [33], proposes the DownBCL layer which operates on a discrete domain, similar to standard CNNs, and more precisely on a permutohedral lattice. The permutohedral convolution (BCL) is composed of three steps, as illustrated in Fig. 1.3. The splat operation distributes the points' features to adjacent lattice points based on their coordinates. The convolution is then performed on the permutohedral lattice. Similar to regular 2D convolution on images. Finally, the slice operation updates the features of the points with the corner lattice points of their simplex. DownBCL only uses the splat-conv steps, where the non-empty lattice points at the previous layer become the input points for the next layer, each time decreasing the scale and increasing the receptive field of the points.

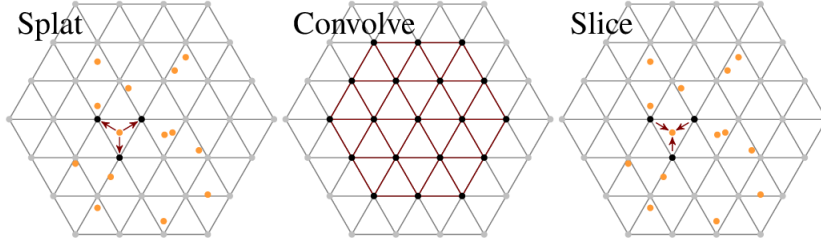


Figure 1.3: **BCL operation [33]**. Splat (left), Convolve (middle), and Slice (right) operations on the permutohedral lattice.

PointNet++ [45] is time-consuming as farthest point sampling (FPS) is a computationally expensive operation. By working on a discrete domain, HPLFlowNet [28] manages to reach a lower inference time. Another issue with FPS-based extraction methods is that they produce unstable downsampling due to randomness, tampering with the scene flow estimation.

More recent methods use the attention mechanism to cope with this issue or to learn more relevant features. FESTA [58] proposes the Spatial Abstraction with Attention layer (SA²), first using PointNet [44] to compute a global descriptor for each local region. For each point p_i in the local region, an attention weight w_i is computed using an MLP taking as input the global descriptor and the point's feature. The new synthesized point representing the group is $\sum_{i=1}^k w_i \cdot p_i$. Similarly, SCTN [35] introduces a Point Transformer based extraction. The attention weight $w_{i,j}$ for point p_i with respect to its neighboring point p_j is computed using both points' features and spatial encodings with MLPs. The neighboring features are then aggregated with a weighted sum. The advantage is that the attention weights can learn the importance of the neighbors instead of a simple MAX pooling.

PSNet [36] also argues that FPS and KNN-based extraction methods are too computationally expensive and proposes to sample and group the points simultane-

ously using its Spatial Feature Transform Function (SFTF). The points are mapped to a s -dimensional space with an MLP, s being the number of local regions, giving the correlation between the points and the local regions. The K most correlated points with a local region are the ones located in the region, effectively grouping and sampling the points at the same time, and allowing to reach higher inference speeds.

2 Point information fusion and matching

The next building block generally found in many networks is point information fusion between the source point cloud S at instant t and the target point cloud T at instant $t + 1$. This way, matchings or soft correspondences between points can be computed to estimate and refine the scene flow in the next stage. This usually implies building some correlation or cost matrix from which we can infer a cost volume or a flow embedding that is used to generate the final scene flow [37].

FlowNet3D [39] introduces the Point mixture and Flow embedding layer, illustrated in Fig. 1.2, that produces embeddings e_i encoding the motion of point i based on the feature similarities and the spatial relationships with its neighbors in the target point cloud T . The idea is to aggregate flow votes from the neighbors where the features passed to the MLP are learning weights to aggregate the neighboring offsets. This layer is also used in FESTA [58] to compute an initial flow.

Similarly, HALFlow [57] and WhatMatters [56] improve this embedding using the attention mechanism. They propose the Attentive Point mixture and Attentive Flow embedding layer that outputs attentive flow embeddings using two attentive layers. First, for each neighbor j in the target point cloud T , a vector $d_{i,j}$ capturing spatial information and a flow embedding vector $h_{i,j}$ are computed. Given these, an attentive weight $w_{i,j}$ for the neighbor j is computed using an MLP. The first attentive flow embedding e_i is then obtained by aggregating the neighboring flow embedding vectors $h_{i,j}$ using the attention weights. The same process is applied a second time, considering the neighbors k in the source point cloud S and aggregating their first flow embeddings $e_{i,k}$ to give the final flow embeddings e'_i .

Another line of works, such as FLOT [43], SCTN [35], FlowStep3D [34] or even Rigid3DSceneFlow [26] attempts to predict the scene flow based on correlation and cost matrices, from which soft correspondences can be determined. FLOT [43] suggests using the cosine distance and defines a cost matrix $C \in \mathbb{R}^{n_1 \times n_2}$ to encode similarities between points p_i and q_j in the source and target point clouds, respectively. Inspired by FLOT [43], SCTN [35] also predicts the scene flow based on a correlation matrix using cosine similarity instead of cosine distance. Similarly, FlowStep3D [34] uses the cost matrix proposed by FLOT [43], replacing as well the cosine distance by the cosine similarity, but further defines a soft correlation matrix $M \in \mathbb{R}^{n_1 \times n_2}$. Rigid3DSceneFlow [26] estimates soft correspondences with a similarity matrix $D \in \mathbb{R}^{n_1 \times n_2}$ in the feature space. FLOT [43], FlowStep3D [34], Rigid3DSceneFlow [26] and SCTN [35] all use these cost matrices or soft correspondences to compute an initial flow.

Inspired by previous stereo matching and optical flow works, PointPWC-Net [61], HPLFlowNet [28], PV-RAFT [59] and LiDARSceneFlow [21] employ a cost volume to predict the scene flow. A cost volume essentially stores the matching costs for associating a point in the first frame with its corresponding points in the second frame. These matching costs can be computed using the features and the offset vector passed to an MLP. PointPWC-Net [61] then proposes to aggregate those matching costs in a patch-to-patch manner to create the cost volume. HPLFlowNet [28] proposes the CorrBCL layer, which conveys the same idea as PointPWC-Net but works on the permutohedral lattice. The lattice points are aggregated in a patch-to-patch manner using aggregating functions and this process is illustrated in Fig. 1.5. In general, constructing cost volumes based on local neighborhoods only allows us to capture short-range relationships, hence not being able to predict large motions.

To alleviate this problem and capture long-range dependencies, PV-RAFT [59] proposes to combine their point-based cost volume with a voxel-based cost volume. First, a correlation field $C \in \mathbb{R}^{n_1 \times n_2}$ is computed using a dot product between every pair of points' features. Given a cube resolution a , resulting in a $a \times a \times a$ Rubik's cube centered around p_i , its voxel cost volume $C_v(p_i)$ is defined by aggregating the correlations of the target points q_j with respect to p_i , in each sub-cube or voxel, as illustrated in Fig. 1.4. LiDARSceneFlow [21] makes use of the soft correlation matrix from FlowStep3D [34] to construct an initial cost volume, without further explanation in their paper [21]. One downside of cost volumes is that they require a lot of memory for storage.

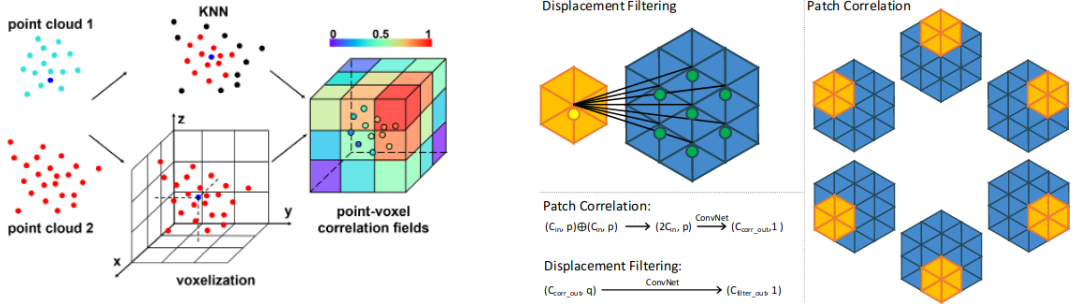


Figure 1.4: **Point-voxel cost volume** [59]. Figure 1.5: **CorrBCL** [28]. *CorrBCL* aggregating neighboring information in a patch-to-patch manner. *Point-based* cost volume C_p using KNN (top) and *voxel-based* cost volume C_v aggregating correlations in voxels (bottom).

3 Flow generation and refinement

Given the previous building blocks, The final step in all methods used in the literature is to generate the final scene flow. Although they all come with their own particularities, they usually work in a coarse to fine fashion, relying on operations such as up-sampling through interpolation or up-convolution, warping, refining through convolutions and MLPs or with iterative procedures such as a gated recurrent unit

(GRU) [4].

The pioneering work FlowNet3D [39] leads the way with its set_upconv layer, illustrated in Fig. 1.2. It essentially consists of a set_conv layer, aggregating neighboring flow embeddings at specified locations which we need to interpolate instead of locations given by the farthest sampling. Given the coarse flow embeddings at level l , $\{e_i\}_{i=1}^n$, the layer outputs denser embeddings $\{e'_i\}_{i=1}^{n'}$ at the previous level $l-1$, where $n' > n$. By stacking set_upconv layers, it finally outputs the scene flow F^* at the original resolution, or level 0. FESTA [58] goes one step further and introduces the Temporal abstraction with Attention layer (TA²) which works with an initial flow. As the initial flow F_0 is $\mathbf{0}$, it behaves as FlowNet3D [39] to output an initial dense flow F_1 . They then use this initial flow to warp the downsampled source point cloud at level L , i.e. apply the flow F_1 . After that, the process of computing flow embeddings and passing them through multiple set_upconv is repeated, producing an updated scene flow F_2 which is the final scene flow F^* .

HALFlow [57] and WhatMatters [56] follow the same idea as FESTA [58], but working in a coarse to fine manner instead of iteratively. At each level, they first up-sample sparser flow embeddings using the set_upconv layer to get the denser flow embeddings $\{e'_i\}_{i=1}^{n'}$. The sparser flow is also up-sampled using a 3-Nearest neighbor interpolation to obtain a denser flow $F' = \{f'_i\}_{i=1}^{n'}$. The source point cloud at level $l-1$ is then warped, $S_w = S + F'$, and flow re-embeddings $\{r'_i\}_{i=1}^{n'}$ are computed with the attentive flow embedding layer, presented in section 2, with the newly warped point cloud S_w . Finally, a scene flow predictor layer composed of an MLP followed by a fully connected layer (FC) refines the scene flow using the features, the flow embeddings, and the flow re-embeddings. The refined scene flow becomes the input for the next level, until reaching the final resolution flow F^* .

PointPWC-Net [61] also works in a pyramidal way using a K-nearest neighbor interpolation to get the denser flow F' and then warp the source point cloud. As PointPWC-Net works with a cost volume CV , a new cost volume CV' is computed using the warped point cloud S_w . Its scene flow predictor is composed of multiple PointConv layers to merge the features followed by an MLP and takes as input CV' , the features, the up-sampled flow F' , and the up-sampled features from the previous level scene flow predictor to output a flow at each level.

PointPWC-Net [61], FlowNet3D [39], HALFlow [57] and WhatMatters [56] all operate in a hierarchical way, meaning the number of iterations is limited by the number of levels in the pyramid. FH-Net [19] argues that previous methods [39, 43] rely on expensive deep feature decoding of feature matching after fully up-sampling. They propose a fast-hierarchical feature network outputting flows F_0 , F_1 , and F_2 at three different resolutions and guided from the local relative positions. The finer flows are obtained by up-sampling the coarser flows using the attention mechanism.

On the other hand, FESTA [58] is an iterative procedure not limited by the number of levels, although they iterate only twice for computational reasons. Other

works are also based on an iterative process, using a gated recurrent unit (GRU) [4]. FlowStep3D [34] first estimates a global flow F_0 given its soft correlation matrix M . At each iteration $k = 1, \dots, K$, the source point cloud is warped, $S_{k-1} = S_0 + F_{k-1}$, from which features are extracted using set_conv layers. Flow embeddings are then computed with the flow embedding layer from FlowNet3D [39], using the warped point cloud, and flow features are produced from set_conv layers. Given the previous hidden state h_{k-1} , the GRU gives the updated hidden state h_k given all the previously computed features and embeddings. The new hidden state h_k is then passed to two set_upconv layers to produce the residual flow ΔF_k , used to update the global scene flow, $F_k = F_{k-1} + \Delta F_k$.

PV-RAFT [59] also makes use of a GRU, but feeds its cost volume, $C_t = C_p + C_v$, composed of its point-based and voxel-based cost volumes C_p, C_v , to the GRU, as well as the features of the source point cloud and the current flow F_{k-1} . The cost volume is calculated from the warped point cloud S_{k-1} and the hidden state h_k is also used to predict the flow residuals from a flow head.

LiDARSceneFlow [21] exploits rigidity constraints by first obtaining a background mask using a pre-trained segmentation network. The foreground points are clustered using DBSCAN [22] to obtain N foreground object masks, giving in total $N+1$ masks $\mathcal{M} = \{M_n\}_{n=0}^N$. They introduce an error-awarded GRU that additionally takes an embedding of a point-metric error as input. The output hidden state h_k is then used to compute the residual flow ΔF_k . The updated flow F_k and the masks $\mathcal{M}$ are used by a Rigid flow converter to output $N+1$ rigid transformations $\mathcal{T}_k = \{T_n\}_{n=0}^N$ using the Kabsch algorithm [6], composing the rigid flow RF_k . This rigid flow is then hybridized with the scene flow F_k to give a hybrid flow HF_k as the final output of iteration k . It is worth noting that iterative methods such as FlowStep3D [34], LiDARSceneFlow [21], and PV-RAFT [59] are usually computationally expensive.

Rigid3DSceneFlow [26] constructs N masks $\mathcal{M} = \{M_n\}_{n=0}^N$ in the same manner as LiDARSceneFlow [21]. A transport plan $\mathbf{T}$ is then computed and serves as input for the Kabsch algorithm [6] to compute the rigid transformations $\mathcal{T} = \{T_n\}_{n=0}^N$. From these transformations, the point-wise rigid scene flow F_{rigid} is calculated. As DBSCAN can leave points unclustered, they attribute to these points a point-wise flow F computed directly from the soft correspondences, which is then refined through convolutions.

FLOT [43] uses optimal transport theory, and more particularly the Sinkhorn algorithm to estimate a transport plan $\mathbf{T}$ from its cost matrix. The initial scene flow F is computed using the transport plan $\mathbf{T}$ and then refined through multiple conv layers followed by an FC to give the final flow F^* . SCTN [35] follows the same approach as FLOT [43], using its own cost matrix.

This already gives quite an extensive view of all the components used for scene flow estimation, from feature learning to the final prediction, with some advantages and drawbacks of certain methods along with their own specificities.

4 Model complexities and inference time

Overall, while some methods [21, 34, 59, 43] propose a model with few parameters, almost no method reaches an inference speed above 10Hz ($<100\text{ms}$) except for FESTA [58] that only works with 2048 points as input and FH-Net [19]. Some methods, such as HALFlow [57] and WhatMatters [56] do not even discuss these measures in their papers. For LiDARSceneFlow [21], FlowStep3D [34], and PV-RAFT [59], the number of input points is not specified, not allowing any conclusions to be drawn. In terms of scalability, we notice from FlowNet3D [39] that methods employing farthest sampling (FPS) in their feature extractor do not scale well with the number of input points whereas a method such as HPLFlowNet [28], relying on a regular structure (the permutohedral lattice), performs better in that sense. It is worth noting that these are just general observations and that one should in practice compare all methods on the same GPU with the same dataset and number of input points for a fair comparison. In this thesis, an accent is put on the runtime of the proposed solutions, and the learning-based sampling of PSNet [36] is explored instead of regular FPS-based learning and downsampling and is plugged into the hierarchical network of FH-Net [19].

A more detailed view of the model complexities and runtime is given in Appendix 1.2 Table 1. For the more curious, an explanation of the different training approaches and loss functions used by the methods is available in Appendix 1.1.

Chapter 2

KITTI Vision Benchmark Suite

In the literature, two datasets are generally used. The first one is FlyingThings3D (FT3D) [41], consisting of 25000 synthetic stereo images with ground truth for optical and disparity images, from which 3D point clouds can be lifted for the task of scene flow. The second one is from the KITTI Vision Benchmark suite that was first introduced in [25] as a novel and challenging benchmark for computer vision tasks. The motivation is to reduce bias towards state-of-the-art solutions, which are usually not generalizing well in real-world scenarios, by offering a real-world, challenging dataset. It comprises 389 stereo and optical flow images, along with the disparity images to generate the points clouds. This is the chosen dataset in this thesis since it also offers the possibility to compute and cancel the ego-motion of the car, which is of interest for this work and for the subsequent steps.

1 General Description

The raw data [24] was captured by a VW Passat station wagon while driving in the metropolitan area of Karlsruhe, Germany. Fig. 2.1 shows the car, its dimensions, along with the sensors, marked in red, and their position with respect to the vehicle body. The heights, with respect to the ground surface, are marked in green, and the transformations between the sensors' reference frames are marked in blue. The car is equipped with the following sensors :

- $2 \times$ PointGray Flea2 grayscale cameras, denoted Cam0 and Cam1
- $2 \times$ PointGray Flea2 color cameras, denoted Cam2 and Cam3
- $1 \times$ Velodyne HDL-64E rotating 3D laser scanner.
- $1 \times$ OxTS RT3003 inertial and GPS navigation system.

Raw data

The cameras provide both color and grayscale images with an initial resolution of 1392×512 pixels. Corresponding rectified images with a resolution of 1240×376 pixels are also provided. Given two images taken from a left and right camera, respectively, the rectification is a transformation process that projects the two images on a common plane, simplifying the search for correspondences between pixels and

the computation of disparity images. The distance, or baseline $\mathbf{b}$, between each stereo pair is 54cm. The cameras are synchronized at about 10 Hz with respect to the Velodyne and triggered when the Velodyne is facing forward. The OxTS captures 30 GPS/IMU values for each frame: the geographic coordinates, including altitude, latitude, longitude, global orientation, velocities, accelerations, angular rates, accuracies, and satellite information. It provides updates at a frequency of 100 Hz and is not synchronized with the cameras/Velodyne, meaning the OxTS data associated with an image or scan can have a time difference of 5ms at worst. The Velodyne provides 3D scans at a frequency of 10 Hz, with an average of $\sim 120,000$ 3D points and reflectance values per scan. Unfortunately, we will not use them as no ground-truth scene flow is provided. Finally, KITTI provides calibration, projection, and rectification matrices for the cameras along with transformation matrices to switch between the sensors' coordinate frames.

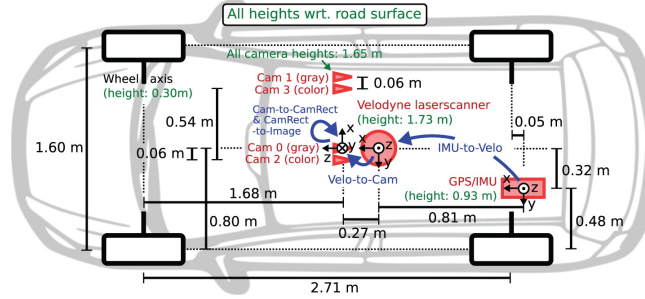


Figure 2.1: **The car and its sensors** [24]. Schematic of the car, its dimensions, its sensors (red), the heights (green), and the transformations between sensors' coordinate frames (blue).

2 KITTI Scene flow (2015)

The KITTI scene flow 2015 [42] is built from the KITTI raw data by annotating 400 dynamic scenes and consists of 200 training scenes and 200 test scenes [7]. Each scene is made of two consecutive frames at time t and $t + 1$, and comprises two pairs of stereo images, as illustrated in Fig. 2.2 (a,b,d,e). Two disparity maps, one for each frame, are also provided as shown in Fig. 2.2 (c,f), which are used to generate the point clouds. A disparity map gives, for each pixel, the horizontal displacement between the left and right image of the stereo pair. The larger the grayscale intensity of the pixel, the larger its displacement is. Finally, Fig. 2.2 (g) shows the optical flow between the left images at both timestamps. The optical flow is the displacement, in any direction, of a pixel between two time instances. The length and direction of the vector depend on the RGB intensity of the pixel.

3 Generating point clouds from images

Using the disparity maps and the optical flow, we generate the source point cloud S at time t and the target point cloud T at time $t + 1$ along with the scene flow F . Before, we introduce the pinhole camera model to understand how a camera works.

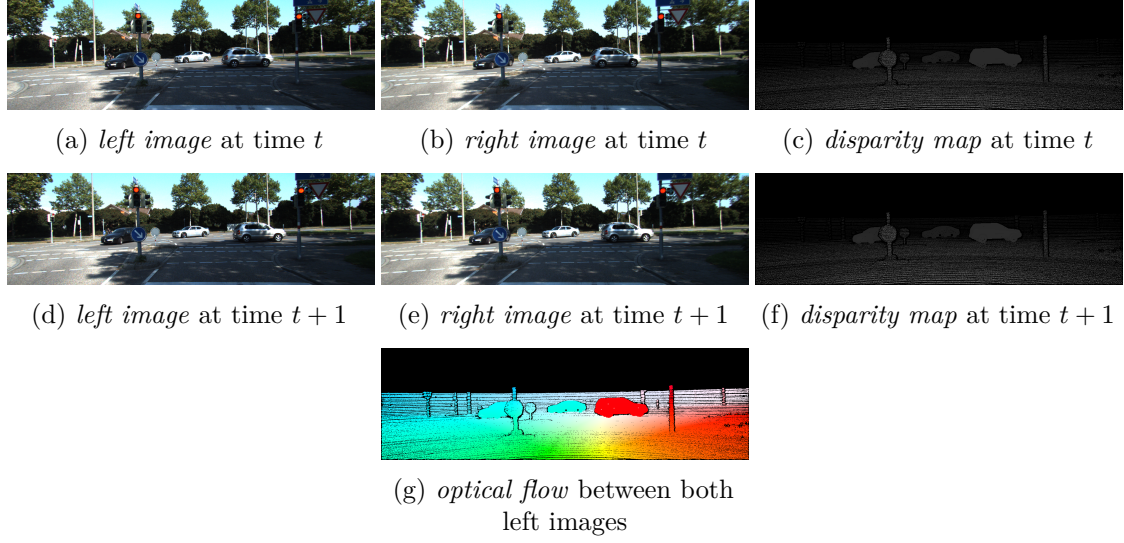


Figure 2.2: **Images of a dynamic scene.** RGB stereo pairs at instant t (a,b) and instant $t + 1$ (d,e) along with their disparity maps (c,f) and the optical flow between the two frames (g).

3.1 Pinhole camera model

The pinhole camera model [30] describes how a point in 3D space is mapped onto the image plane of the camera, also called the **central projection**, as shown in Fig. 2.3. Given the focal length $\mathbf{f}$, which is the distance between the camera center $\mathbf{C}$ and the principal point $\mathbf{P}$, and using similar triangles, a point with coordinates $\mathbf{X} = (X, Y, Z)^\top$ can be mapped to:

$$\mathbf{X} = (X, Y, Z)^\top \mapsto \left(\frac{fX}{Z}, \frac{fY}{Z} \right)^\top \quad (2.1)$$

The central projection can also be expressed in matrix notation using homogeneous coordinates. The idea behind homogeneous coordinates is to add an extra coordinate 1, so the point (x, y) becomes $(x, y, 1)$. In fact, all points (kx, ky, k) represent the same point (x, y) since we can get back its original coordinates by dividing by k . Homogeneous coordinates ease computations by allowing matrix multiplication. The central projection can be rewritten as:

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \mapsto \begin{pmatrix} fX \\ fY \\ Z \end{pmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \quad (2.2)$$

3.2 Intrinsic parameters

In practice, additional parameters need to be taken into account. First, the origin of the image plane is usually at the top-left corner and not at the principal point and an offset of (p_x, p_y) has to be included, where (p_x, p_y) is the coordinate of the

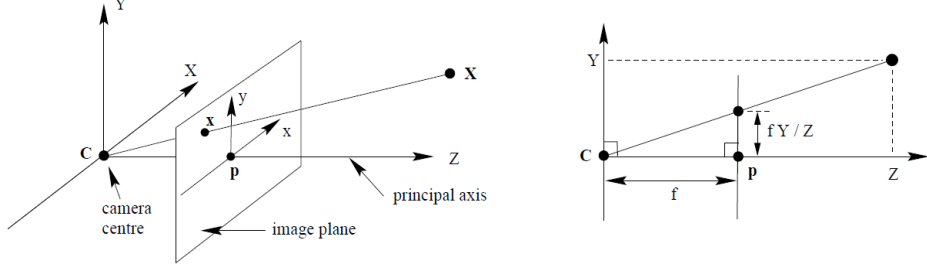


Figure 2.3: **Pinhole camera model** [30]. $\mathbf{C}$ is the *camera centre* and is the origin coordinate. $\mathbf{P}$ is the *principal point* and $\mathbf{f}$ is the *focal length*. The *principal axis* pierces the image plane at $\mathbf{P}$, which is at the center of the *image plane*.

principal point on the image plane. The projection now becomes:

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \mapsto \begin{pmatrix} fX + Zp_x \\ fY + Zp_y \\ Z \end{pmatrix} = \begin{bmatrix} f & 0 & p_x & 0 \\ 0 & f & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \quad (2.3)$$

From the right-hand side of 2.3, we can extract the **intrinsic matrix** $\mathbf{K}$:

$$\mathbf{K} = \begin{bmatrix} \alpha_x & 0 & p_x \\ 0 & \alpha_y & p_y \\ 0 & 0 & 1 \end{bmatrix} \quad (2.4)$$

where $\alpha_x = fm_x$ and $\alpha_y = fm_y$ represent the focal lengths in pixel dimensions in the x and y directions. m_x and m_y represent the number of pixels per unit distance in the image coordinates. These additional parameters allow us to go from the image coordinates to the pixel coordinates. Using the intrinsic parameters $\mathbf{K}$, we can transform a 3D point $\mathbf{X}_{\text{CAM}}$, from camera coordinates to pixel coordinates:

$$\mathbf{x} = \mathbf{K}[\mathbf{I}|\mathbf{0}]\mathbf{X}_{\text{CAM}} \quad (2.5)$$

where $\mathbf{x} = (\mathbf{u}, \mathbf{v})^\top$ is the point in pixel coordinates, $\mathbf{I}$ is the identity matrix and $[\cdot|\cdot]$ denotes the column concatenation of two matrices.

3.3 Extrinsic parameters

With the intrinsic parameters, we have the transformation from 3D camera coordinates to 2D-pixel coordinates. In general, points are expressed in the world coordinate frame and we also need the transformation from the world coordinates to the camera coordinates. As shown in Fig. 2.4, this transformation is characterized by a rotation matrix $\mathbf{R} \in \mathbb{R}^{3 \times 3}$ and a translation vector $\mathbf{t} \in \mathbb{R}^{3 \times 1}$. The point $\mathbf{X}_{\text{CAM}}$ is then expressed as:

$$\mathbf{X}_{\text{CAM}} = [\mathbf{R}|\mathbf{t}]\mathbf{X} \quad (2.6)$$

where $\mathbf{X} = (X, Y, Z, 1)^\top$ is now the point in the world coordinate frame and $[\mathbf{R}|\mathbf{t}] \in \mathbb{R}^{3 \times 4}$ is called the **extrinsic matrix**, with 6 degrees of freedom. Eq. 2.5 is updated in order to derive the pixel coordinates:

$$\mathbf{x} = \mathbf{K}[\mathbf{R}|\mathbf{t}]\mathbf{X} \quad (2.7)$$

where the projection matrix $\mathbf{P} = \mathbf{K}[\mathbf{R}|\mathbf{t}] \in \mathbb{R}^{3 \times 4}$ defines our final camera model and the correspondence between a 3D point in world coordinates and its 2D pixel on the image plane.

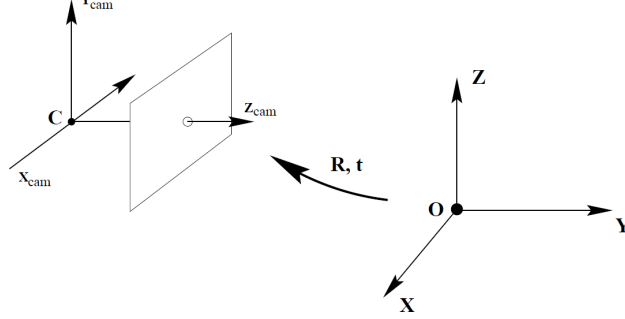


Figure 2.4: **Camera rotation and translation** [30]. $\mathbf{O}$ is the origin of the *world frame* and $\mathbf{C}$ the origin of the *camera frame*. $\mathbf{R}, \mathbf{t}$ are the rotation matrix and translation vector to change the reference frame.

3.4 Point clouds lifting

To generate a point cloud, we need to transform in the opposite direction, that is from the 2D pixel $\mathbf{x} = (\mathbf{u}, \mathbf{v})^\top$ of the left rectified image, to the 3D left rectified camera point $\mathbf{X}_{\text{CAM}} = (X, Y, Z)^\top$. For that, the intrinsic matrix $\mathbf{K}$ of the rectified left camera is useful but not sufficient. We also need the right rectified image of the stereo pair to infer the depth component, Z , of the corresponding point. As shown in Fig. 2.5, given the focal length $\mathbf{f}$ and the optical centers, $\mathbf{O}_1$ and $\mathbf{O}_2$, of the left and right camera respectively, separated by the baseline $\mathbf{b}$, the depth Z can be obtained using similar triangles [12].

$$\left. \begin{aligned} \frac{f}{Z} &= \frac{u_1}{X} \\ \frac{f}{Z} &= \frac{-u_2}{b - X} \end{aligned} \right\} \Rightarrow Z = \frac{bf}{u_1 - u_2} \quad (2.8)$$

where $\mathbf{d} = u_1 - u_2$ is the **disparity**, or the horizontal displacement between the u coordinates of the two pixels. As KITTI provides the disparity map for each stereo pair, we can easily retrieve Z . In order to retrieve X and Y , we can simply use the intrinsic parameters $\mathbf{K}$ and inverse the operation in equation 2.3.

$$\begin{aligned} X &= \frac{Z(u - p_x)}{f} \\ Y &= \frac{Z(v - p_y)}{f} \end{aligned} \quad (2.9)$$

Given the rectified stereo pair, the disparity map and the intrinsic matrix $\mathbf{K}$ that are provided by KITTI and following equations 2.8 and 2.9, we can create for each pixel $\mathbf{x} = (\mathbf{u}, \mathbf{v})^\top$ its corresponding 3D point $\mathbf{X}_{\text{CAM}} = (X, Y, Z)^\top$ and thus create a point cloud. To lift the source point cloud S at time t , we can use equations 2.8 and

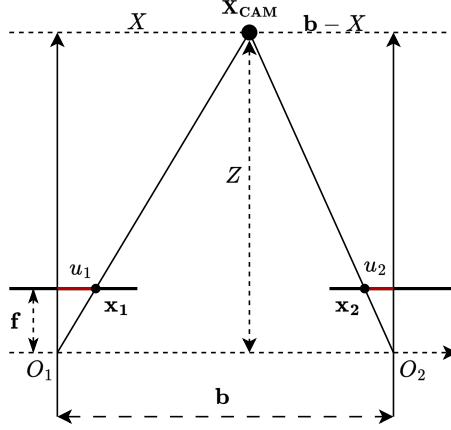


Figure 2.5: **Depth estimation** [12]. Estimation of the depth given the focal length f , the baseline b and the disparity $u_1 - u_2$. O_1, O_2 are the optical centers of the cameras.

2.9 as is but in order to lift the target point cloud T at time $t + 1$, we actually need to modify equation 2.9. The reason is that, as shown in Fig. 2.2 (f), the disparity image at time $t + 1$ given by KITTI is actually mapped into the reference frame of the left image at time t using the optical flow. The equation then becomes:

$$\begin{aligned} X &= \frac{Z(u + o_x - p_x)}{f} \\ Y &= \frac{Z(v + o_y - p_y)}{f} \end{aligned} \quad (2.10)$$

where $\mathbf{o} = (o_x, o_y)^\top$ represents the optical flow of the pixel $\mathbf{x} = (\mathbf{u}, \mathbf{v})^\top$. As proposed by [38], the target point cloud T is then back-projected onto the image plane, and only points falling inside the image plane are kept, creating some occlusion.

In theory, as rectified images have a resolution of 1240×376 pixels, they should generate point clouds with 466240 points. In practice, not all pixels have a valid disparity or a valid optical flow, leaving us with point clouds with an average size of ~ 83992 points. Finally, both source and target point clouds are randomly downsampled to 65536 points, giving 200 pairs of point clouds S and T for all dynamic scenes at time t and $t + 1$ respectively, expressed in the reference frame of the left camera. The ground truth scene flow F for each scene is computed as $F = T - S$. Fig. 2.6 (a) shows the point cloud source S with color information while Fig. 2.6 (b) shows both the source point cloud S and target point cloud T from the images in Fig. 2.2, where you can observe the motion between the two.

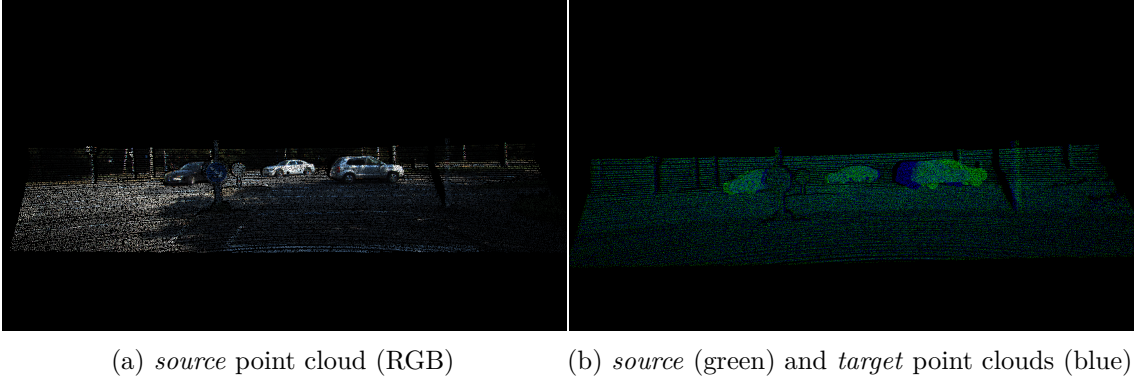


Figure 2.6: **Point clouds generation.** Generated source point cloud with RGB information (a) and generated source (green) and target (blue) point clouds (b).

4 Ego-motion compensation

The cameras are triggered every $100ms$ while the car is still moving and the car's reference frame at time t is not the same at time $t + 1$ due to this motion. The transformation between the two frames, $\mathbf{T}_{ego}$, is called the **ego-motion** of the car. As a consequence, the reference frame of the left camera at time $t + 1$ is also transformed and the target point cloud T is not expressed in the same reference frame as the source point cloud S . Static points in the scene are assigned the flow resulting from the ego-motion instead of a zero flow, whereas moving points are assigned the ego-motion flow combined with their real flow, hence not representing their true motion. Given the GPS/IMU data captured by the OxTS, we can estimate the motion of the points induced by $\mathbf{T}_{ego}$ and use it to express the source and target point clouds in the same reference frame.

4.1 Rigid-body transformations

We first introduce the concept of **rigid-body transformation** [53]. The homogeneous transformation ${}^B\mathbf{T}_A \in \mathbb{R}^{4 \times 4}$ transforms points from coordinate frame A to coordinate frame B.

$${}^B\mathbf{T}_A = \begin{pmatrix} {}^B\mathbf{R}_A & {}^B\mathbf{t}_A \\ \mathbf{0} & \mathbf{1} \end{pmatrix} \quad (2.11)$$

${}^B\mathbf{R}_A \in \mathbb{R}^{3 \times 3}$ is the rotation matrix of frame A relative to frame B and ${}^B\mathbf{t}_A \in \mathbb{R}^{3 \times 1}$ represents the origin of frame A relative to frame B. A point ${}^A\mathbf{x}$ expressed relative to coordinate frame A can be expressed relative to the coordinate frame B with:

$${}^B\mathbf{x} = {}^B\mathbf{T}_A {}^A\mathbf{x} \quad (2.12)$$

where ${}^B\mathbf{x}$ is the point expressed relative to coordinate frame B and ${}^A\mathbf{x}$, ${}^B\mathbf{x} \in \mathbb{R}^{4 \times 1}$ are both in homogeneous coordinates. The inverse ${}^B\mathbf{T}_A^{-1}$ transforms points from coordinate frame B to coordinate frame A and the composition of homogeneous transformation matrices can be achieved through matrix multiplication.

$${}^B\mathbf{T}_A^{-1} = {}^A\mathbf{T}_B = \begin{pmatrix} {}^B\mathbf{R}_A^\top & -{}^B\mathbf{R}_A^\top {}^B\mathbf{t}_A \\ \mathbf{0} & \mathbf{1} \end{pmatrix} \quad (2.13)$$

$${}^C\mathbf{T}_A = {}^C\mathbf{T}_B {}^B\mathbf{T}_A \quad (2.14)$$

4.2 Mercator's projection

Given the geographic coordinates (alt, lat, lon) provided by the OxTS, where alt is the altitude, lat the latitude, and lon the longitude, one can compute the position of the IMU, ${}^W\mathbf{x}_{IMU} = (X, Y, Z)^\top$, relative to the world coordinate frame, in the Euclidean space [24]:

$$\begin{aligned} X &= s \times r \times \frac{\pi lon}{180} \\ Y &= s \times r \times \log\left(\tan\left(\frac{\pi(90 + lat)}{360}\right)\right) \\ Z &= alt \end{aligned} \quad (2.15)$$

where $r \approx 6378137$ meters is the earth radius and scale $s = \cos(\frac{lat_0 \times \pi}{180})$ determines the Mercator scale, where lat_0 denotes the latitude of the first frame at time t . Given the angles of the car axis ($roll, pitch, yaw$), one can compute the orientation of the IMU, ${}^W\mathbf{R}_{IMU}$, relative to the world coordinate frame:

$$\begin{aligned} {}^W\mathbf{R}_{IMU} &= \mathbf{R}_z \mathbf{R}_y \mathbf{R}_x \\ &= \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\gamma) & -\sin(\gamma) \\ 0 & \sin(\gamma) & \cos(\gamma) \end{bmatrix} \end{aligned} \quad (2.16)$$

where γ is the roll angle of the x-axis, β is the pitch angle of the y-axis and α is the yaw angle of the z-axis. Given these two equations, we can compute the transformations from the IMU coordinate frame to the world coordinate frame, ${}^W\mathbf{T}_{IMU_t}$ and ${}^W\mathbf{T}_{IMU_{t+1}}$, at both timestamps. The goal is now to compute the transformation ${}^{C2_{t+1}}\mathbf{T}_{C2_t}$, where $C2$ is the left color camera, such that we can warp the source point cloud S into the reference frame of the left color camera at $t + 1$. Following the notations of Fig. 2.1 for the cameras, the transformations given by KITTI are used.

- ${}^{velo}\mathbf{T}_{IMU}$, from the IMU to the velodyne frame.
- ${}^{C0_{unrect}}\mathbf{T}_{velo}$, from the Velodyne to the left gray unrectified camera frame.
- ${}^{C0_{rect}}\mathbf{T}_{C0_{unrect}}$, from the unrectified to the rectified left gray camera frame.
- ${}^{C2}\mathbf{T}_{C0_{rect}}$, from the rectified left gray to the rectified left color camera frame.

We can then compute ${}^{C2_{t+1}}\mathbf{T}_{C2_t}$ using composition of transformations as illustrated in Fig. 2.7.

$$\begin{aligned} {}^{IMU_{t+1}}\mathbf{T}_{IMU_t} &= {}^W\mathbf{T}_{IMU_{t+1}}^{-1} {}^W\mathbf{T}_{IMU_t} \\ {}^{C2}\mathbf{T}_{IMU} &= {}^{C2}\mathbf{T}_{C0_{rect}} {}^{C0_{rect}}\mathbf{T}_{C0_{unrect}} {}^{C0_{unrect}}\mathbf{T}_{velo} {}^{velo}\mathbf{T}_{IMU} \\ {}^{C2_{t+1}}\mathbf{T}_{C2_t} &= {}^{C2_{t+1}}\mathbf{T}_{IMU_{t+1}} {}^{IMU_{t+1}}\mathbf{T}_{IMU_t} {}^{C2_t}\mathbf{T}_{IMU_t}^{-1} \end{aligned} \quad (2.17)$$

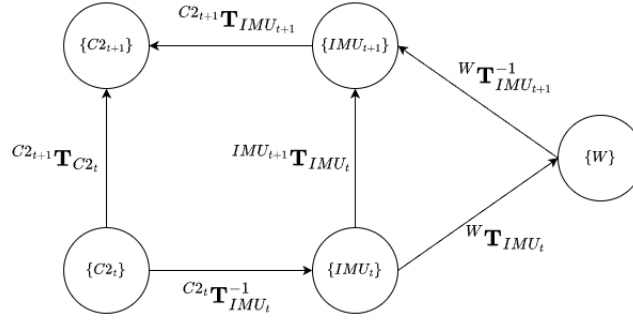


Figure 2.7: **Composition of transformations to compute $C^{2_{t+1}}T_{C^{2_t}}$.** Nodes represent different coordinates frames while the arrows indicate the transformations from one reference frame to another.

Eq. 2.12 can be applied to all points of the source point cloud S such that they are expressed in the rectified left color camera coordinates at $t + 1$. Fig. 2.8 (a) shows both source and target point clouds before compensation, where we clearly see that traffic posts and signs appear to be moving forward because the car actually moved towards them in between the two frames. The effect of compensation is shown in Fig. 2.8 (b), where traffic posts and signs now appear to be static, as they should. It should be noted that OxTS data are only available for 142 out of the 200 frames, further reducing the size of the dataset.

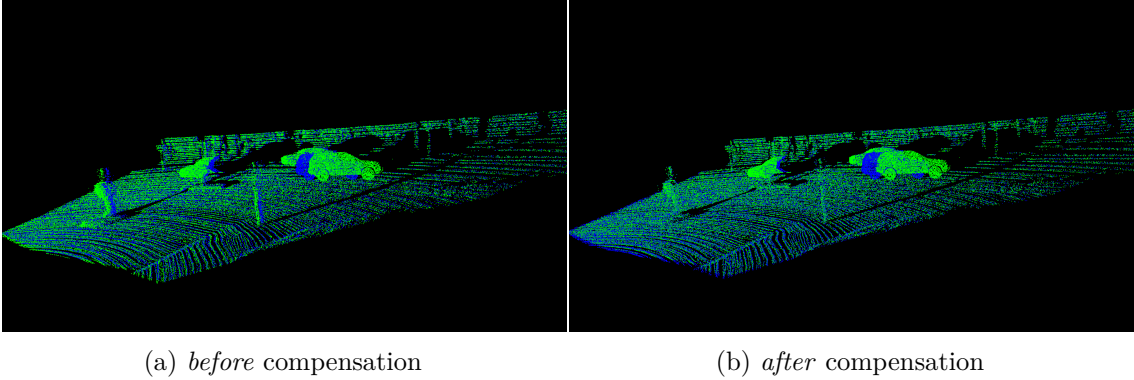


Figure 2.8: **Ego-motion compensation.** Source (blue) and target (green) point clouds *before* (a) and *after* (b) compensation of the ego-motion.

Chapter 3

Ground segmentation and Ego-motion refinement

Inspired by the Rigid3DSceneFlow [26] and LiDARSceneFlow [21] methods that divide the scene into foreground and background, our goal is to implement a similar approach. However, these methods rely on a trained background segmentation head, further increasing the number of parameters and complexity of their models. As a first preprocessing step, we would like to classify points as ground and non-ground in a simpler and non-learning way. The motivation is that, in a scene, many points are actually static and we would like to feed to the final network only points of interest that are moving and part of the foreground, whereas other previous methods presented in Chapter 1 do not make any distinction between the background and the foreground objects or remove the ground naïvely.

To begin with, an algorithm using traditional operations on point clouds to obtain a ground mask is presented. In a second phase, the ego-motion computed in Chapter 2 Section 4 is refined using the newly classified ground points with the point-to-plane ICP algorithm [18]. The reason is that the OxTS sensor is not perfectly synchronized with the cameras and the velodyne, and the measurements can be noisy, tempering with the initial ego-motion estimation. Both algorithms are implemented in C++, leveraging the Point Cloud Library (PCL) [50] that implements many standard operations on point clouds.

1 Ground segmentation

In this section, the algorithm used for ground segmentation is detailed along with all the operations it relies on. There exist multiple methods in the literature [27] that are based on different techniques ranging from occupancy grids and elevation maps, ground modeling such as plane fitting and line extraction, clustering and region growing algorithms to deep learning-based segmentation. In our case, we want the algorithm to remain simple as this is only a preprocessing step, intending to get rid of as many points as possible while avoiding false positives, i.e. non-ground points that could be moving and classified as ground.

With this in mind, a simple algorithm relying on local plane fitting is devised. The general idea is the following: the point cloud is first down-sampled using a voxel-grid filtering technique, and then for each point, its K -nearest neighbors are queried using a Kd-Tree data structure. Given the K -nearest neighbors, the normal of each point is computed and the points are classified as ground based on two thresholds, an angle threshold α and a height threshold $\mathcal{H}$. Points for which the angle between their normal and the y -axis is less than α and for which their height is below $\mathcal{H}$ are classified as ground points. One downside of this approach is that it cannot deal with uneven ground or sloped terrain, which can lead to limited accuracy in some cases. It is not such a huge concern in our case as the data from the KITTI benchmark suite [25] is mainly taken in a metropolitan area but even if the road is very steep, it is still parallel to the car and its sensors, resulting in a flat road in appearance.

1.1 Voxel-grid filtering

One of the very first steps when working with point clouds is usually to down-sample the point cloud for efficiency reasons. One way to achieve this result is through voxel-grid filtering. Given an input point cloud data $\{x_i \in \mathbb{R}^3\}_{n=1}^N$ containing N points, it first builds a 3D discrete grid of voxels, i.e. 3D bounding boxes, over the input space. Given a user-defined voxel size, this voxelization process outputs N' voxels $\mathcal{V} = \{V_i\}_{n=1}^{N'}$. For each voxel V_i , a centroid C_i from all the points contained in the voxel is computed as:

$$C_i = \frac{1}{|V_i|} \sum_{x_j \in V_i} x_j \quad (3.1)$$

The voxel-grid filtering, illustrated in Fig. 3.1, outputs a set of centroids $\mathcal{C} = \{C_i \in \mathbb{R}^3\}_{n=1}^{N'}$ of size N' representing the underlying distribution of points.

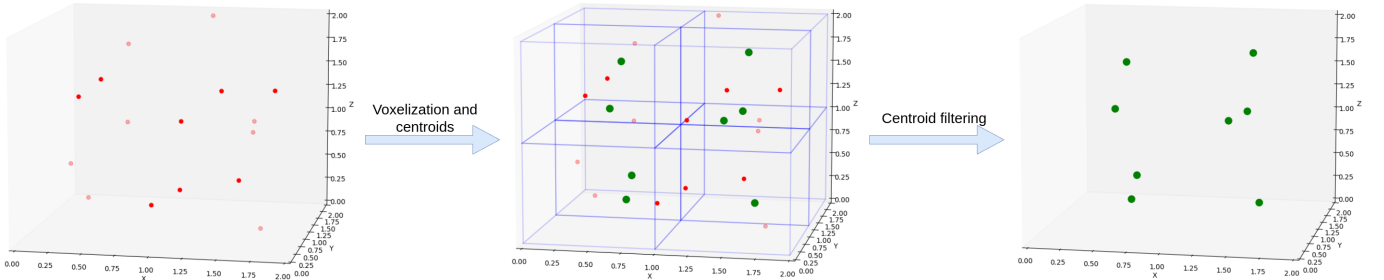


Figure 3.1: **Voxel-grid Filtering.** The space is first divided into voxels. For each voxel, a centroid is extracted, composing in the end the down-sampled point cloud.

1.2 K-nearest neighbors search

For the next steps, i.e. the K -nearest neighbors search and the estimation of normal vectors, we consider the down-sampled data $\mathcal{C}$. Finding the nearest neighbor of a point can be done in a naïve way by linearly scanning the data and keeping track of the closest point so far. To make the search more efficient, the Kd-tree data structure is used. A Kd-tree [14] builds a binary search tree organizing points in

a K -dimensional space. The tree recursively splits the current partition in half at each node, alternating between the dimensions at each level, as illustrated in Fig. 3.2.

To find the nearest neighbor of a query point, its region is found by traversing the tree from top to bottom down to a leaf node. This leaf node might not be its nearest neighbor so we have to traverse the tree back up, checking at each node if it is closer than the current closest neighbor and if so, the sub-regions of that point have to be explored. At worst, the whole tree has to be explored but the complexity is on average $O(\log N)$, where N is the number of nodes in the tree or samples in the K -dimensional space. Similarly, the K -nearest neighbors can be searched efficiently using a Kd-tree.

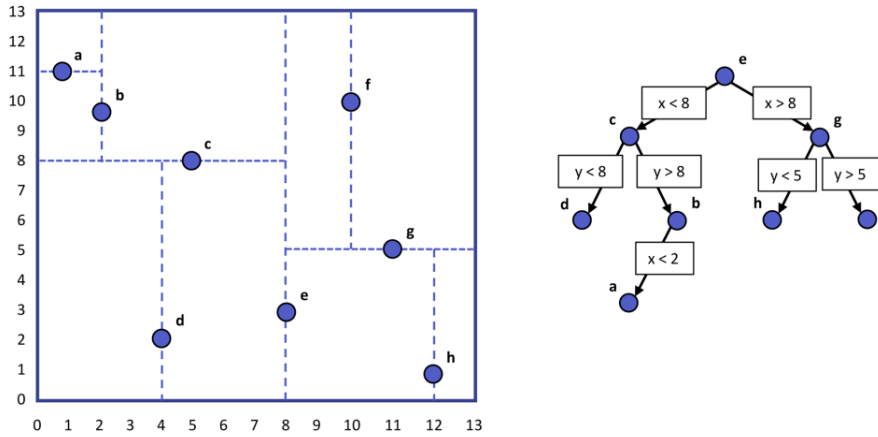


Figure 3.2: **Kd-tree partitioning** [54]. Each level is assigned a dimension and each node divides the remaining space into two parts following the dimension of its current level. The left child has a smaller value along that dimension while the right child has a greater value.

1.3 Normal estimation

The final building block for our ground segmentation algorithm is the estimation of the points' normal vector. Estimating the normal of a point is approximated by the problem of estimating the normal of the plane tangent to the underlying surface of the point [47]. Given our down-sampled point cloud C , we want to estimate the normals $\{\vec{n}_i \in \mathbb{R}^3\}_{i=0}^{N'}$, $\vec{n}_i$ being the normal of point C_i . A plane can be defined by its normal vector $\vec{n}$ and a point $\mathbf{x} \in \mathbb{R}^3$. Given our point of interest C_i , its K -nearest neighbors $\mathcal{N}_i = \{x_k\}_{k=0}^K$ are first queried. The point $\mathbf{x}$ of the plane is then defined as:

$$\mathbf{x} = \bar{x} = \frac{1}{K} \sum_{k=1}^K x_k \quad (3.2)$$

where $\mathbf{x}$ is the centroid of the neighborhood $\mathcal{N}_i$. The normal vector $\vec{n}_i$ can be found by computing the eigenvalues and eigenvectors of the covariance matrix $\mathbb{C} \in \mathbb{R}^{3 \times 3}$ of

$\mathcal{N}_i$ using the principal component analysis (PCA) [8]:

$$\mathbb{C} = \frac{1}{K} \sum_{k=1}^K w_k \cdot (x_k - \bar{x}) \cdot (x_k - \bar{x})^\top \quad (3.3)$$

$$\mathbb{C} \cdot \vec{v}_j = \lambda_j \cdot \vec{v}_j, j \in \{0, 1, 2\}$$

where w_k is a possible weight for x_k . The eigenvalues of $\mathbb{C}$ are $\lambda_j \in \mathbb{R}$ and $\vec{v}_j \in \mathbb{R}^3$ are the eigenvectors, forming an orthogonal frame corresponding to the principal components of $\mathcal{N}_i$. If the eigenvalues are ordered as $0 \leq \lambda_0 \leq \lambda_1 \leq \lambda_2$, then the eigenvector $\vec{v}_0$ corresponding to the smallest eigenvalue λ_0 is the approximation of $\vec{n}_i = \{n_i^x, n_i^y, n_i^z\}$, the normal vector of $\mathcal{N}_i$. The normal approximation of a point is shown in Fig. 3.3.

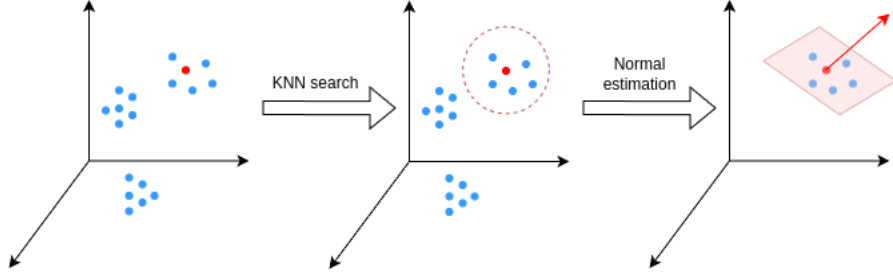


Figure 3.3: **Normal approximation.** The K -nearest neighbors of the query point (red) are computed followed by the estimation of its normal using PCA.

1.4 Classification

The algorithm for classifying points as ground or non-ground is depicted in Alg. 1. The input point cloud P of size N is first down-sampled through the voxel-grid filtering, given the voxel size s , to obtain the centroids C of size N' from which the neighborhoods $\mathcal{N}$ are inferred with respect to K . The normals for each centroid are computed from the neighborhoods and given their normals, the centroids are classified based on the angle α and the height $\mathcal{H}$ thresholds. In the end, we want to classify the points at the original resolution meaning that each point x_i is assigned the label of its corresponding centroid as the final step, resulting in the mask $\mathcal{M}$.

Algorithm 1: Ground segmentation algorithm

Input: a point cloud $P = \{x_i\}_{i=1}^N$, a voxel size s , a number of neighbors K , an angle threshold α and a height threshold $\mathcal{H}$

Output: a segmentation mask $\mathcal{M} = \{m_i \in \{0, 1\}\}_{i=1}^N$ where $m_i = 1$ if ground and $m_i = 0$ if non-ground

```
1:  $y\text{-axis} \leftarrow (0, 1, 0)$ 
2:  $C \leftarrow \text{VoxelGridFiltering}(P, s)$ 
3:  $\mathcal{N} \leftarrow \text{KdTreeSearch}(C, k)$ 
4:  $\vec{n} \leftarrow \text{NormalEstimation}(\mathcal{N})$ 
5:  $\mathcal{M}_{\text{downsampled}} \leftarrow \{0\}_{i=1}^{|C|}$ 
6: for  $i = 1$  to  $|C|$  do
7:    $angle \leftarrow \cos^{-1}(y\text{-axis} \cdot \vec{n}_i)$ 
8:   if  $angle < \alpha$  and  $C[i][y] < \mathcal{H}$  then
9:      $\mathcal{M}_{\text{downsampled}}[i] = 1$ 
10:  end if
11: end for
12:  $\mathcal{M} \leftarrow \{0\}_{i=1}^N$ 
13: for  $i = 1$  to  $N$  do
14:    $\text{CentroidIdx} \leftarrow \text{GetCentroidIndex}(x_i)$ 
15:    $\mathcal{M}[i] = \mathcal{M}_{\text{downsampled}}[\text{CentroidIdx}]$ 
16: end for
17: return  $\mathcal{M}$ 
```

2 Ego-motion refinement

Using the mask $\mathcal{M}$, both the source and target point clouds S and T can be indexed to obtain the source ground points $G_s \in \mathbb{R}^{N' \times 3}$ and the target ground points $G_t \in \mathbb{R}^{M' \times 3}$. As those points are static, they can be used to compute a new transformation $\mathbf{T} \in \mathbb{R}^{4 \times 4}$, aligning them more closely.

Finding this transformation boils down to the problem of point cloud registration [32]. Given two point clouds $X \in \mathbb{R}^{M \times 3}$ and $Y \in \mathbb{R}^{N \times 3}$ and supposing they have K pairs of correspondences, the goal is to find the rigid transformation given by a rotation matrix $\mathbf{R} \in \mathbb{R}^{3 \times 3}$ and a translation vector $\mathbf{t} \in \mathbb{R}^{3 \times 1}$ which best aligns the point clouds:

$$\arg \min_{\mathbf{R}, \mathbf{t}} d(X, \mathbf{R}Y + \mathbf{t}) = \arg \min_{\mathbf{R}, \mathbf{t}} \sum_{k=1}^K \|X_k - \mathbf{R}Y_k - \mathbf{t}\|_2^2 \quad (3.4)$$

The correspondences between the two sets of points are either known or must be found. Given the correspondences, the rigid transformation $\mathbf{T}$ can be obtained using the singular value decomposition (SVD) [10].

2.1 ICP

In the setup where the correspondences are unknown, one of the most well-known approaches to solve the point cloud registration problem is the Iterative Closest point [15] algorithm, which can be expressed as a non-linear least squares problem and solved using the Gauss-Newton algorithm [5]. As its name suggests, it is an iterative procedure that, given $\mathbf{R}$ and $\mathbf{t}$ at the current iteration, selects for each point X_k in the first point cloud its closest point Y_k in the target point cloud as its correspondence:

$$Y_k = \arg \min_{Y_i} \|X_k - \mathbf{R}Y_i - \mathbf{t}\|_2^2 \quad (3.5)$$

where $\mathbf{R}$ and $\mathbf{t}$ are the current estimates and are updated using Eq. 3.4 at each iteration. The process stops when $d(X, \mathbf{R}Y + \mathbf{t})$ becomes small enough.

2.2 Point-to-Plane ICP

In our case, it is worth noting that G_s and G_t represent ground surfaces and consequently two distinct planes. Therefore, it can be interesting to take that into account and to use an improved version of ICP called point-to-plane ICP [18], which minimizes the orthogonal distances between the points in one point cloud and the corresponding local planes in the other [32]. The error minimized along the surface normal, illustrated in Fig. 3.4 becomes:

$$\arg \min_{\mathbf{R}, \mathbf{t}} d(X, \mathbf{R}Y + \mathbf{t}) = \arg \min_{\mathbf{R}, \mathbf{t}} \sum_{k=1}^K w_k (n_k \cdot (X_k - \mathbf{R}Y_k - \mathbf{t}))^2 \quad (3.6)$$

where w_k is a weight that can be attributed to a correspondence to give it more or less importance and n_k is the normal estimated at point X_k following Section 1.3. Using the transformation ${}^{C2t+1}\mathbf{T}_{C2t}$ from Eq. 2.17 as our initial transformation, the ICP procedure with the point-to-plane error metric is summarized in Alg. 2. For efficiency reasons, G_s and G_t are down-sampled using the voxel-grid filtering, as explained in Section 1.1.

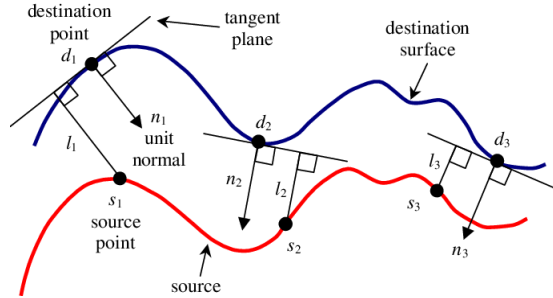


Figure 3.4: **point-to-plane error metric** [40]. s_i are here the source points and d_i the target points. n_i are the normal vectors and l_i are the distances between the source points and their corresponding plane.

Algorithm 2: Point-to-plane ICP for ego-motion refinement

Input: The source ground points G_s and the target ground points G_t , the initial transformation ${}^{C^{2t+1}}\mathbf{T}_{C^{2t}}$ and a distance threshold ϵ

Output: The final transformation $\mathbf{R}, \mathbf{t}$ best aligning G_s and G_t

```
1:  $\mathbf{R}, \mathbf{t} \leftarrow {}^{C^{2t+1}}\mathbf{R}_{C^{2t}}, {}^{C^{2t+1}}\mathbf{t}_{C^{2t}}$ 
2:  $d \leftarrow \infty$ 
3: while  $d > \epsilon$  do
4:   for  $k = 1$  to  $|G_s|$  do
5:      $Y_k \leftarrow \arg \min_{Y_i} \|X_k - \mathbf{R}Y_i - \mathbf{t}\|_2^2$ 
6:   end for
7:    $\mathbf{R}, \mathbf{t} \leftarrow \arg \min_{\mathbf{R}, \mathbf{t}} \sum_{k=1}^K w_k (n_k \cdot (X_k - \mathbf{R}Y_k - \mathbf{t}))^2$ 
8:    $d \leftarrow \sum_{k=1}^K w_k (n_k \cdot (X_k - \mathbf{R}Y_k - \mathbf{t}))^2$ 
9: end while
10: return  $\mathbf{R}, \mathbf{t}$ 
```

In the end, the ground segmentation algorithm followed by the point-to-plane ICP procedure allows for obtaining a ground mask, which can in turn be used to refine the ego-motion.

Chapter 4

Moving object segmentation

Using their segmentation head, LiDARSceneFlow [21] and Rigid3DSceneFlow [26] both obtain a background and a foreground mask while we only have a ground mask at our disposal for now. However, background objects can still be present in the non-ground points and consequently, we need to divide the non-ground points into background and foreground objects, which is achieved through clustering. While LiDARSceneFlow [21] and Rigid3DSceneFlow [26] cluster their foreground into individual objects for which they estimate a flow, we go one step further and classify the background and foreground objects obtained from the non-ground points into moving and static objects. This way, foreground static objects, e.g. a parked car on the side of the road, can be assigned the ego-motion flow and do not need to be fed to the neural network whereas LiDARSceneFlow and Rigid3DSceneFlow predict a flow for all foreground objects, including static ones.

More formally, we denote the non-ground points in the source and target point clouds S and T as NG_s and NG_t , respectively. We then use a clustering algorithm to cluster the non-ground points in both point clouds NG_s and NG_t . The clusters of NG_s and NG_t are denoted $C_s = \{c_{s1}, c_{s2}, \dots, c_{sn}\}$ and $C_t = \{c_{t1}, c_{t2}, \dots, c_{tm}\}$, respectively, where n is the number of clusters in NG_s and m the number of clusters in NG_t . The clusters of C_s are then associated with the clusters of C_t using point cloud descriptors, where for each pair, we classify the points as static or moving. Ideally, we would like to have a one-to-one mapping between C_s and C_t , and that $n = m$, to facilitate and obtain more reliable associations.

1 DBSCAN

As proposed by LiDARSceneFlow [21] and Rigid3DSceneFlow [26], the DBSCAN [22] algorithm is used. DBSCAN is a density-based clustering algorithm that uses a simple minimum density level estimation based on two user-defined parameters: k , a threshold for the number of neighbors, and ε , the radius within the neighbors are queried. The intuition of DBSCAN is to find the areas that can be seen as 'islands', which satisfy this minimum density and are separated by areas of lower density, which can be seen as the 'sea' level [52].

We now introduce some definitions from [16] to formalize the intuition of clusters. Given a set of n points $\mathbf{X} = \{x_1, x_2, \dots, x_n\}$, let D be a $n \times n$ matrix containing the pairwise distances $d(\mathbf{x}_p, \mathbf{x}_q)$, for a given distance metric $d(\cdot, \cdot)$.

Definition 1. A point $\mathbf{x}_p$ is called a *core* point w.r.t. ε and k if its ε -neighborhood, $\mathbf{N}_\varepsilon(\mathbf{x}_p) = \{\mathbf{x} \in \mathbf{X} \mid d(\mathbf{x}, \mathbf{x}_p) \leq \varepsilon\}$, contains at least k points, i.e. if $|\mathbf{N}_\varepsilon(\mathbf{x}_p)| \geq k$, where $|\cdot|$ denotes the cardinality.

Definition 2. A point $\mathbf{x}_p$ is called a *border* point if it is not a core point but is contained in the ε -neighborhood of a core point, i.e. if $\mathbf{x}_p \in \mathbf{N}_\varepsilon(\mathbf{x}_q)$, where $\mathbf{x}_q$ is a core point.

Definition 3. Two core points $\mathbf{x}_p$ and $\mathbf{x}_q$ are ε -*reachable* if $\mathbf{x}_p \in \mathbf{N}_\varepsilon(\mathbf{x}_q)$ and $\mathbf{x}_q \in \mathbf{N}_\varepsilon(\mathbf{x}_p)$.

Definition 4. Two core points $\mathbf{x}_p$ and $\mathbf{x}_q$ are *density-connected* if they are directly or transitively ε -*reachable*.

Definition 5. A *Cluster* $\mathbf{C}$ is a non-empty maximal subset of $\mathbf{X}$ s.t. every pair of points in $\mathbf{C}$ is density-connected.

Definition 6. A point $\mathbf{x}_p$ is considered as *noise* if it does not belong to any cluster, i.e. $\text{noise} = \{\mathbf{x} \in \mathbf{X} \mid \forall i : \mathbf{x} \notin \mathbf{C}_i\}$

Fig. 4.1 illustrates the concepts just defined for $k = 4$ and where the radius ε is indicated by the circles. A is a core point, whereas B and C are border points, and N is a noise point. Arrows indicate direct ε -reachability. B and C are density-connected because they are both transitively ε -reachable through A. N is not ε -reachable and considered as noise [52]. From those concepts, a simple algorithm that linearly scans

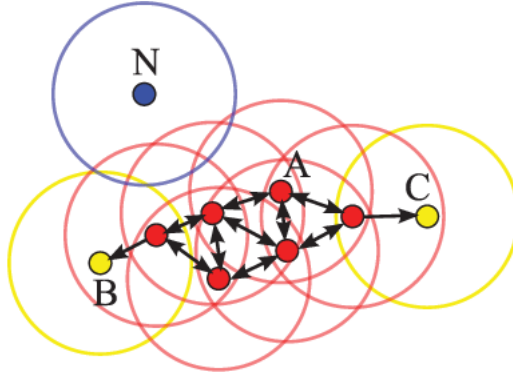


Figure 4.1: **DBSCAN Clustering model** [52]

points that have not been labeled yet can be designed. If a point is non-core, then it is assigned to noise, otherwise, a new cluster is formed and its neighbors are iteratively expanded and added to the cluster. Points that were assigned to noise can be reassigned to a cluster. Alg. 3 gives an abstraction of the DBSCAN algorithm [52]. To construct the neighborhood of each point, a range query operation has to be executed, leading to a total complexity $\Theta(n^2 \cdot d)$, where the dimensionality d is 3 for

Algorithm 3: Abstract DBSCAN Algorithm

```
1 Compute neighbors of each point and identify core points ;    // Identify
   core points
2 Join neighboring core points into clusters ;                  // Assign core points
3 foreach non-core point do
4   | Add to a neighboring core point if possible ; // Assign border points
5   | Otherwise, add to noise ;                          // Assign noise points
```

the Euclidean space. The authors of the original paper [22] claim that the average time complexity is $O(n \cdot \log n)$ if an efficient data structure such as an R*-tree is used for the range query operation and if ϵ is not too large. The paper [23] contradicts this result and proves that no universal index can guarantee an ϵ -range search in $O(\log n)$ and has a lower-bound of $\Omega(n^{1/3})$, meaning the lower-bound of DBSCAN is $\Omega(n^{4/3})$ when $d \geq 3$ and its worst-time complexity is still $O(n^2 \cdot d)$.

While DBSCAN can find clusters of arbitrary shape, it can struggle to find clusters with varying densities. k and ϵ define a fixed minimum density threshold that is assumed to be homogeneous across the clusters. A higher k and a lower ϵ indicate a higher density needed to form a cluster, and conversely [1]. HDBSCAN* [16] is a hierarchical density-based algorithm based on DBSCAN*, which is a simplified version of DBSCAN where the notion of border points disappears and which can be useful for dealing with clusters of varying densities. Instead of relying on a fixed ϵ value, HDBSCAN* builds a hierarchy of clusters for all possible ϵ values, from 0 to ∞ , and uses a minimum spanning tree computed with Prim’s algorithm. The MST computation is a limiting factor as its time complexity is $O(n^2)$ using a list search instead of a heap as the number of edges $E = V^2$, where $V = n$ is the number of vertex or points.

In this thesis, only the DBSCAN algorithm is considered as it can still be accelerated using a reasonable ϵ value and indexes for the range query. Furthermore, if we make the hypothesis that the ground segmentation results in clusters well separated in space, there is no need for a more advanced method for clustering. The implementation of RAPIDS cuML [9] compatible with GPU is used in this work.

2 Cluster association

2.1 Point cloud descriptor

The first step towards cluster association is to retrieve a descriptor, or signature, for each cluster. There exist local-based descriptors [29] that encode local information for each point. Local reference frame-based methods compute a local reference frame (LRF) given a query point and its neighborhood. The local region is then divided into areas using the local reference frame, in which spatial distributions or geometric attributes are computed and accumulated into a descriptor. Local reference axis-based (LRA) methods work in the same manner, dividing the space

according to a single axis. Non-local reference frame-based methods do not use any axis but rely on spatial and geometric information such as the eigenvalues from the local covariance matrix, normal vectors, or the distribution of points within each cube of a 3D grid.

Global descriptors encode information about the whole point cloud and are more interesting for our case. Geometry-based solutions such as VFH [49] (or its extended versions CVFH [13]) rely on the local-based PFH [51] and FPFH [48] descriptors, that compute angles given pair of points' reference frames and normal vectors, for local regions. VFH [49] bins the components (angles) into sub-histograms and concatenates them into a single histogram, also adding a viewpoint feature histogram, making it variant to the viewpoint.

For our cluster association, we use the GASD [20] (Globally aligned Spatial Distribution) descriptor, which is a spatial distribution-based method. The authors claim to be around $5\times$ faster than VFH [49], and more than $100\times$ faster than CVFH [13]. Another advantage is that the color can be used, further increasing its discriminative power.

GASD

Given the set of points P , its centroid $\bar{\mathbf{P}}$ is first computed. Similar to the normal computation in Chapter 3 Section 1.3, the covariance matrix $\mathbb{C} \in \mathbb{R}^{3 \times 3}$ from P and $\bar{\mathbf{P}}$ is obtained, giving the the eigenvectors $\vec{v}_0, \vec{v}_1, \vec{v}_2$ corresponding to the eigenvalues $\lambda_0 \leq \lambda_1 \leq \lambda_2$. The eigenvectors are the x, y, z axis, forming a global reference frame, as illustrated in Fig. 4.2a. A rigid transformation $\mathbf{T} \in \mathbb{R}^{4 \times 4}$ can be computed to align the global reference frame with the canonical basis, making the descriptor pose invariant. After that, a grid of size $m_s \times m_s \times m_s$ centered at the origin of the frame divides the space, as shown in Fig. 4.2b. The shape descriptor simply counts the number of points in each cube, forming a histogram with $m_s \times m_s \times m_s$ bins. Similarly, the color descriptor is obtained from a grid of cubes given the resolution m_c . In each cube, the HSV space is divided into l bins, and the hue values are accumulated, forming a color histogram with $m_c \times m_c \times m_c \times l$ bins. The shape histogram and color histogram are finally concatenated to give a descriptor of length 984 when using $m_s=6$, $m_c=4$ and $l=12$, as proposed by the paper [20].

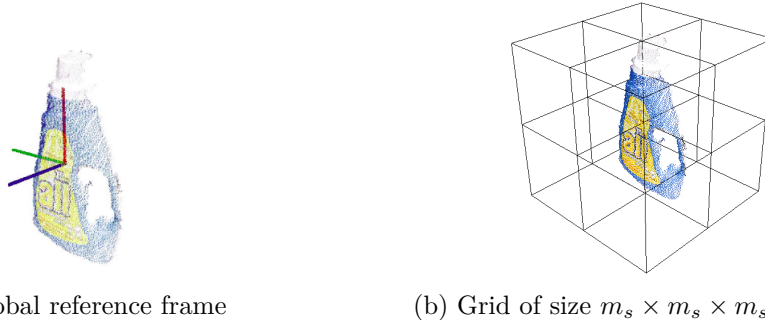


Figure 4.2: **GASD descriptor** [20]. The global reference frame (left) and the grid used for the shape and color descriptors (right).

2.2 Comparing histograms

To associate clusters between them, we want to get a measure of similarity or dissimilarity between their histograms [17]. Let $X = \{x_0, x_1, \dots, x_{b-1}\}$ be the set of b measurements or bins of the histogram. Given a set of n elements whose measurements/bins are $A = \{a_1, a_2, \dots, a_n\}$ where $a_j \in X$, then the histogram $H(A) = [H_0(A), H_1(A), \dots, H_{b-1}(A)]$ is defined as:

$$H_i(A) = \sum_{j=1}^n c_{ij}, \quad c_{ij} = \begin{cases} 1 & \text{if } a_j = x_i, \\ 0 & \text{otherwise.} \end{cases} \quad (4.1)$$

In our case, the histogram is composed of two sub-histograms where the measurements are the 3D spatial cubes for the shape part of the histogram and the 3D spatial cubes combined with the HSV space partitions for the color part of the histogram. The n elements are the points from the cluster, which are associated with two measurements instead of one. In this work, we consider the following measures between histograms [17] for future experiments:

$$Euclidean = D_{L_2}(H(A), H(B)) = \sqrt{\sum_{i=0}^{b-1} (H_i(A) - H_i(B))^2} \quad (4.2)$$

$$Intersection = D_{\cap}(H(A), H(B)) = \frac{\sum_{i=0}^{b-1} \min(H_i(A), H_i(B))}{\sum_{i=0}^{b-1} H_i(A)} \quad (4.3)$$

$$Bhattacharyya = D_{bhat}(H(A), H(B)) = -\log \sum_{i=0}^{b-1} \sqrt{P_i(A)P_i(B)} \quad (4.4)$$

where $P_i(A)$ and $P_i(B)$ are the probabilities for measurement x_i . The *Euclidean* and *Bhattacharyya* are distances or dissimilarity measures while the *Intersection* measures the similarity between histograms and they all have a complexity in $\Theta(b)$. Although there exist many more measures to compare histograms or probability density functions, we only evaluate our methods on these rather simple distance metrics. A more thorough analysis would be required to find the best measure for comparing histograms produced by the GASD descriptor.

2.3 Association process

Putting all the pieces together, we can design an algorithm further classifying the non-ground points as static or moving. First, the source non-ground points NG_s and the target non-ground points NG_t are clustered using the DBSCAN [22] algorithm, giving the source clusters $C_s = \{c_{s1}, c_{s2}, \dots, c_{sn}\}$ and the target clusters $C_t = \{c_{t1}, c_{t2}, \dots, c_{tm}\}$. Then, for each cluster in C_s and C_t respectively, the source GASD [20] and target GASD descriptors are computed. The distances/similarities between every pair of source and target histograms are then computed and sorted in increasing/decreasing order according to the distances/similarities.

The sorted list of distances/similarities is then scanned and each cluster in C_s is associated with its closest/most similar cluster in C_t . The same process is done for

each cluster C_t , ensuring some backward reliability. Only pairs of clusters associated together both in the forward and backward way are considered for removal, otherwise, they are automatically classified as moving and fed to the final network.

3 Classification

Algorithm 4: Moving vs. Static object classification

Input: source and target non-ground points NG_s and NG_t , a distance threshold D_{thresh} and a ratio threshold R_{thresh} , the number of neighbors k and the radius ϵ

Output: Two masks $\mathcal{M}_s = \{m_{si} \in \{0, 1\}\}_{i=1}^{|C_s|}$ and $\mathcal{M}_t = \{m_{tj} \in \{0, 1\}\}_{j=1}^{|C_t|}$ where $m_i = 1$ if the cluster is static and $m_i = 0$ if the cluster is moving

- 1: $C_s, C_t \leftarrow \text{DBSCAN}(NG_s, k, \epsilon), \text{DBSCAN}(NG_t, k, \epsilon)$
- 2: $GASD_s, GASD_t \leftarrow \text{computeGASD}(C_s), \text{computeGASD}(C_t)$
- 3: $HistDistances \leftarrow \emptyset$
- 4: **for** $i = 1$ **to** $|C_s|$ **do**
- 5: **for** $j = 1$ **to** $|C_t|$ **do**
- 6: $histDist \leftarrow \text{computeHistDistance}(GASD_s[i], GASD_t[j])$
- 7: $HistDistances.push((i, j, histDist))$
- 8: **end for**
- 9: **end for**
- 10: $HistDistances \leftarrow \text{sortByDistance}(HistDistances)$
- 11: $reliableAssociations \leftarrow \text{computeAssociations}(HistDistances)$
- 12: $\mathcal{M}_s, \mathcal{M}_t \leftarrow \{0\}_{i=1}^{|C_s|}, \{0\}_{j=1}^{|C_t|}$
- 13: **for all** (i, j) **in** $reliableAssociations$ **do**
- 14: $ratio \leftarrow |1 - \frac{|c_{si}|}{|c_{tj}|}|$
- 15: $clusterDist \leftarrow \text{computeClusterDist}(c_{si}, c_{tj})$
- 16: **if** $ratio < R_{thresh}$ **and** $clusterDist < D_{thresh}$ **then**
- 17: $\mathcal{M}_s[i], \mathcal{M}_t[j] \leftarrow 1, 1$
- 18: **end if**
- 19: **end for**
- 20: **return** $\mathcal{M}_s, \mathcal{M}_t$

Finally, for each pair of associated clusters (c_{si}, c_{tj}) , we consider the following distance metrics:

$$chamfer(c_{si}, c_{tj}) = \frac{1}{n} \sum_{x \in c_{si}} \|x - \text{NN}(x, c_{tj})\|_2 + \frac{1}{m} \sum_{x \in c_{tj}} \|x - \text{NN}(x, c_{si})\|_2 \quad (4.5)$$

$$L_{2-\text{centroid}} = \|\bar{c}_{si} - \bar{c}_{tj}\|_2 \quad (4.6)$$

where $\text{NN}(x, c)$ is the nearest neighbor function, n and m are the number of points in the source and target cluster, and $\bar{c}$ is the centroid of the cluster c . For the chamfer distance, the nearest neighbor of each point has to be computed, which can be done in $O(\log n)$ using a Kd-tree, leading to a total complexity in $O(n \log m + m \log n)$.

Therefore, we also investigate a simpler distance metric taking only into account the two centroids of the clusters, which can be obtained in $O(m + n)$. In the end, we classify the pairs of clusters as static if their distance is below a user-defined threshold, D_{thresh} . As DBSCAN can produce different clusterings in both point clouds, we add an additional threshold, R_{thresh} , stipulating that the absolute ratio (i.e. $|1 - \frac{n}{m}|$) between the two cluster sizes cannot be above a certain percentage, otherwise leading to wrong associations and classifications.

Alg. 4 gives a pseudo-code describing the algorithm. The part of the algorithm that computes the cluster associations given the list of histogram distances is abstracted for the sake of clarity. The algorithm is implemented in C++, leveraging the PCL library [50] as before.

Chapter 5

Scene flow estimation

Given the points classified as moving by the ground segmentation followed by the moving object classification, we want to make the final scene flow prediction for these points, while ground points and static points are assigned a zero flow $\vec{0}$. This chapter presents the network used for feature learning and for the final prediction of scene flow along with all its components and layers. A new feature learning and downsampling method [36] is explored to improve the performance and runtime of the prediction, and plugged before the decoding part of an existing network [19]. As discussed in Chapter 1, the scene flow estimation usually consists of three steps: feature learning and downsampling, point information fusion and matching, and finally generating and refining the flow.

1 Feature learning and downsampling

The most used downsampling algorithm is the farthest point sampling (FPS) while kNN or ball query along with PointNet [44] aggregation are the most used approaches for learning new features. This sampling, grouping, and aggregating scheme is repeated multiple times, each time reducing the size of the point cloud while increasing the receptive field and number of features of each point. However, these components present major drawbacks.

Festa [58] and PSNet [36] argue that FPS is unstable and produces different downsampling from one execution to another due to the randomly selected starting points. These operations rely on Euclidean distances only and do not take into account the features of the points, reducing the effectiveness of the sampling. Furthermore, they give the same results no matter the task, the architecture, or the loss function as there are no learnable parameters neither in FPS nor in kNN. Last but not least, these operations are computationally expensive. FPS does not scale well with the number of points, in addition to being an iterative procedure that requires s iterations, s being the number of sampled points, preventing the parallelization of the algorithm. Similarly, a range search or a kNN can become quite expensive when the number of points grows, as explained in Chapter 4 Section 1.

In this thesis, one of the objectives is to improve the execution time and quality of feature learning for scene flow prediction using the set abstraction of PSNet [36] that employs a learning-based sampling and grouping layer. PSNet [36] replaces FPS and kNN, using a different approach. The sampling of the s points and grouping of their neighbors is done simultaneously using MLPs, facilitating the parallelization of these operations and improving their efficiency. By mapping the spatial coordinates of the points into a s -dimensional space, one can obtain the points' probabilities to belong to each of the s local regions. Each region is then composed of the points with the highest probabilities and the points are effectively grouped and sampled at the same time.

2 Predicting the scene flow

Following the PSNet [36] set abstractions for feature learning, the subsequent steps involve computing an initial flow and refining it. In this work, we make use of the fast hierarchical network from FH-Net [19] to predict scene flow vectors at different resolutions. The authors of FH-Net [19] claim to achieve real-time inference with a speed well above 10Hz, the frequency at which LiDAR point clouds are scanned. The reason why they can reach such high speeds is because they do not rely on complex matching or feature decoding mechanisms, which are computationally expensive.

FH-Net [19] is composed of three layers, namely the Trans-flow layer, the Trans-up layer, and the Flow propagation layer. Given the original resolution point cloud S_0 with N points and its two downsampled versions S_1 and S_2 containing respectively N_1 and N_2 points with $N_2 < N_1 < N$, the layers predict a scene flow at each resolution, in a coarse to fine fashion, leading to the raw resolution scene flow at the last layer. The coarse flows of the Trans-Flow layer are obtained by aggregating the offsets of neighboring target points using the attention mechanism. The denser flows of the Trans-up layer are simply obtained by aggregating neighboring coarse flows resulting from the previous layer, again using the attention mechanism. The Flow propagation layer outputs the raw resolution flow by interpolating the denser flows with a 3-Nearest neighbors interpolation and refining them with an MLP. The layers of PSNet [36] and FH-Net [19] are now introduced in more detail.

3 PSNet

In this section, we introduce the main components of PSNet [36], namely the Spatial Features Transform Function followed by the sampling and grouping of points, and finally the aggregation of neighboring features.

SFTF

given a point cloud $P = \{x_i \in \mathbb{R}^d\}_{i=1}^m$, it first starts by mapping the spatial features of each point to a higher dimension using its Spatial Features Transform Function (SFTF) defined as $T(P) : \mathbb{R}^{m \times d} \implies \mathbb{R}^{m \times s}$, where $s < m$ is the number of points

sampled from P or the number of local regions. Shared MLPs implemented as 1×1 conv layers transform the points independently and can be denoted in matrix notation as:

$$F = T(P) \quad (5.1)$$

where $F \in \mathbb{R}^{m \times s}$ and $f_i \in \mathbb{R}^s$ is the vector of correlations degrees between the point x_i and the s local regions. Using the SFTF, PSNet [36] does not rely only on the Euclidean distance between points, but also on their features. The sigmoid function is then applied to F :

$$Q = \text{sigmoid}(F) \quad (5.2)$$

where $Q \in \mathbb{R}^{m \times s}$ and its columns $e_j \in \mathbb{R}^m$, $j=1, \dots, s$, give the probabilities that each point x_i belongs to the local region l_j . The indices of the points included in each local region l_j and their centroids, or subsampled points *sub*, are obtained by simply applying *argtop*:

$$\begin{aligned} \text{indices}_j &= \text{argtop}_n(e_j) \\ \text{sub}_j &= \text{argtop}_1(e_j) \end{aligned} \quad (5.3)$$

where sub_j is the index of the centroid point of local region l_j and indices_j contains the n indices of the points located in l_j . Using these indices, one can easily retrieve the downsampled points and their associated neighborhoods. In the end, the SFTF performs both the sampling and grouping simultaneously, and its different phases are illustrated in Fig. 5.1 left, where the matrices F and Q are shown along with the indices sub_j of the downsampled points in red, and the associated indices indices_j of their local regions in purple.

In practice, the *argtop* is not a differentiable function and is approximated during training by the Gumbel-Softmax function applied on the matrix Q . It outputs a sampling matrix $\tilde{Q}$ that can be used to directly index the subsampled points with a matrix multiplication, making the operation differentiable. As you may have noticed in Fig. 5.1, the input points to the SFTF contain two additional features, a polar angle θ and an azimuthal angle ρ , which are used to avoid symmetrical points in a symmetrical point cloud to result in similar feature vectors f_i .

Aggregation

Using the n indices of the local regions, the corresponding points $P_j = \{x_{ij}\}_{i=1}^n$ of l_j are retrieved and the traditional PointNet [44] aggregation defined in Eq. 1.1 is applied, effectively doubling the number of features of the neighboring points. It is then followed by a channel-wise MAX pooling operation, aggregating the neighboring features. The aggregation phase of PSNet is depicted in Fig. 5.1 right.

4 FH-Net

In this section, the components of the Trans-Flow, Trans-up, and Flow propagation layers from FH-Net [19] are presented with more details.

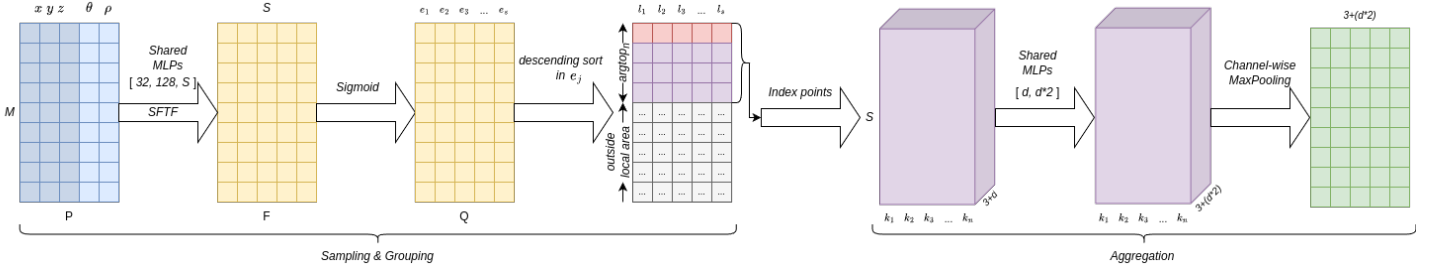


Figure 5.1: **Set abstraction of PSNet** [52]. Sampling and grouping operations of PSNet followed by the classical aggregation of local regions, as in PointNet [44].

Trans-flow layer

For each point $s_i = (x_i, u_i)$ in S_2 , where $x_i \in \mathbb{R}^3$ are its spatial coordinates and $u_i \in \mathbb{R}^d$ its features, the first layer starts by finding its neighbors in the downsampled target point cloud T_2 :

$$T_2^i = \{t_j = (y_j, v_j) \mid \|y_j - x_i\|_2 < R\}_{j=1}^K \quad (5.4)$$

where K is the number of neighbors and R the radius. The transformer attention mechanism is then used and attention scores between the points s_i and their neighbors t_j are computed as:

$$\alpha_{ij} = \gamma(\varphi(u_i) - \psi(v_j) + \sigma(y_j - x_i)) \quad (5.5)$$

where φ and ψ are learnable linear transformations, γ is an MLP, and σ the relative position encoding, which will guide the prediction of the flow instead of deep feature decoding or matching that is expensive. Finally, the layer outputs a sparse scene flow $F_2 = \{f_i^2 \in \mathbb{R}^3\}_{i=1}^{N_2}$ by aggregating the neighboring offsets using the attention weights:

$$f_i^2 = \sum_{j=1}^K \rho(\alpha_{ij})(y_j - x_i) \quad (5.6)$$

where ρ is the softmax function. The features u_i of the source points are updated for further usage, aggregating the local context from their neighborhood:

$$u'_i = \text{MAX}_{j=1, \dots, K} \{h(u_i \oplus v_j \oplus \sigma(y_j - x_i))\} \quad (5.7)$$

where u'_i are the enhanced features, $\oplus$ is the concatenation operator, h a learnable linear transformation and MAX the max pooling operator.

Trans-up layer

Given the sparse scene flow F_2 , this layer upsamples it to give a denser scene flow F_1 for all points s_i^1 in S_1 . The same operations used in the Trans-flow layer are applied but with different inputs. For each point s_i^1 , its neighborhood is constructed by searching for neighbors s_j^2 in S_2 instead. Again, attention weights for those neighbors

are computed using Eq. 5.5 and the neighboring flows f_j^2 from the coarser level are aggregated to give the denser scene flow $F_1 = \{f_i^1 \in \mathbb{R}^3\}_{i=1}^{N_1}$:

$$f_i^1 = \sum_{j=1}^K \rho(\alpha_{ij}) f_j^2 \quad (5.8)$$

The features u_i^1 are also updated using Eq. 5.7 and are used in the last Flow propagation layer to refine the final flow. The Trans-flow and the Trans-up layers are illustrated in Fig. 5.2.

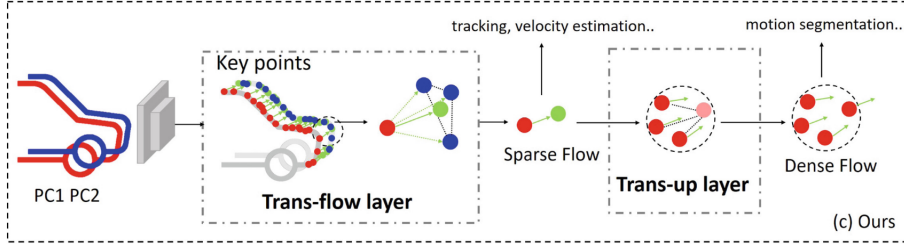


Figure 5.2: **Trans-flow and Trans-up layers from FH-Net [19]**. The Trans-flow layer produces a sparse flow using the target point cloud while the Trans-up layer upsamples this flow using the denser source point cloud, producing a dense flow.

Flow propagation layer

The final layer predicts the raw resolution flow $F_0 = \{f_i^0 \in \mathbb{R}^3\}_{i=1}^N$ using the dense flow F_1 . However, no features are available for the raw resolution points s_i^0 and as a result, no attention weights can be computed. Instead, for each point s_i^0 in S_0 , a simple 3D interpolation using its 3-Nearest neighbors in S_1 is performed, where the weights are inversely proportional to the L_2 distances to the neighbors:

$$f_i^0 = \sum_{j=1,2,3} \frac{1}{\|x_i^0 - x_j^1\|_2} f_j^1 + MLP\left(\sum_{j=1,2,3} \frac{1}{\|x_i^0 - x_j^1\|_2} u_j^1\right) \quad (5.9)$$

The enhanced features from the neighboring points at the previous layer are also aggregated and fed to an MLP, learning a residual flow that is added to the interpolated flow, giving the final raw resolution flow F_0 .

5 The final network

The final implementation of the network adopts the set abstraction from PSNet [36] for the feature learning, hoping to learn a more effective and more efficient sampling for the scene flow prediction task, whereas FH-Net [19] initially utilizes PointNet++ [45] for feature extraction, relying on FPS for the downsampling. The set abstractions are shared between the two point clouds and the implementation in PyTorch already available at their GitHub repository¹ is used.

¹<https://github.com/lly007/PointStructuringNet/tree/main>

For the prediction part, the 3 layers of FH-Net [19] are used and the final network is depicted in Fig. 5.3. For the neighborhood grouping in the Trans-flow layer, ball query is used to ease the search for longer correspondences, allowing the prediction of larger motions. In the Trans-up and Flow propagation layers, as we are only interpolating for the source point cloud, a kNN is used as the motions of distant points are less interesting with regard to the query point. FH-Net code is not publicly available but the layers are implemented in PyTorch and integrated with the implementation of PSNet [36]. In both PSNet and FH-Net, the shared MLPs are implemented as one or multiple Linear/BatchNorm/ReLU blocks. The learnable linear transformations are simply a Linear/BatchNorm block. More details about the implementation of the FH-Net layers and the operations on tensors are provided in Appendix 3.

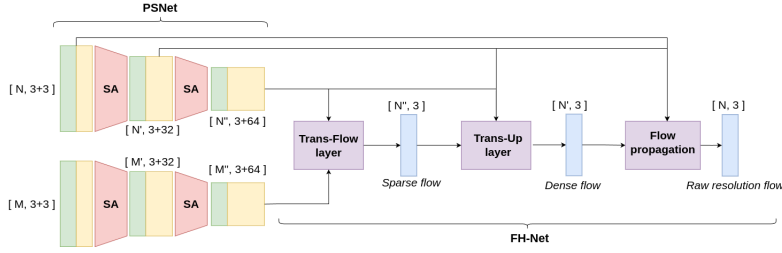


Figure 5.3: **Final network.** PSNet set abstractions for the feature learning and FH-Net Trans-flow, Trans-up, and Flow propagation layers to predict the flow.

Table 5.1 shows the parameters of the network. Starting from 16384 points, the two set abstractions from PSNet [36] downsample the point cloud to 2048 then 1024 points using each time $K=16$, the number of neighbors contained in each local region for aggregation. The Trans-Flow layer predicts the flow for 1024 points, using a radius $R=6m$ and searching for $K=32$ neighbors in this region. The Trans-up layer upsamples and predicts the flows for 2048 points using $K=16$. Finally, the Flow propagation layer outputs the flows at the raw resolution with a 3-Nearest neighbors interpolation. All MLPs and Linear transformations layers' widths are indicated, including the input and output dimensions. In total, the network contains 793K learnable parameters, which is less than most architectures, as depicted in Chapter 1 Section 4.

Layer	#points	K	MLPs width
SA1	2048	16	SFTF=[5,32,128,2048], PointNet=[3,16,16,32]
SA2	1024	16	SFTF=[5,32,128,1024], PointNet=[32,64,64,128]
TransFlow	1024	32	$\varphi=[128,128], \psi=[128,128], \sigma=[3,64,128], \gamma=[128,512,1], h=[384,128]$
TransUp	2048	16	$\varphi=[64,64], \psi=[128,64], \sigma=[3,64,64], \gamma=[64,256,1], h=[256,256]$
FlowPropagation	16384	3	$MLP=[256,256,128,3]$
#Parameters in total: 793K			

Table 5.1: **Network architecture.** The number of points sampled by the two set abstractions are indicated, as well as the number of points for which the flow is predicted in the FH-Net [19] layers. The number of neighbors K taken into account at each layer is indicated as well as the MLPs' widths at each layer.

Chapter 6

Experiments and Results

This chapter presents the experiments conducted for the ground segmentation, the ego-motion refinement, the moving object classification, and the final scene flow prediction along with their results. All experiments except for the DBSCAN [22] algorithm and the final scene flow prediction using the network are conducted on an AMD Ryzen 7 PRO 6850U CPU with 8 cores and 16 threads at 2.7GHz. The DBSCAN experiments and the network training are conducted on a Tesla V100 GPU using the CÉCI clusters providing access to GPUs [3].

1 Ground segmentation

In this section, experimental results for the ground segmentation algorithm from Chapter 3 are presented. To evaluate our proposed method, one needs point-level ground truth segmentation labels. Leveraging the KITTI vision benchmark suite [25], [11] annotated the 200 publicly available images with pixel-level semantic labels, as shown in Fig. 6.1a,b. The point-level labels can simply be retrieved when lifting the image to a 3D point cloud, similar to the point-level RGB information.

Since we are interested in binary labels (ground and non-ground), labels for different objects need to be grouped together. The labels *terrain*, *rail track*, *parking*, *sidewalk*, *road*, *ground* are merged together to form a single *ground* label whereas all other labels are considered as the *non-ground* label, as illustrated in Fig. 6.1c.

1.1 Evaluation metrics

Given the predicted labels $\text{ground}=1$, $\text{non-ground}=0$, and the ground truth labels, the confusion matrix can be computed, i.e. the true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN). Our method acts as a first filter for the ground points, therefore false positives should be avoided as much as possible (classifying non-ground as ground), otherwise no motion can be estimated later on.

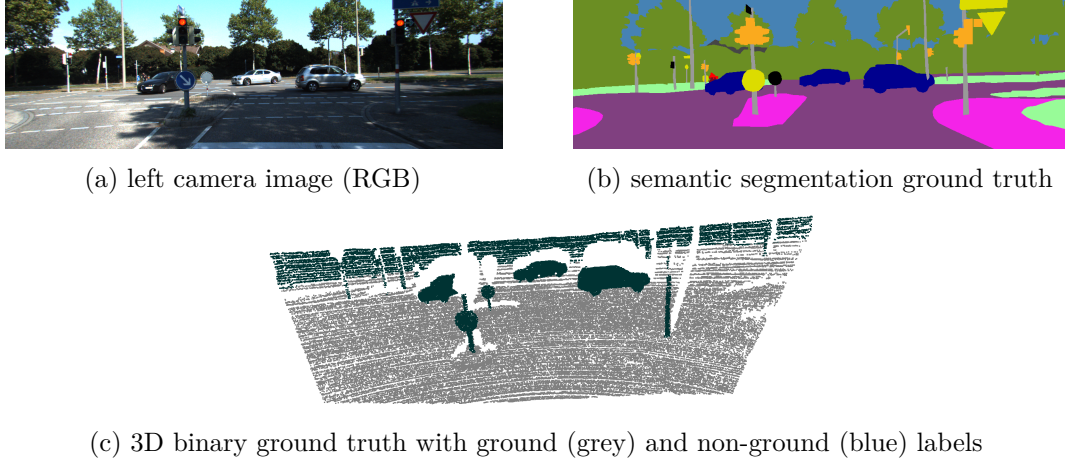


Figure 6.1: **Semantic segmentation ground truth [11]**. The color image (a) with its corresponding ground truth (b). The lifted point cloud with the binary ground truth segmentation labels is shown in (c).

With this in mind, the *recall* and *specificity* defined as follows can be used [2]:

$$\begin{aligned}
 recall &= \frac{TP}{TP + FN} \\
 specificity &= \frac{TN}{TN + FP}
 \end{aligned} \tag{6.1}$$

where the *recall* is the proportion of ground points well classified among all ground points and the *specificity* is the proportion of non-ground points well classified among all non-ground points. Minimizing the number of false positives amounts to maximizing the *specificity*. Meanwhile, the *recall* should be kept as high as possible, minimizing the number of false negatives and maximizing the number of true positives.

1.2 Results

Our ground segmentation algorithm presented in Chapter 3 Section 1 includes the following parameters that need to be tuned: the voxel size s , the number of neighbors K , the angle threshold α and the height threshold $\mathcal{H}$. The following values for each parameter are tested:

- s (m): [0.05, 0.10, 0.15, 0.20, 0.25, 0.30]
- K : [4, 8, 16, 32]
- $\mathcal{H}$ (m): [0.2, 0.3, 0.4, 0.5, 0.6]
- α (°): [5, 10, 15, 20]

leading to 480 combinations. The results are shown in Table 6.1 where only the best solutions with respect to the *recall* and where the *specificity* ≥ 0.99 are kept. The *recall* and *specificity* are averaged over the 142 frames of the dataset.

K	α ($^\circ$)	s (m)	$\mathcal{H}$ (m)	$recall \uparrow$	$specificity \uparrow$
32	10	0.15	0.5	0.8266	0.9901
	10	0.20	0.4	0.8209	0.9901
	10	0.10	0.6	0.8231	0.9901
	10	0.10	0.5	0.8172	0.9907
16	10	0.30	0.4	0.8184	0.9901
	10	0.25	0.4	0.8123	0.9903
	10	0.15	0.6	0.8089	0.9901
	10	0.20	0.4	0.8068	0.9905
8	10	0.25	0.6	0.7899	0.9901
	10	0.30	0.4	0.7892	0.9905

Table 6.1: **Ground segmentation performance.** best solutions w.r.t the *recall* where *specificity* ≥ 0.99 , grouped by K .

The best angle threshold α for all solutions is 10° . The height threshold $\mathcal{H}$ ranges from 0.4m and 0.6m. Although these parameters give the best results, a small angle can fail to classify sloped terrain or sharp changes such as a sidewalk curb, as shown in Fig. 6.2. A higher angle combined with a lower height threshold could mitigate this issue, in turn sacrificing the proper classification of higher grounds or small ground-level objects. A more advanced technique would be required to handle such cases.

Intuitively, a larger voxel size s is usually associated with a smaller number of neighbors K , otherwise leading to less precise normal and plane approximations. Conversely, a smaller voxel size s allows for a larger number of neighbors K , resulting in more precise normal approximations. Another reason why a small voxel size s is also associated with larger K values is due to the distribution of the points on some frames. As depicted in Fig. 6.3, ground points are usually distributed in concentric circles and if the gap between two adjacent circles is too large, a small K can fail to sample neighbors in adjacent circles, leading to inaccurate normal estimations.

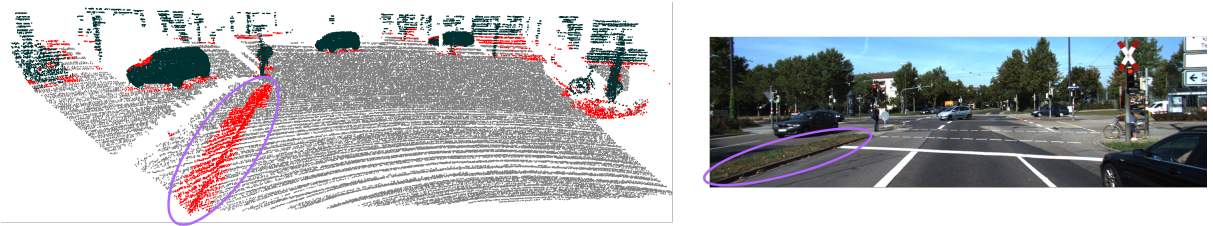


Figure 6.2: **Classification error.** The classification error (FP and FN) is shown in red. The misclassified sidewalk curb is circled in purple.

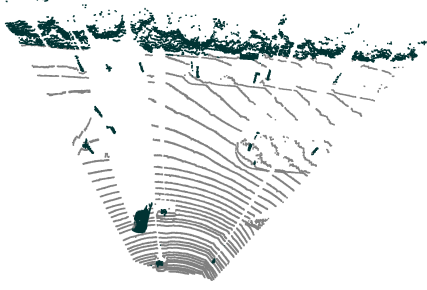


Figure 6.3: **Concentric distribution.** Typical ground point distribution in concentric circles.

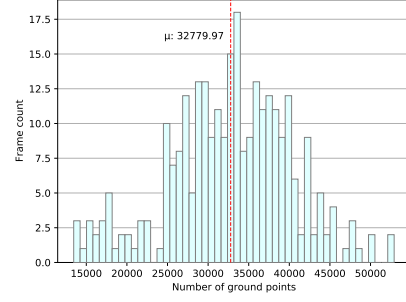


Figure 6.4: **Number of ground points distribution.** The mean μ is indicated in red.

To choose the parameters s and K , Fig. 6.5 (left) compares the runtime between different pairs of s, K values from Table 6.1, ignoring α and $\mathcal{H}$ since they do not affect the computation time. The runtimes are obtained with 8 threads, the number of threads above which no improvement is observed for the best solution, as shown in Fig. 6.5 (right). From 2 threads up to 16, only the mask computation benefits from adding more threads, whereas the normal computation is not faster and the voxel-grid filtering simply does not utilize multi-threading.

The pair with the largest $s=0.3$ and the smaller $K=8$ is unsurprisingly the fastest one. A large value s implies a faster voxel-grid filtering and fewer points whose normal needs to be computed. A smaller K value also implies fewer neighbors to search for, and thus faster normal computations. For a fixed value s , a larger K thus increases the normal computation. Decreasing the voxel size s slightly increases the computation time of the mask, as the computation of the intermediate down-sampled mask includes more points.

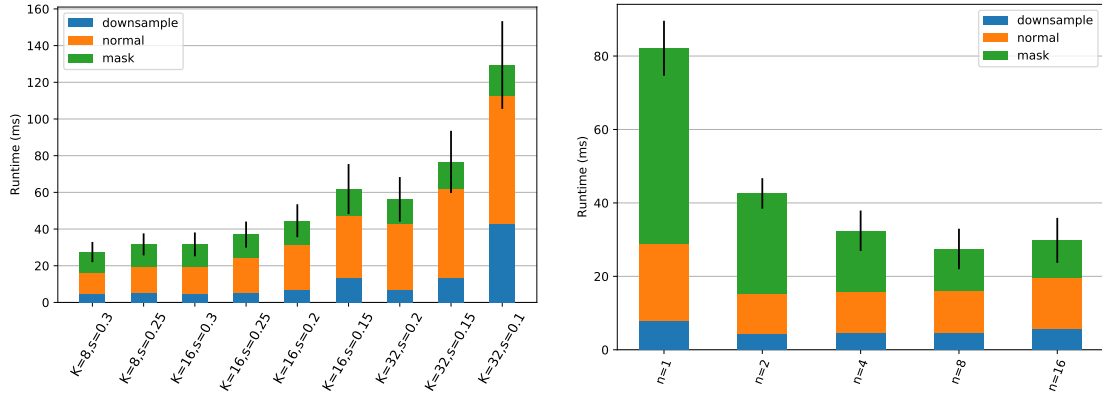


Figure 6.5: **Runtime comparison.** Runtime in ms for different K, s values using 8 threads and averaged over both point clouds of the 142 frames (5 runs each) on the left and runtime for different numbers of threads for the best solution on the right, showing the different steps of ground segmentation: down-sampling, computing normals and computing the ground mask.

For these reasons, the solution with parameters $K = 8$, $\alpha = 10$, $s = 0.3$, $\mathcal{H} = 0.4$ from Table 6.1 is the one retained, although achieving a lower *recall* score. Its runtime

is ~ 27.5 ms on average, making it suitable for real-time applications. Finally, ~ 32762 points on average are classified as ground in each point cloud, already eliminating around half the points out of the original 65536 points, as shown in Fig. 6.4. It is worth noting that the number of points classified as ground ranges from around 15000 to 50000 depending on the frame.

2 Ego-motion refinement

As explained in Chapter 3, the initial estimation of the ego-motion can be inaccurate in some cases, subsequently leading to erroneous results regarding the true motion of moving points. In this section, leveraging the ground truth flow vectors and the ground truth ground labels, we evaluate the point-to-plane ICP algorithm on the ground points.

2.1 Evaluation metrics

The ground points are always assigned a zero flow $\vec{\mathbf{0}} = (0, 0, 0)^\top$ and by updating the ground truth flow vectors with the refined ego-motion flow, we can use the traditional metrics for the scene flow estimation problem [37]:

- *EPE3D*: $\frac{1}{N} \sum_{i=1}^N \|F_i^{GT} - F_i\|_2$, representing the average L_2 distance between the predicted and ground truth flow over the N points.
- *Acc3DS*: The percentage of points whose *EPE3D* < 0.05 m.
- *Acc3DR*: The percentage of points whose *EPE3D* < 0.1 m.
- *Outliers*: The percentage of points whose *EPE3D* > 0.3 m.

Considering the ego-motion flow $F_{em} = S_w - S$, where S_w represents the warped point cloud obtained using the rigid transformation $C^{2_{t+1}}\mathbf{T}_{C^{2_t}}$ from Eq. 2.17, we can update the ground truth flow, $F_{GT} = F_{GT} - F_{em}$, and compute the metrics on ground points for which the predicted flow is $F = \vec{\mathbf{0}}$, establishing a baseline performance, as depicted in Table 6.2.

2.2 Results

The point-to-plane ICP algorithm includes different parameters that need to be tuned: the voxel size s for the down-sampling, the number of neighbors K to estimate the normals, and the search radius for the correspondences. For each parameter, we consider the following values:

- s (m): $[0.1, 0.2, 0.3]$
- K : $[4, 6, 8, 10, 12, 14, 16]$
- $radius$ (m): $[0.04, 0.08, 0.12, 0.16]$

The maximum number of iterations is set to 10 and the transformation ϵ , i.e. the maximum allowable translation squared difference between two consecutive transformations, is set to 0.001 to consider the algorithm as having converged.

	$EPE3D \downarrow$ (m)	$Acc3DS \uparrow$	$Acc3DR \uparrow$	$Outliers \downarrow$
Before ICP	0.0565	0.5050	0.9433	0.0047

Table 6.2: **Baseline performance.** Evaluation metrics on ground points before the point-to-plane ICP refinement, averaged on all frames.

Table 6.3 shows the best results for each voxel size s . We notice that the best results are obtained when using the smallest voxel size $s=0.1$, resulting in a less severe down-sampling and more ground points being present. In combination with a larger K for normal approximations, it allows for more precise computations and a gain in performance.

Solutions for the largest voxel size $s=0.3$ perform somewhat worse with an average $EPE3D$ of ~ 0.015 m and a smaller $Acc3DS$ around $\sim 98\%$, although performing equally well with respect to the $Acc3DR$ and $Outliers$ metrics. As pointed out before, larger voxel sizes are usually combined with a smaller K .

For the *radius* value, one can notice that the value 0.04 is never used, resulting in poor correspondence estimations if the ego-motion is too large. The value 0.16 is used three times while average values of 0.08 and 0.12 are the most used, leading to a better compromise.

s (m)	K	$radius$ (m)	$EPE3D \downarrow$ (m)	$Acc3DS \uparrow$	$Acc3DR \uparrow$	$Outliers \downarrow$
0.1	12	0.08	0.0095	0.9949	0.9951	0.0047
	14	0.08	0.0098	0.9949	0.9951	0.0047
	10	0.12	0.0099	0.9948	0.9951	0.0047
	8	0.16	0.0105	0.9948	0.9951	0.0047
0.2	6	0.12	0.0109	0.9950	0.9951	0.0047
	6	0.16	0.0115	0.9949	0.9951	0.0047
	8	0.08	0.0118	0.9938	0.9949	0.0047
	8	0.12	0.0121	0.9943	0.9950	0.0047
0.3	6	0.08	0.0151	0.9862	0.9946	0.0047
	6	0.12	0.0151	0.9891	0.9946	0.0047
	4	0.08	0.0159	0.9776	0.9943	0.0048
	6	0.16	0.0160	0.9808	0.9933	0.0048

Table 6.3: **Post refinement performances.** Best solutions w.r.t to the $EPE3D$ grouped by voxel size s , averaged on all frames.

The runtime comparison, shown in Fig. 6.6, is taken into account to choose the final parameters. Again, the largest voxel size $s=0.3$ achieves the best computational time, while smaller voxel sizes increase the computational time for the normal approximations and the ICP, as more ground points are present due to milder down-sampling.

For fixed voxel size s and *radius* values, a small increase in K slightly increases the normal computational time. Similarly, for fixed voxel size s and K values, a small increase in the *radius* size slightly increases the ICP computational time, even

though these changes are barely significant.

The runtimes are measured using 2 threads, allowing the down-sampling and normal computations on both point clouds to happen simultaneously. However, the ICP is an iterative process, preventing its multithreading. The final parameters kept are $s=0.3$, $K=6$, and $radius=0.08$, achieving sufficient results while being fast enough for a real-time application with an average runtime of ~ 16.2 ms. At last, we observe that the standard deviation is usually quite large, coming from the fact that the ground segmentation and the subsequent down-sampling can result in a varying number of points as input to the ICP refinement process, as observed in Fig. 6.4.

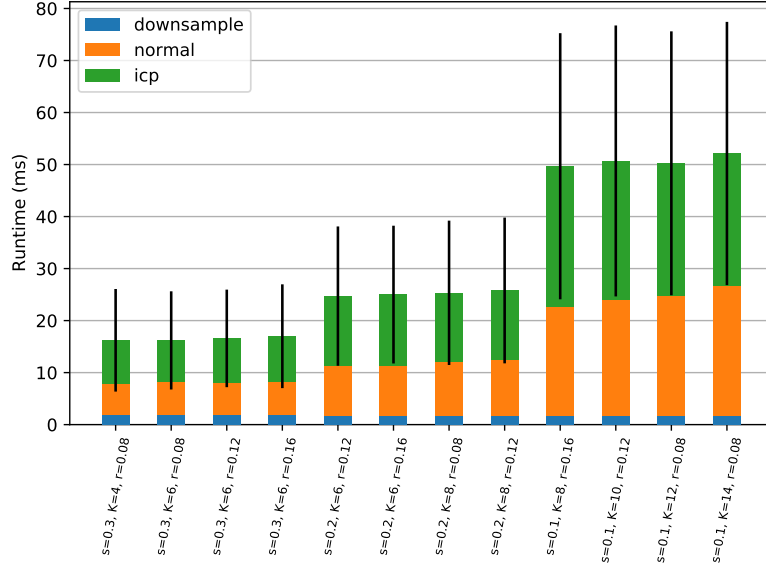


Figure 6.6: **Runtime comparison.** Runtime in ms for different K , s , and $radius$ values using 2 threads and averaged over the 142 frames (5 runs each), showing the different steps of refinement: down-sampling, computing normals, and point-to-plane ICP.

3 Moving object segmentation

In this section, we investigate the results of the DBSCAN [22] clustering, followed by the association of clusters, and the final classification of clusters as *static* or *moving*. The runtimes of these different parts are also reported.

3.1 Clustering metrics

The clusters returned by DBSCAN are important with respect to the next steps, i.e. associating and classifying the clusters. As stated in Chapter 4, we want $|C_s| \simeq |C_t|$, but also $n_s \simeq n_t$, where n_s and n_t are the numbers of points classified as noise in the

source and target non-ground points. There we can introduce two ratio metrics:

$$\begin{aligned} Ratio_{cluster} &= abs\left(1 - \frac{|C_s|}{|C_t|}\right) \\ Ratio_{noise} &= abs\left(1 - \frac{n_s}{n_t}\right) \end{aligned} \tag{6.2}$$

where abs is the absolute value. Preferably, $Ratio_{noise}$ and $Ratio_{cluster}$ should tend to 0, indicating similar clusterings across the two point clouds.

Regarding cluster validation [46], two main methods exist: Internal vs. External validation. External validation relies on prior information or ground truth labels for the clusters, which are not available, while internal validation relies on intrinsic information from the clustered data itself. As clusters should be well separated in space, the silhouette index, taking into account the compactness and the separation between clusters, seems like a natural choice. Given a cluster c_j and its i^{th} point, It is defined as:

$$s(i) = \frac{a(i) - b(i)}{\max(a(i), b(i))} \tag{6.3}$$

where $a(i)$ is the average distance between point i and the other points in c_j and $b(i)$ is the average distance between point i and the points in the closest cluster c_k . $s(i)$ ranges from -1 to 1 and the higher the value, the more likely the point is assigned to the right cluster. The silhouette score is then averaged over all points. Finally, the number of clusters $|C_s|$ and $|C_t|$ returned by DBSCAN is not fixed but depends on the parameters k and ϵ and should ideally not be too large nor too small.

3.2 Association metrics

For the association of clusters, we want the sum of histogram distances/similarities between associated clusters to be minimal/maximal:

$$\arg \min_{\mathcal{A}} \frac{\sum_{(i,j) \in \mathcal{A}} dist(H(c_i), H(c_j))}{|\mathcal{A}|}, \text{ or } \arg \max_{\mathcal{A}} \frac{\sum_{(i,j) \in \mathcal{A}} sim(H(c_i), H(c_j))}{|\mathcal{A}|} \tag{6.4}$$

where $dist$ and sim are a distance or similarity measure between histograms and $\mathcal{A}$ contains the pairs of associated clusters. The sum is normalized by the number of associations for fair comparisons between different associations. We also compare the three distance/similarity measures, i.e. *Euclidean*, *Intersection* and *Bhattacharyya*, for histogram comparison with respect to the final classification. Next, we can look at the proportion of reliable associations that are kept in relation to the minimum number of clusters:

$$Ratio_{\mathcal{A}} = \frac{|\mathcal{A}|}{\min(|C_s|, |C_t|)} \tag{6.5}$$

Intuitively, we want this ratio to be as close to 1 as possible, indicating that all the same pairs are matched in the forward and backward associations.

3.3 Classification metrics

Finally, for the classification, it is desirable to remove as many static points as possible. Unfortunately, *static* and *moving* segmentation labels are not provided but similar to Section 2, we can leverage the ground truth flow vectors and evaluate the classical metrics for scene flow on the newly classified static points for which we predict a zero flow $\vec{0}$. The two distance metrics, i.e. *chamfer* and $L_2\text{-centroid}$, are compared in terms of classification performance and runtime.

3.4 Results

The parameters that need to be tuned for the whole procedure are k and ϵ from DBSCAN [22], defining the number of neighbors and the radius of the ϵ -neighborhood, the histogram comparison measure in the association and finally the cluster distance measure, the distance threshold D_{thresh} and the ratio threshold R_{thresh} between the cluster sizes for the classification part. The following values are tested:

- $k = [6, 12, 24, 48]$
- $\epsilon \text{ (m)} = [0.5, 1, 1.5, 2, 2.5, 3]$
- $HistMeasure = ['Euclidean', 'Intersection', 'Bhattacharyya']$
- $DistMeasure = ['chamfer', 'L_2\text{-centroid}']$
- $D_{thresh} \text{ (m)} = [0.1, 0.2, 0.3]$
- $R_{thresh} = [0.1, 0.2, 0.3]$

leading to a total of 1296 combinations. Table 6.4 shows the metrics of interest for different parameter values for k and ϵ in the DBSCAN algorithm. We notice that small ϵ in combination with increasing values for k tends to produce more noise points. This is explained by the fact that the more you reduce ϵ and increase k , the more dense the neighborhood of a point needs to be considered a core point, leading naturally to more noise. Increasing the radius ϵ while keeping k constant reduces the number of clusters $|C|$ as the minimum density level is lowered, making it easier for clusters to merge, and reducing the number of noise points.

The $Ratio_{noise}$ highly increases when increasing the radius ϵ but at the same time the number of noise points decreases by a great amount. When the number of noise points is smaller, a small absolute difference in the number of noise points between the two point clouds affects more severely $Ratio_{noise}$ although the noise points might be more or less similar. The same effect is observed with $Ratio_{cluster}$ when $|C|$ decreases, albeit less pronounced because the absolute difference between $|C_s|$ and $|C_t|$ when $|C|$ is low is usually smaller.

Silhouette scores tend to increase as the number of clusters $|C|$ drops, which can be intuitively explained. When k and ϵ are too small, it creates a lot of micro clusters close to each other, especially in the background as illustrated in Fig. 6.7a, reducing the inter-cluster distance $b(i)$ and silhouette score. When k and ϵ are too large, fewer but bigger clusters appear as illustrated in Fig. 6.7b, increasing the inter-cluster distance $b(i)$ and silhouette scores, although the intra-cluster distance is increased.

We notice that when $|C|=12.2$ on average for $k=48$ and $\epsilon=3$, separate foreground objects are grouped together as can be observed in Fig. 6.7b, not making it a suitable solution. On the other hand, for $k=6$ and $\epsilon=0.5$ where $|C|=144.44$ on average, foreground objects are well separated but there are too many micro-clusters in the background. Parameters somewhere between these two extremes should balance these issues and offer a better solution, while still maintaining a decent silhouette score.

k	ϵ (m)	$noise$	$Ratio_{noise} \downarrow$	$ C $	$Ratio_{cluster} \downarrow$	$silhouette \uparrow$
6	0.5	536.78	0.086	144.44	0.0755	0.2043
	1.0	139.65	0.142	73.80	0.0700	0.2270
	1.5	65.62	0.2192	47.65	0.0834	0.2411
	2.0	36.86	0.2647	32.63	0.0836	0.2607
	2.5	23.03	0.4183	23.34	0.1123	0.2687
	3.0	218.59	0.5198	17.77	0.109	0.2988
12	0.5	1426.65	0.0772	100.47	0.0809	0.2329
	1.0	382.08	0.1368	57.84	0.0766	0.2719
	1.5	178.82	0.1806	39.05	0.0738	0.281
	2.0	100.33	0.2195	27.65	0.0901	0.2945
	2.5	60.99	0.3024	20.44	0.121	0.2966
	3.0	41.62	0.3942	15.63	0.1199	0.3231
24	0.5	3418.11	0.0489	62.68	0.0988	0.2397
	1.0	974.42	0.0958	43.67	0.0845	0.3254
	1.5	472.98	0.1253	31.09	0.0847	0.3308
	2.0	268.25	0.2054	23.35	0.0889	0.3363
	2.5	166.44	0.2592	17.84	0.1107	0.3409
	3.0	111.1	0.2981	13.74	0.1177	0.3561
48	0.5	6801.3	0.0406	31.62	0.1197	0.2123
	1.0	2316.07	0.0679	29.74	0.091	0.3511
	1.5	1156.35	0.0909	23.93	0.1021	0.3775
	2.0	683.81	0.1186	19.15	0.0796	0.3792
	2.5	441.51	0.1453	15.15	0.1063	0.3762
	3.0	294.33	0.1744	12.2	0.1129	0.3877

Table 6.4: **DBSCAN results.** Metrics of interest for all combinations of DBSCAN parameters k and ϵ , averaged over the 142 frames of the dataset and on both source and target point clouds.

Fig. 6.8 compares the runtime of all combinations of DBSCAN [22] parameters k and ϵ . As explained in Chapter 4 Section 1, the parameter influencing the most the runtime is ϵ , involved in the range query operation, with smaller values reducing the runtime. We notice that the best runtime is $\sim 75\text{ms}$ for $k=48$ and $\epsilon=0.5$, which makes the clustering less suitable for a real-time application and a bottleneck of the pipeline. Again, the results present a large standard deviation coming from the fact that the number of ground points and non-ground points arising from the ground segmentation can largely vary as stated in Section 1 and Section 2.

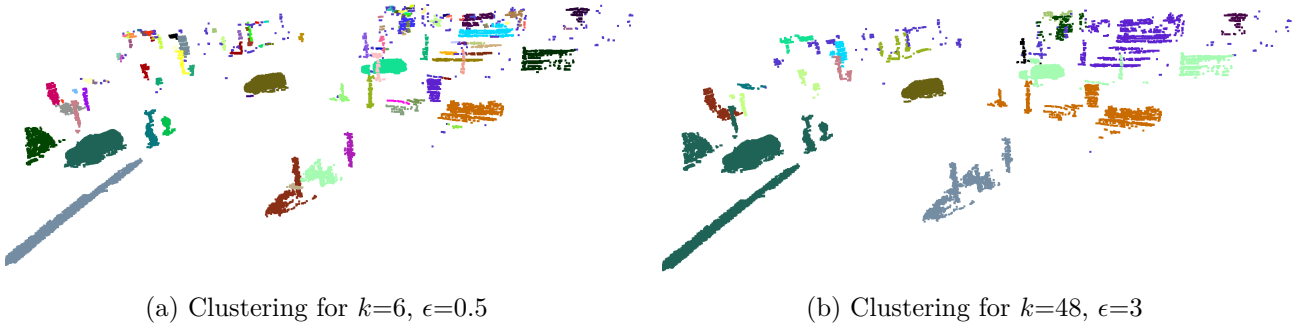


Figure 6.7: **DBSCAN visualization.** Visualization of DBSCAN clusters with parameters $k=6$ and $\epsilon=0.5$ (left) and with parameters $k=48$ and $\epsilon=3$ (right).

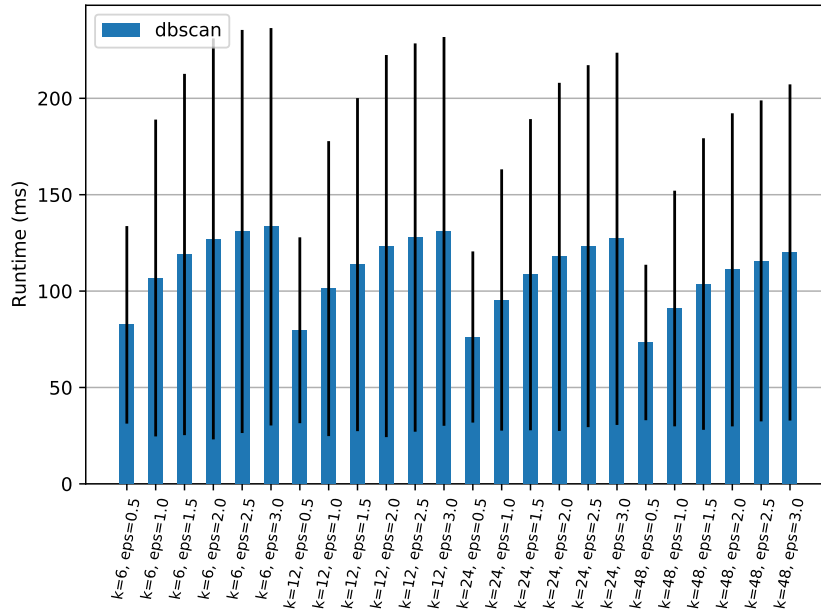


Figure 6.8: **DBSCAN runtime comparison.** Runtime of DBSCAN in ms for different k and ϵ values for a pair of point clouds averaged over the 142 frames (5 runs each).

The final results of the classification are now analyzed jointly with the clustering and the association of clusters. Table 6.5 shows the best results with regard to the number of points classified as static for the *Euclidean* distance histogram measure and both the *chamfer* and $L_{2-\text{centroid}}$ cluster distance measures and report relevant classification and association metrics. The results for the *Bhattacharyya* and *Intersection* distances are shown in Appendix 2 as they are the same.

The best distance and ratio threshold D_{thresh} and R_{thresh} are both 0.3m and 0.3 respectively. The *chamfer* distance metric between the clusters is the one classifying the most points as static, compared to the $L_{2-\text{centroid}}$ metric. the Ratio_A tends to be larger when the *Euclidean* distance is minimized, which is expected. Regarding the clustering, only small ϵ values give satisfactory results while all ranges for k are observed. The *EPE3D* and *Acc3DS* are reported for the ground points and the newly classified static points combined. Compared to the results from the ground

points after the ego-motion refinement in Table 6.3, a slight increase in the *EPE3D* is observed while the *Acc3DS* decreases a bit for the *chamfer* distance. The values remain within acceptable limits, indicating our confidence that the clusters identified as static are indeed static. Concerning the $L_{2-\text{centroid}}$ cluster distance, a slight improvement in the *EPE3D* score is even noticed.

The parameters $k=48$, $\epsilon=1.0$ with the *Euclidean* histogram distance measure, the *chamfer* cluster distance measure and the thresholds $D_{\text{thresh}}=0.3$, $R_{\text{thresh}}=0.3$ are the one retained. It gives one of the best runtimes for DBSCAN along with a good silhouette score, $\text{Ratio}_{\text{noise}}$, $\text{Ratio}_{\text{cluster}}$ and number of clusters $|C|$ while reaching decent results for the classification and the best results for the sum of histogram distances and the Ratio_A .

<i>Euclidean</i>									
	k	ϵ	D_{thresh}	R_{thresh}	$\sum D_{L_2} \downarrow$	$\text{Ratio}_A \uparrow$	<i>static</i> $\uparrow$	<i>EPE3D</i> $\downarrow$ (m)	<i>Acc3DS</i> $\uparrow$
<i>chamfer</i>	6	0.5	0.3	0.3	18.82	0.222	13742.27	0.0195	0.9668
	6	1.0	0.3	0.3	18.51	0.317	13472.57	0.0207	0.9658
	24	1.0	0.3	0.3	16.44	0.496	13428.19	0.0206	0.9658
	48	1.0	0.3	0.3	14.89	0.627	13421.31	0.0194	0.9676
$L_{2-\text{centroid}}$	6	1.0	0.3	0.3	18.51	0.317	7295.06	0.0196	0.971
	6	0.5	0.3	0.3	18.82	0.222	7272.76	0.0135	0.977
	12	1.0	0.3	0.3	17.61	0.397	7227.73	0.0195	0.9712
	12	0.5	0.3	0.3	17.08	0.288	7044.22	0.0135	0.977

Table 6.5: **Clusters classification results.** Shows the best results with regard to the number of points classified as *static* for the *Euclidean* histogram distance measure and both the *chamfer* and $L_{2-\text{centroid}}$ cluster distance measures and report metrics averaged over the 142 frames.

Finally, we measure the runtime of the association process with respect to the number of threads and the two cluster distance measures *chamfer* and $L_{2-\text{centroid}}$, as shown in Fig. 6.9. As for the ground segmentation, no improvement is observed from 8 threads and more. Computing the *chamfer* distance is more expensive as expected, due to the nearest neighbor calculation for each point, leading to a runtime of $\sim 50\text{ms}$, shown in Fig. 6.9b. The $L_{2-\text{centroid}}$ distance, on the other hand, allows to reach a runtime of $\sim 17\text{ms}$, as shown in Fig. 6.9a, although performing less well in terms of number of points classified as static.

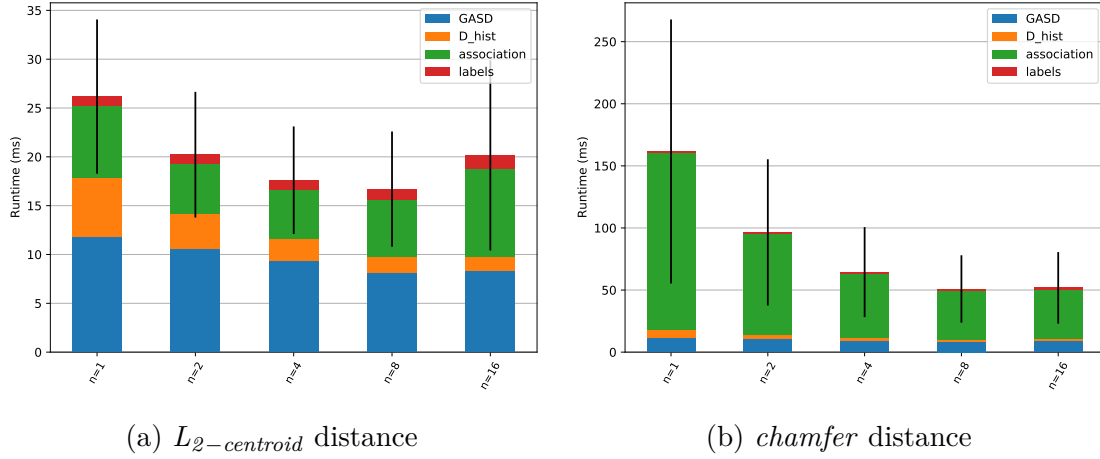


Figure 6.9: **Association runtime comparison.** Runtime in ms for different numbers of threads and averaged over the 142 frames (5 runs each), showing the different steps of association: computing the GASD descriptor, the histogram distances between all pairs, the associations and corresponding cluster distance, and the final labels.

4 Scene Flow prediction

In this final section, we discuss the training and quantitative results of the network for scene flow prediction. The network is first pre-trained on SF-KITTI [19] and then fine-tuned and evaluated on KITTI scene flow 2015 [42]. The runtime and scalability of the proposed network with PSNet [36] are discussed and compared with PointNet++ [45] as the feature extractor.

4.1 Dataset

Since KITTI Scene flow 2015 [42] only contains 142 pairs of point clouds with ground truth flow vectors, which is too small for proper training, we use the SF-KITTI dataset proposed by FH-Net [19] consisting of 5700 pairs of point clouds. They use LiDAR-scanned point clouds from the raw data of KITTI [24], cropping them within the camera view, and leverage ground truth 3D bounding boxes to compute a rigid-transformation, serving as the pseudo scene flow ground truth. For points not belonging to any 3D box, they use the ego-motion transformation between the two time instants as ground truth. They then augment the dataset by collecting pairs of dynamic objects and pasting them in safe areas in the source point cloud S and the target point cloud T , respectively. The objects are placed in random poses and the ground truth flow is computed from these two poses.

4.2 Loss function

The loss function proposed by FH-Net [19] defined as follows is used:

$$\mathcal{L}_h = \sum_{k=0}^2 \alpha_k \frac{1}{N_k} \sum_{i=1}^{N_k} \beta_i \|f_i^k - \hat{f}_i^k\|_2^2 \quad (6.6)$$

where β_i is a weight set to 5 for foreground points and 0.5 for background points, hence forcing the flows of foreground points to be better predicted, N_k is the number of points at level k and $f_i^k, \hat{f}_i^k$ are the predicted and ground truth flow, respectively. As FH-Net [19] outputs flows at 3 different resolutions, a weight for each level is defined by α_k . We set $\alpha_0=0.2$, $\alpha_1=0.2$ and $\alpha_2=0.6$ by default, as proposed by FH-Net [19].

4.3 Training and Evaluation

pre-training on SF-KITTI [19]

We first start by training 20 epochs on SF-KITTI [19] with the Adam optimizer, a batch size of 8, and a learning rate of 0.001. Regularization techniques such as Dropout [31] and weight decay (i.e. L_2 penalty on the weights) are also investigated to prevent overfitting and help generalize on unseen data. Given a probability p , a dropout [31] layer zero out the output of the neurons with a probability p . This prevents neurons from co-adapting their weights to predict the training set perfectly, hence reducing overfitting. The dataset is split in 80%/20% for the training and validation set, respectively. We do not use any test set as the model is fine-tuned and evaluated on KITTI Scene flow 2015 [42] only.

Fig. 6.10 shows the first training for different weight decays λ and dropout probabilities p . The stronger the L_2 regularization on the weights, the higher the training loss is, which is expected as the model is less prone to overfitting. The validation losses for $\lambda=0.01$ seem to be more stable at the beginning but quickly diverge after 10 epochs. For $\lambda=0.001$ and $\lambda=0$, the validation losses seem to oscillate even though for $p=0.2$, they are a bit lower. In general, the model struggles with training and convergence, maybe because the learning rate is too high. Table 6.6 shows the evaluation of the different models on the 142 frames of the KITTI scene flow 2015 [42] dataset, without fine-tuning, giving as input to the network the points classified as moving by the previous steps. Overall, the performances are highly unsatisfactory with the best solution found for $\lambda=0.001$ and $p=0.1$.

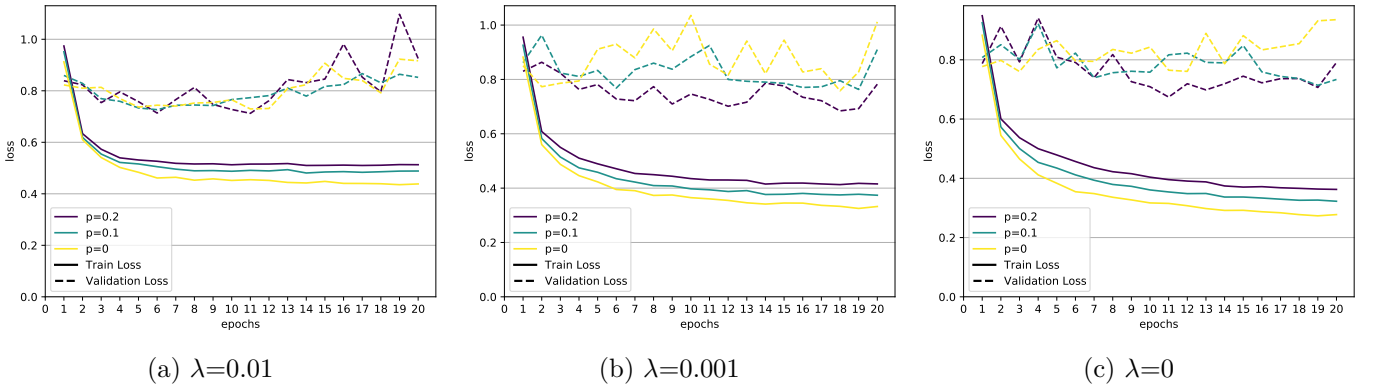


Figure 6.10: **Training on SF-KITTI (20 epochs) [19]**. Training on SF-KITTI for 20 epochs using different dropout probabilities p and weight decay values λ , showing the training and validation loss.

λ	p	$EPE3D \downarrow$ (m)	$Acc3DS \uparrow$	$Acc3DR \uparrow$	$Outliers \downarrow$
0.01	0.2	0.6162	0.0715	0.1642	0.5791
	0.1	0.5920	0.0582	0.1377	0.6085
	0	0.6701	0.0740	0.1545	0.6273
0.001	0.2	0.7355	0.0288	0.0706	0.7343
	0.1	0.6101	0.0742	0.1806	0.5802
	0	0.6329	0.0738	0.1524	0.6064
0	0.2	0.5777	0.0417	0.1314	0.6057
	0.1	0.7696	0.0138	0.0303	0.8251
	0	0.6669	0.0272	0.0818	0.6829

Table 6.6: **Evaluation on 20 epochs.** Evaluation metrics on moving points of the 142 frames of KITTI Scene flow 2015 [42] w/o fine-tuning. Models are taken from the epoch with the best validation loss.

We now train the 3 models for $\lambda=0.01$, the 2 best models for $\lambda=0.001$, and the best model for $\lambda=0$ for 40 epochs, starting at a learning rate of 0.0005 and dividing it by 2 at epochs=[5, 15, 30], hoping for a more stable learning with a better convergence. Fig. 6.11 shows the learning curves of the selected models. The training losses are lower compared to Fig. 6.10 except for $\lambda=0$, and the validation losses exhibit fewer oscillations but still plateau at relatively high values. Table 6.7 presents the evaluation results on KITTI scene flow 2015 [42]. All models perform worse except for the model trained with $\lambda=0.001$ and $p=0$, using only regularization on the weights.

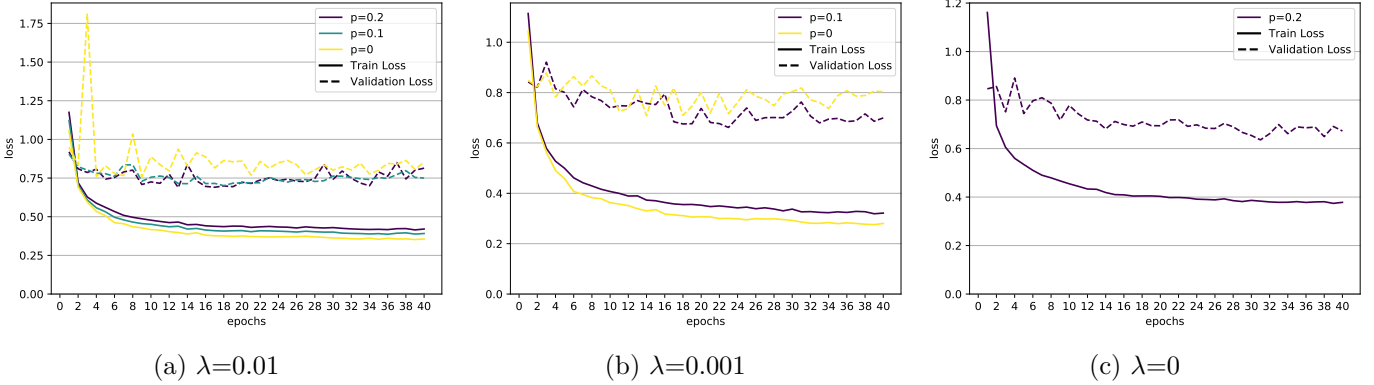


Figure 6.11: **Training on SF-KITTI (40 epochs) [19].** Training on SF-KITTI for 40 epochs using different dropout probabilities p and weight decay values λ , showing the training and validation loss.

It is worth noting that up until now, all models are trained on SF-KITTI [19] and then evaluated on KITTI scene flow 2015 [42], and that there is a gap between the domains of the two datasets. Indeed, SF-KITTI directly works with LiDAR-scanned point clouds while KITTI scene flow 2015 comprises point clouds obtained from images. Still, the best model trained with $\lambda=0.001$ and $p=0$ achieves insufficient results.

λ	p	$EPE3D \downarrow$ (m)	$Acc3DS \uparrow$	$Acc3DR \uparrow$	$Outliers \downarrow$
0.01	0.2	0.6536	0.0327	0.0934	0.6497
	0.1	0.6827	0.0360	0.0967	0.6675
	0	0.7356	0.0493	0.1018	0.7294
0.001	0.1	0.6576	0.0432	0.0983	0.6787
	0	0.5855	0.1312	0.2290	0.5791
0	0.2	0.6071	0.0191	0.0624	0.6594

Table 6.7: **Evaluation on 40 epochs.** Evaluation metrics on moving points of the 142 frames of KITTI Scene flow 2015 [42] w/o fine-tuning. Models are taken from the epoch with the best validation loss.

In this network, the downsampling is learned throughout the training and it is crucial that the sampling is consistent across the two point clouds. Otherwise, correct correspondences cannot be established, resulting in wrong coarse flow estimations which in turn result in wrong dense and raw resolution flow predictions. In this regard, we first pre-train the best model ($\lambda=0.001$ and $p=0$) with $\alpha_0=1$, $\alpha_1=0$, $\alpha_2=0$ for 25 epochs, only learning weights for the two set abstractions and the Trans-Flow layer and forcing the network to better predict the coarse flow. Next, we train for 25 epochs with $\alpha_0=0.3$, $\alpha_1=0.7$, $\alpha_2=0$, forcing the network to better predict the denser flows. The learning rate starts at 0.0005 and is divided by 2 at epochs=[5, 10, 15, 20]. Finally, the network is trained normally with $\alpha_0=0.2$, $\alpha_1=0.2$, $\alpha_2=0.6$ for 50 epochs, dividing the learning by 2 at epochs=[5, 15, 30, 40].

Fig. 6.12 and Table 6.8 show the results of the training and the evaluation of the model, respectively. For the first pre-training on the Trans-Flow layer only, the validation and training losses again saturate, not showing any improvement after epoch 13. We observe the same phenomenon for the pre-training on the Trans-Up layer and the final pre-training on all layers. Their training loss starts at a lower value simply because the network was already trained on other layers. The evaluation results of the model are still highly insufficient.

λ	p	$EPE3D \downarrow$ (m)	$Acc3DS \uparrow$	$Acc3DR \uparrow$	$Outliers \downarrow$
0.001	0	0.6865	0.0578	0.1115	0.6865

Table 6.8: **Final evaluation of pre-training on SF-KITTI [19].** Evaluation metrics on moving points of the 142 frames of KITTI Scene flow 2015 [42] w/o fine-tuning on the best model resulting from the full pre-training.

There are multiple reasons why the network can fail to learn effectively. First, it can come from the model itself. from all the training realized until now, we see that the training and validation losses remain quite high and quickly stagnate. This can be a sign of underfitting, meaning the model is not complex enough to fit the training data. Secondly, it can come from the data. The fact that there is a large gap between the training and the validation loss can indicate that the training set is not representative enough to learn the problem at hand. We also observed in some cases that the validation loss oscillates quite a lot, which can indicate that the validation set is not representative enough and differs from the training set or that there is

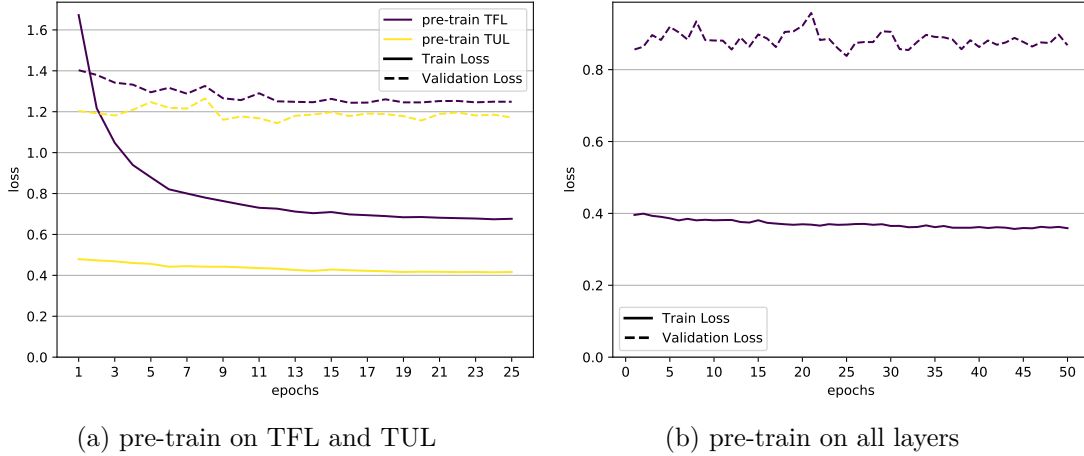


Figure 6.12: **Final pre-training on SF-KITTI [19]**. shows the learning curves for the pre-training on the Trans-Flow and Trans-Up layers (left) followed by the final pre-training on all layers (right).

not enough data in the validation set. This could be improved by increasing the size of the validation set to reduce the variance, but this would in turn reduce the training set size and potentially increase bias. We would need to increase the size of the whole dataset but unfortunately, SF-KITTI [19] only provides a limited amount of data. Finally, it can come from the training itself and its hyper-parameters. In this regard, different optimizers, more learning rates and regularization values for λ and p could be tried.

Fine-tuning on KITTI scene flow 2015 [42]

We now fine-tune the final model obtained from the pre-training on SF-KITTI [19] on the KITTI scene flow 2015 [42] dataset. It is split in 75%/25% for the training and test set, respectively. 20% of the training set is kept as the validation set. This leaves out of the total 142 frames 85 frames for the training set, 21 frames for the validation set, and 36 frames for the test set. The network is fine-tuned for 50 epochs with a batch size of 4, starting at a learning rate of 0.00025 and dividing it by 2 at epochs=[5, 15, 30, 40]. The same values for λ and p are tried.

Fig. 6.13 and Table 6.9 show the training and the evaluation of the best fine-tuned model, respectively. We observe a highly unstable validation loss coming from the fact that the validation set is this time really small (21 frames). The training loss remains high, showing signs of underfitting again. It is not possible to tell whether or not the model does or would transfer well from the domain of the SF-KITTI dataset to the domain of the KITTI scene flow 2015 dataset as it is not learning properly. The performance metrics are still in the same order of magnitude as before, showing no improvement.

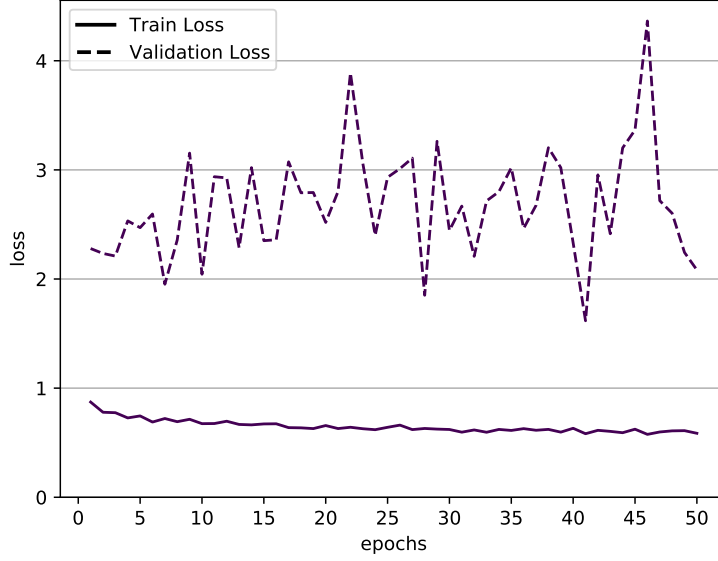


Figure 6.13: **Fine-tuning on KITTI scene flow 2015 [42]**. Fine-tune training for $\lambda=0$ and $p=0$, giving the best evaluation results.

λ	p	$EPE3D \downarrow$ (m)	$Acc3DS \uparrow$	$Acc3DR \uparrow$	$Outliers \downarrow$
0	0	0.7556	0.0497	0.1024	0.6999

Table 6.9: **Final evaluation on KITTI scene flow 2015 [42]**. Evaluation metrics on moving points of the 142 frames of KITTI Scene flow 2015 w/ fine-tuning on KITTI scene flow 2015.

4.4 Runtime and scalability

At last, we evaluate the inference time of the model with PSNet [36] as the set abstraction and compare it with PointNet++ [45] plugged in as the set abstraction instead. The implementation of PointNet++ is taken from their GitHub repository¹ and the GPU time is measured on the Tesla V100. Different values for the number of input points are tried to evaluate the scalability of the models. The two set abstractions sample 2048 and then 1024 points, respectively.

The results are shown in Table 6.10. PSNet [36] is much faster with runtimes below 100ms except for 65536 points, making it suitable for real-time applications while PointNet++ [45] is running above 1s, demonstrating the expensive computational resources needed for the farthest point sampling. In terms of scalability, we observe the runtime with PSNet doubling each time after 16384 points, while it seems more stable with PointNet++, at least when sampling 2048 and 1024 points.

¹https://github.com/yanx27/Pointnet_Pointnet2_pytorch

Set abstraction	#points	runtime ↓ (ms)
PSNet [36]	4096	22.47
	8192	24.77
	16384	32.39
	32768	64.94
	65536	114.79
PointNet++ [45]	4096	1084.72
	8192	1105.30
	16384	1164.29
	32768	1208.77
	65536	1312.83

Table 6.10: **Inference runtime of PSNet [36] vs. PointNet++ [45]**. Runtime in ms for different number of input points averaged over 100 runs.

To conclude, the effectiveness of the network using PSNet [36] was not demonstrated due to the failure of the network in learning although showing good results in terms of runtime where the network could in theory be used for a real-time application if trained properly.

Chapter 7

Discussion and Further improvements

Several solutions have been proposed in this thesis to try and improve the overall scene flow prediction of current methods. In this final chapter, we discuss the different proposed solutions along with further improvements and directions that could potentially be explored.

The ground segmentation, although achieving sufficient results overall, is quite naïve and can produce bad results in specific scenarios. Its capacity is limited in the case where the ground is composed of both flat and sloped terrain or presents sharp changes, like a sidewalk curb, and this limitation comes from the angle threshold that is set for the normal vectors. Approaches based on 2.5D grid [27] can provide solutions for dealing with uneven ground while still achieving real-time. The second limitation is when the points are distributed in concentric circles too far from each other, which is generally the case with LiDAR point clouds. Techniques involving the division of space in a polar grid while extracting lines instead of planes [27] could be investigated to solve this issue.

The convergence time of point-to-plane ICP could be improved with a better initial ego-motion estimate. The GPS/IMU measurements are not synchronized with the cameras but timestamps of the measurements are available. One could interpolate the values between two GPS/IMU measurements, providing in turn a better estimate of the car’s pose and thus of the ego-motion. Point-to-plane ICP could then converge in fewer iterations.

The moving object segmentation using DSCBAN [22] for the clustering of objects further eliminates a significant number of static points. However, DBSCAN is a bottleneck in terms of runtime for a real-time application, even when running on a GPU. In combination with a better ground segmentation, a simpler and faster clustering algorithm could be used, as there would be no ground connecting the clusters. There exist multiple clustering algorithms in the literature running in $\Theta(n)$ and suited for large-scale and/or spatial data, such as WaveCluster, STING, and BIRCH [62] that are worth exploring. Better metrics should also be defined

to compare the clustering between the two point clouds, not relying only on the number of clusters but also on the intrinsic information of the clustered data. Finally, there exists a plethora of distance and similarity measures between histograms or probability density functions [17] that should be studied in combination with other distance metrics between clusters to improve the association of clusters and eliminate more static points. Exploring other distance metrics between clusters could improve the runtime of the association, which is also a limitation for real-time applications when using the *chamfer* distance.

The proposed network for scene flow prediction using PSNet [36] presents an interesting approach for sampling and grouping the points that looks promising in theory, avoiding expensive operations such as FPS and KNN. In practice, it was shown that the network struggles with training, and several reasons why this is the case were mentioned. Concerning the datasets, there is not much that can be done as they are given as is. However, as the training shows signs of underfitting, the parameters of the network can be tuned to increase the complexity of the model. The number of features or layers' width of the MLPs can be increased as well as the depth of the network, i.e. the number of hidden layers inside the MLPs. Other K values or radius values R in the Trans-Flow and Trans-Up layers can be investigated, as well as in the set abstractions.

It should be noted that the number of points sampled at the first and second set abstraction was set to 2048 and 1024, which might be too small when working with input point clouds of size 16384 for a correct prediction. However, increasing the number of sampled points increases the number of parameters of PSNet [36] as the points are mapped to s -dimensional vectors in the SFTF. It could be interesting to try and train FH-Net [19] with PointNet++ [45] instead of PSNet to determine if the issue might be coming from the feature learning and the learning-based down-sampling. Originally, PSNet was proposed as an alternative to PointNet++ for the tasks of object classification, part segmentation, and scene segmentation, where only a single point cloud is taken as input. In our case, we have two point clouds as input and it is crucial that the sampling is consistent across the two points clouds. In this regard, maybe a learning-based sampling adapted for such a case could be studied. Here, the set abstractions are shared between the two point clouds, and separate learning for the two point clouds can be investigated as well. Another possibility could be to pre-train the layers of PSNet [36] on a completely different task to learn the sampling, such as point cloud reconstruction, and then fine-tune on the task of scene flow prediction.

Finally, more training parameters should be tested in combination with the previously discussed further improvements. Other training losses can be investigated or an additional term to the current loss function of FH-Net [19] taking into account the fact that the downsampling is learning-based could be added.

Conclusion

The scene flow is a low-level scene understanding component and is crucial for autonomous vehicles. The objective of this thesis was to improve its prediction as well as some pre-processing steps. Indeed, current methods take point clouds as input and predict the flow directly, without taking additional factors into account. These factors include the ego-motion of the car, the real-time requirement of such an application, and also the fact most of the points in a scene are static, and their flow need not be predicted.

Non-learning-based pre-processing steps for the ground segmentation, the refinement and cancelation of the ego-motion, and the moving object classification were proposed. The effectiveness and limitations of the approaches were discussed, with the moving object classification being the biggest bottleneck, especially in terms of runtime, and some further improvements were proposed.

A learning-based downsampling and feature learning approach was proposed to be plugged into an existing fast hierarchical network to predict the final scene flow of moving points. Although promising on paper and in terms of inference time, as shown by experiments, the network's inability to train results in highly unsatisfactory results. Some elements of response to why it is the case were provided and further improvements concerning the network and the training were suggested.

Bibliography

- [1] Clustering. <https://scikit-learn.org/stable/modules/clustering.html#>. Accessed: 2024-04-10.
- [2] Confusion matrix. https://en.wikipedia.org/wiki/Confusion_matrix. Accessed: 2024-05-14.
- [3] CÉCI-HPC. <https://www.cec-hpc.be/>. Accessed: 2024-05-14.
- [4] Gated recurrent unit. https://en.wikipedia.org/wiki/Gated_recurrent_unit. Accessed: 2024-05-25.
- [5] Gauss-newton algorithm. https://en.wikipedia.org/wiki/Gauss%E2%80%9993Newton_algorithm. Accessed: 2024-05-29.
- [6] Kabsch algorithm. https://en.wikipedia.org/wiki/Kabsch_algorithm. Accessed: 2024-05-29.
- [7] The kitti vision benchmark suite. https://www.cvlibs.net/datasets/kitti/eval_scene_flow.php. Accessed: 2024-04-04.
- [8] Principal component analysis. https://en.wikipedia.org/wiki/Principal_component_analysis. Accessed: 2024-05-09.
- [9] Rapids, gpu accelerated data science. <https://rapids.ai/>. Accessed: 2024-05-21.
- [10] Singular value decomposition. https://en.wikipedia.org/wiki/Singular_value_decomposition. Accessed: 2024-05-13.
- [11] Hassan Abu Alhaija, Siva Karthik Mustikovela, Lars Mescheder, Andreas Geiger, and Carsten Rother. Augmented reality meets computer vision: Efficient data generation for urban driving scenes. *International Journal of Computer Vision*, 126:961–972, 2018.
- [12] M Aharchi and M Ait Kbir. A review on 3d reconstruction techniques from 2d images. In *Innovations in Smart Cities Applications Edition 3: The Proceedings of the 4th International Conference on Smart City Applications 4*, pages 510–522. Springer, 2020.

- [13] Aitor Aldoma, Markus Vincze, Nico Blodow, David Gossow, Suat Gedikli, Radu Bogdan Rusu, and Gary Bradski. Cad-model recognition and 6dof pose estimation using 3d cues. In *2011 IEEE international conference on computer vision workshops (ICCV workshops)*, pages 585–592. IEEE, 2011.
- [14] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.
- [15] Paul J Besl and Neil D McKay. Method for registration of 3-d shapes. In *Sensor fusion IV: control paradigms and data structures*, volume 1611, pages 586–606. Spie, 1992.
- [16] Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander. Density-based clustering based on hierarchical density estimates. In *Pacific-Asia conference on knowledge discovery and data mining*, pages 160–172. Springer, 2013.
- [17] Sung-Hyuk Cha and Sargur N Srihari. On measuring the distance between histograms. *Pattern Recognition*, 35(6):1355–1370, 2002.
- [18] Yang Chen and Gérard Medioni. Object modelling by registration of multiple range images. *Image and vision computing*, 10(3):145–155, 1992.
- [19] Lihe Ding, Shaocong Dong, Tingfa Xu, Xinli Xu, Jie Wang, and Jianan Li. Fh-net: A fast hierarchical network for scene flow estimation on real-world point clouds. In *European Conference on Computer Vision*, pages 213–229. Springer, 2022.
- [20] Joao Paulo Silva do Monte Lima and Veronica Teichrieb. An efficient global point cloud descriptor for object recognition and pose estimation. In *2016 29th SIBGRAPI conference on graphics, patterns and images (SIBGRAPI)*, pages 56–63. IEEE, 2016.
- [21] Guanting Dong, Yueyi Zhang, Hanlin Li, Xiaoyan Sun, and Zhiwei Xiong. Exploiting rigidity constraints for lidar scene flow estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12776–12785, 2022.
- [22] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd*, volume 96, pages 226–231, 1996.
- [23] Junhao Gan and Yufei Tao. Dbscan revisited: Mis-claim, un-fixability, and approximation. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 519–530, 2015.
- [24] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.

- [25] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3354–3361. IEEE, 2012.
- [26] Zan Gojcic, Or Litany, Andreas Wieser, Leonidas J Guibas, and Tolga Birdal. Weakly supervised learning of rigid 3d scene flow. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 5692–5703, 2021.
- [27] Tiago Gomes, Diogo Matias, André Campos, Luís Cunha, and Ricardo Roriz. A survey on ground segmentation methods for automotive lidar sensors. *Sensors*, 23(2):601, 2023.
- [28] Xiuye Gu, Yijie Wang, Chongruo Wu, Yong Jae Lee, and Panqu Wang. Hplflownet: Hierarchical permutohedral lattice flownet for scene flow estimation on large-scale point clouds. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 3254–3263, 2019.
- [29] Xian-Feng Han, Zhi-Ao Feng, Shi-Jie Sun, and Guo-Qiang Xiao. 3d point cloud descriptors: state-of-the-art. *Artificial Intelligence Review*, 56(10):12033–12083, 2023.
- [30] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, second edition, 2004.
- [31] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.
- [32] Xiaoshui Huang, Guofeng Mei, Jian Zhang, and Rana Abbas. A comprehensive survey on point cloud registration. *arXiv preprint arXiv:2103.02690*, 2021.
- [33] Martin Kiefel, Varun Jampani, and Peter V Gehler. Permutohedral lattice cnns. *arXiv preprint arXiv:1412.6618*, 2014.
- [34] Yair Kittenplon, Yonina C Eldar, and Dan Raviv. Flowstep3d: Model unrolling for self-supervised scene flow estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4114–4123, 2021.
- [35] Bing Li, Cheng Zheng, Silvio Giancola, and Bernard Ghanem. Sctn: Sparse convolution-transformer network for scene flow estimation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 1254–1262, 2022.
- [36] Luyang Li, Ligang He, Jinjin Gao, and Xie Han. Psnet: Fast data structuring for hierarchical deep learning on point cloud. *IEEE Transactions on Circuits and Systems for Video Technology*, 32(10):6835–6849, 2022.
- [37] Zhiqi Li, Nan Xiang, Honghua Chen, Jianjun Zhang, and Xiaosong Yang. Deep learning for scene flow estimation on point clouds: a survey and prospective

- trends. In *Computer Graphics Forum*, volume 42, page e14795. Wiley Online Library, 2023.
- [38] Haisong Liu, Tao Lu, Yihui Xu, Jia Liu, Wenjie Li, and Lijun Chen. Camliflow: bidirectional camera-lidar fusion for joint optical flow and scene flow estimation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 5791–5801, 2022.
 - [39] Xingyu Liu, Charles R Qi, and Leonidas J Guibas. Flownet3d: Learning scene flow in 3d point clouds. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 529–537, 2019.
 - [40] Kok-Lim Low. Linear least-squares optimization for point-to-plane icp surface registration. 01 2004.
 - [41] Nikolaus Mayer, Eddy Ilg, Philip Hausser, Philipp Fischer, Daniel Cremers, Alexey Dosovitskiy, and Thomas Brox. A large dataset to train convolutional networks for disparity, optical flow, and scene flow estimation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4040–4048, 2016.
 - [42] Moritz Menze and Andreas Geiger. Object scene flow for autonomous vehicles. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3061–3070, 2015.
 - [43] Gilles Puy, Alexandre Boulch, and Renaud Marlet. Flot: Scene flow on point clouds guided by optimal transport. In *European conference on computer vision*, pages 527–544. Springer, 2020.
 - [44] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, 2017.
 - [45] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *Advances in neural information processing systems*, 30, 2017.
 - [46] Eréndira Rendón, Itzel Abundez, Alejandra Arizmendi, and Elvia M Quiroz. Internal versus external cluster validation indexes. *International Journal of computers and communications*, 5(1):27–34, 2011.
 - [47] Radu Bogdan Rusu. *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments*. PhD thesis, Computer Science department, Technische Universitaet Muenchen, Germany, October 2009.
 - [48] Radu Bogdan Rusu, Nico Blodow, and Michael Beetz. Fast point feature histograms (fpfh) for 3d registration. In *2009 IEEE international conference on robotics and automation*, pages 3212–3217. IEEE, 2009.

- [49] Radu Bogdan Rusu, Gary Bradski, Romain Thibaux, and John Hsu. Fast 3d recognition and pose using the viewpoint feature histogram. In *2010 IEEE/RSJ international conference on intelligent robots and systems*, pages 2155–2162. IEEE, 2010.
- [50] Radu Bogdan Rusu and Steve Cousins. 3d is here: Point cloud library (pcl). In *2011 IEEE international conference on robotics and automation*, pages 1–4. IEEE, 2011.
- [51] Radu Bogdan Rusu, Zoltan Csaba Marton, Nico Blodow, and Michael Beetz. Persistent point feature histograms for 3d point clouds. In *Proc 10th Int Conf Intel Autonomous Syst (IAS-10), Baden-Baden, Germany*, pages 119–128, 2008.
- [52] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. Dbscan revisited, revisited: why and how you should (still) use dbscan. *ACM Transactions on Database Systems (TODS)*, 42(3):1–21, 2017.
- [53] Bruno Siciliano and Oussama Khatib, editors. *Springer Handbook of Robotics*. Springer, Berlin, Heidelberg, 2008.
- [54] Paul A Ullrich and Colin M Zarzycki. Tempestextremes: A framework for scale-insensitive pointwise feature tracking on unstructured grids. *Geoscientific Model Development*, 10(3):1069–1090, 2017.
- [55] Sundar Vedula, Simon Baker, Peter Rander, Robert Collins, and Takeo Kanade. Three-dimensional scene flow. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 722–729. IEEE, 1999.
- [56] Guangming Wang, Yunzhe Hu, Zhe Liu, Yiyang Zhou, Masayoshi Tomizuka, Wei Zhan, and Hesheng Wang. What matters for 3d scene flow network. In *European Conference on Computer Vision*, pages 38–55. Springer, 2022.
- [57] Guangming Wang, Xinrui Wu, Zhe Liu, and Hesheng Wang. Hierarchical attention learning of scene flow in 3d point clouds. *IEEE Transactions on Image Processing*, 30:5168–5181, 2021.
- [58] Haiyan Wang, Jiahao Pang, Muhammad A Lodhi, Yingli Tian, and Dong Tian. Festa: Flow estimation via spatial-temporal attention for scene point clouds. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14173–14182, 2021.
- [59] Yi Wei, Ziyi Wang, Yongming Rao, Jiwen Lu, and Jie Zhou. Pv-raft: Point-voxel correlation fields for scene flow estimation of point clouds. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 6954–6963, 2021.
- [60] Wenxuan Wu, Zhongang Qi, and Li Fuxin. Pointconv: Deep convolutional networks on 3d point clouds. In *Proceedings of the IEEE/CVF Conference on computer vision and pattern recognition*, pages 9621–9630, 2019.

- [61] Wenxuan Wu, Zhi Yuan Wang, Zhuwen Li, Wei Liu, and Li Fuxin. Pointpwc-net: Cost volume on point clouds for (self-) supervised scene flow estimation. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part V 16*, pages 88–107. Springer, 2020.
- [62] Dongkuan Xu and Yingjie Tian. A comprehensive survey of clustering algorithms. *Annals of data science*, 2:165–193, 2015.

Appendices

1 Background and Related Work

1.1 Training and loss functions

Training approaches and loss functions used by current methods come in different flavors. FlowNet3D [39], HPLFlowNet [28], PV-RAFT [59], HALFlow [57], WhatMatters [56], FLOT [43], SCTN [35] and FESTA [58] are all trained in a **fully-supervised** way. FlowStep3D [34] and PointPWC-Net [61] also propose to train in a **self-supervised** fashion while LiDARSceneFlow [21] and Rigid3DSceneFlow [26] make use of partially-labeled data and train in a **weakly-supervised** manner.

Supervised losses

Hierarchical methods such as PointPWC-Net [61], HALFlow [57] and WhatMatters [56] use a hierarchical L_2 loss as follows:

$$\mathcal{L} = \sum_{l=1}^L \alpha_l \frac{1}{n_1^l} \sum_{i=1}^{N_1^l} \|F_i^{GT(l)} - F_i^{(l)}\|_2 \quad (1)$$

where α_l is the weight for level l and n_1^l is the number of points at level l . SCTN [35], FLOT [43], FlowNet3D [39] and HPLFlowNet [28] use the same loss, either L_2 or L_1 , but only at the last level L . FlowNet3D [39] adds an additional *cycle-consistency* regularization term $\lambda \|F'_i + F_i\|_1$ where F'_i is the backward flow predicted from the warped source point cloud with the predicted forward flow F . SCTN [35], FLOT [43] multiply the term for each point with a binary value m_i indicating if the point p_i is occluded or not in the second frame. SCTN [35] introduces a *feature-aware spatial consistency* regularization term that forces points with similar features in the neighborhood to follow the same flows. Iterative methods such as FESTA [58], PV-RAFT [59] and FlowStep3D [34] compute a L_2 or L_1 loss at each iteration k , similar to Eq. 1. PointPWC-Net [61] and FlowStep3D [34] both define a *Laplacian* regularization to ensure that the Laplacian of the warped point cloud is preserved, where the Laplacian at point p_i is approximated by its distances from its neighbors.

Self-supervised losses

PointPWC-Net [61] and FlowStep3D [34] both use the Chamfer loss, which is a common choice in a self-supervised or weakly-supervised setting, and which is defined as:

$$\mathcal{L}^{ch} = \sum_{p_i \in S_w} \min_{q_j \in T} \|p_i - q_j\|_2^2 + \sum_{q_j \in T} \min_{p_i \in S_w} \|p_i - q_j\|_2^2 \quad (2)$$

where S_w is the warped source point cloud with current flow F_k and T the target point cloud. They also use the Laplacian regularization along with the chamfer loss as in the supervised training.

Weakly-supervised losses

Finally, methods such as Rigid3DSceneFlow [26] and LiDARSceneFlow [21], that predict rigid transformations, propose a first loss term for the ego-motion, making

use of labels:

$$\mathcal{L}^{ego} = \frac{1}{n_{BG}} \sum_{j=1}^{N_{BG}} \left\| (R_{ego} x_j^b + t_{ego}) - (R_{ego}^{GT} x_j^b + t_{ego}^{GT}) \right\|_1 \quad (3)$$

where $(R_{ego}^{GT}, t_{ego}^{GT})$ is the ground truth transformation and n_{BG} the number of background points. They also add the chamfer loss $\mathcal{L}^{ch}$ as an additional term and a rigidity loss term, both not using any labels.

1.2 Model complexities and inference time

Table 1 compares the runtime and complexities (#parameters) of all architectures mentioned in Chapter 1. Most models contain more than 1M parameters except for FLOT [43], PV-RAFT [59] and FlowStep3D [34]. Their runtime is generally above 100ms, the frequency at which LiDAR point clouds are scanned except for FH-Net [19] and SCTN [35], although SCTN only works with 2048. Some methods [56, 57] do not even mention the complexity of their model nor the inference time.

Method	#Parameters (M)	runtime (ms)	#points	GPU
FlowNet3D [39]	4.8	130.8 279.2 770.2	8192 16384 32768	Titan V
HPLFlowNet [28]	19	98.4 115.5 142.8	8192 16384 32768	Titan V
PointPWCNet [61]	7.7	117.4	8192	GTX 1080 Ti
Rigid3DSceneFlow [26]	8.08	147.0	8192	Titan V
LiDARSceneFlow [21]	1.37	538	/	GTX 1050 Ti
FlowStep3D [34]	0.69	2658	/	GTX 1050 Ti
PV-RAFT [59]	0.19	5296	/	GTX 1050 Ti
FLOT [43]	0.11	389.3	8192	GTX 2080 Ti
SCTN [35]	/	242.7	8192	GTX 2080 Ti
FESTA [58]	5.4	67.8	2048	GTX 1080 Ti
HALFlow [57]	/	/	/	/
WhatMatters [56]	/	/	/	/
FH-Net [19]	1.5	17.45	/	RTX 3090

Table 1: **Models Complexity and runtime comparison.** Compares the runtime in ms of state-of-art methods along with their number of parameters. The number of points on which the runtime is computed is denoted when specified. Unknown values are also mentioned.

2 Moving object segmentation

Table 2 shows the results regarding the cluster-based classification for the *Bhattacharyya* distance and *Intersection* similarity measures between histograms. The results are the same as those obtained for the *Euclidean distance* shown in Table 6.5.

The *chamfer* cluster distance provides the best results in both cases, with more points classified as static, while still maintaining acceptable *EPE3D* and *Acc3DS* scores. The distance and ratio threshold D_{thresh} and R_{thresh} are exactly the same and we also observe that when the sum of histogram distances/similarities are minimized/maximized, more associations are present in both the forward and backward associations, leading to a larger $Ratio_A$.

<i>Bhattacharyya</i>									
	k	ϵ	D_{thresh}	R_{thresh}	$\sum D_{bhat} \downarrow$	$Ratio_A \uparrow$	<i>static</i> $\uparrow$	<i>EPE3D</i> $\downarrow$	<i>Acc3DS</i> $\uparrow$
<i>chamfer</i>	6	0.5	0.3	0.3	0.36	0.329	14025.55	0.0193	0.9673
	24	1.5	0.3	0.3	0.21	0.594	13698.98	0.0216	0.9638
	24	1.0	0.3	0.3	0.22	0.534	13690.49	0.0206	0.9658
	48	1.0	0.3	0.3	0.18	0.639	13671.3	0.0197	0.9671
$L_2-centroid$	6	1.0	0.3	0.3	0.32	0.418	7482.32	0.0204	0.9705
	12	1.0	0.3	0.3	0.26	0.468	7413.25	0.0204	0.9708
	6	0.5	0.3	0.3	0.36	0.327	7386.08	0.0136	0.977
	12	0.5	0.3	0.3	0.28	0.349	7158.99	0.0135	0.977
<i>Intersection</i>									
	k	ϵ	D_{thresh}	R_{thresh}	$\sum D_{\cap} \uparrow$	$Ratio_A \uparrow$	<i>static</i> $\uparrow$	<i>EPE3D</i> $\downarrow$	<i>Acc3DS</i> $\uparrow$
<i>chamfer</i>	6	0.5	0.3	0.3	0.55	0.43	14204.77	0.0193	0.9673
	24	1.5	0.3	0.3	0.67	0.654	13810.11	0.0216	0.9638
	24	1.0	0.3	0.3	0.66	0.59	13752.08	0.0206	0.9658
	6	1.0	0.3	0.3	0.58	0.511	13725.84	0.0207	0.9661
$L_2-centroid$	6	1.0	0.3	0.3	0.58	0.513	7575.77	0.0204	0.9704
	6	0.5	0.3	0.3	0.55	0.431	7571.99	0.0136	0.977
	12	1.0	0.3	0.3	0.62	0.543	7504.3	0.0204	0.9708
	12	0.5	0.3	0.3	0.6	0.435	7336.07	0.0135	0.9769

Table 2: **Clusters classification results.** Shows the best results with regard to the number of points classified as *static* for the *Bhattacharyya* and *Intersection* histogram distance measures and report classification and association metrics averaged over the 142 frames.

3 FH-Net implementation

Fig. 1 shows the Trans-flow layers of FH-Net [19] with all its modules and operations on tensors, omitting the batch dimension for the sake of clarity. The Cross-attention module first performs a ball query to obtain the neighborhoods as explained in Eq. 5.4. The offsets or relative positions between the query points and their neighbors are computed and encoded using an MLP, resulting in the relative position encoding. The source and target features are passed to a Linear transformation and then added and subtracted from the relative position encoding. The result is passed into another MLP, giving the attention scores that are fed to a softmax function, resulting in the final attention weights, as computed in Eq 5.5.

The weighted sum of the offsets using the attention weights outputs the final sparse flow, as shown in the Attentional aggregation module and in Eq 5.6. The

Cross-frame feature enhancement module takes the features of the source point cloud, the relative position encoding with respect to the neighbors, as well as the neighboring features and concatenates them. The concatenation is processed by a Linear transformation and a MAX pooling operation aggregates the new features of the neighbors, producing enhanced features. The layers' width of the MLPs and linear transformations are also shown, where the output of each layer is produced by one or multiple Linear/BatchNorm/ReLU blocks for the MLPs and by a single Linear/BatchNorm block for the Linear transformations.

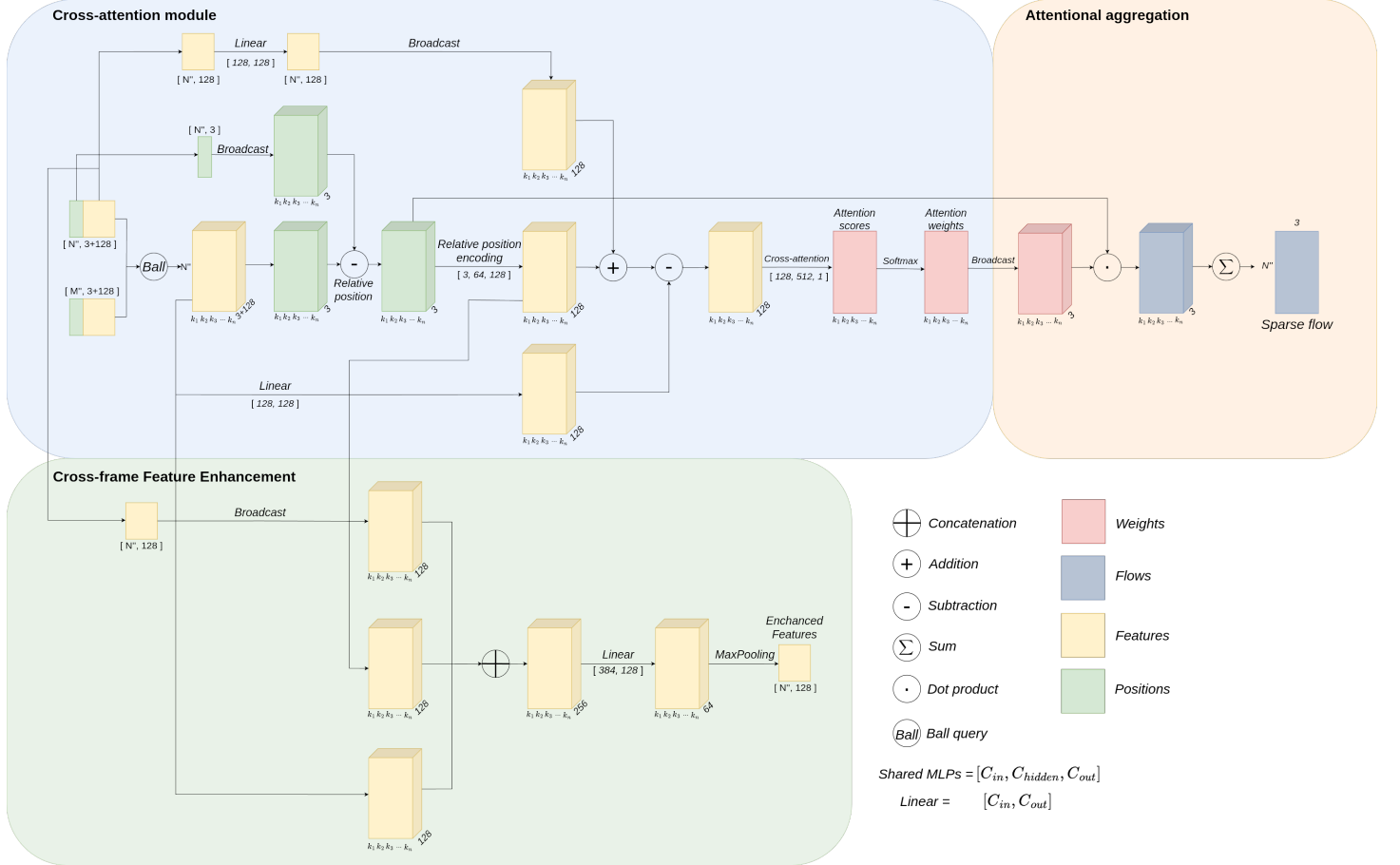


Figure 1: **Trans-flow layer from FH-Net [19].** The Cross-attention module performs the neighborhood grouping and the attention weights computation. The Attentional aggregation module produces the sparse flow using the attention weights and offsets while the Cross-frame feature enhancement updates the features of the source point cloud.

Fig. 2 illustrates the Trans-up layer of FH-Net [19] which is very similar to the Trans-flow layer. The only notable differences are first the neighborhood grouping that uses kNN instead of ball query, and secondly, it aggregates the sparse flow from the Trans-flow layer instead of the relative positions to output the dense flow. The Cross-frame feature enhancement module is abstracted as it is exactly the same as in the Trans-flow layer.

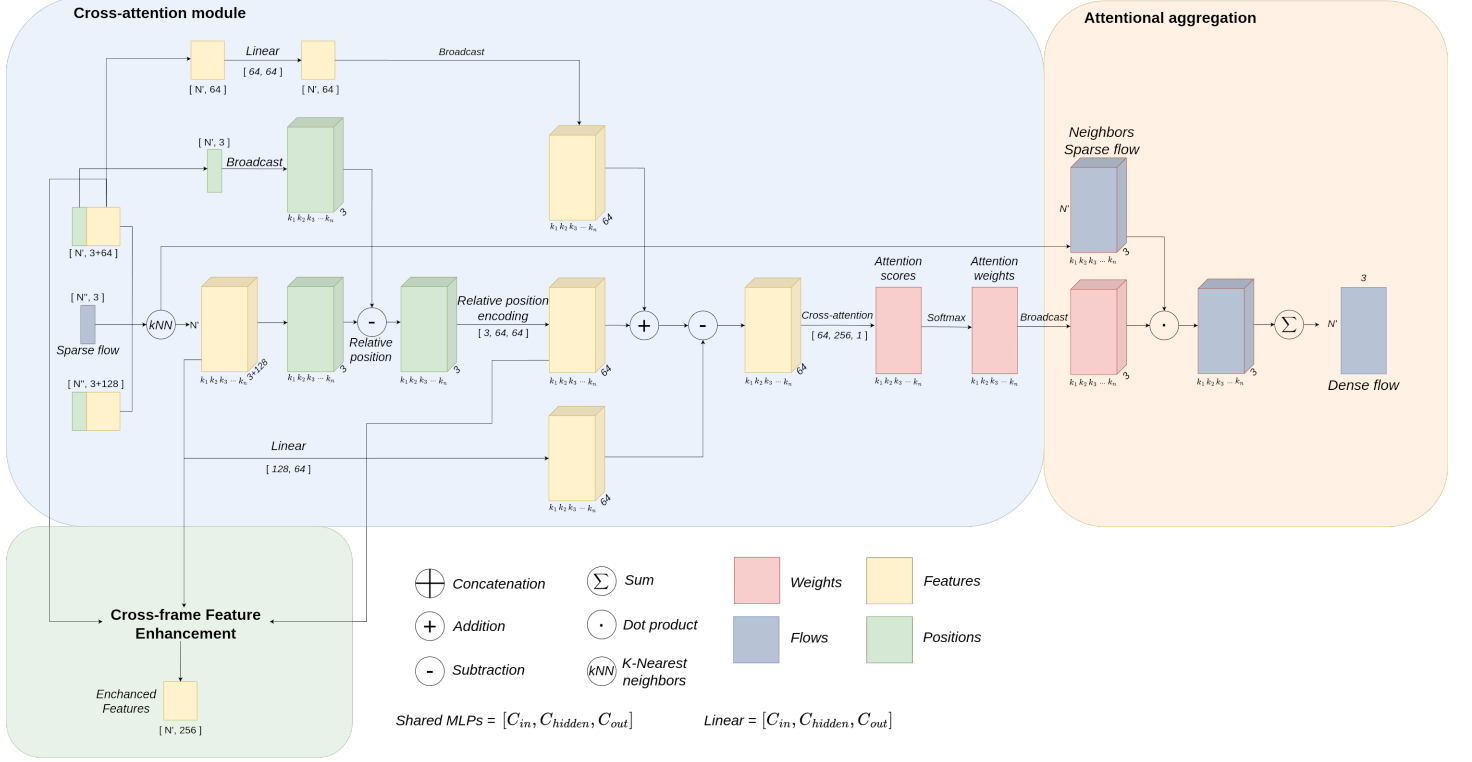


Figure 2: **Trans-up layer from FH-Net [19].** The Cross-attention module is similar to the Trans-Flow one, using kNN instead of ball query. The Attentional aggregation module aggregates the sparse flow vectors using the attention weights to produce the dense flow. The Cross-frame feature enhancement is the same as in the Trans-flow layer.

Fig. 3 depicts the Flow propagation layer which first finds the 3-Nearest neighbors using kNN and then calculates the offsets or relative positions. The weights are measured as the inverse L_2 norm of the offsets with respect to the neighbors. The weights are then used to aggregate the dense flow and the neighboring features that are fed to an MLP to obtain a residual flow. The residuals are then added to the upsampled flow to give the raw resolution flow.

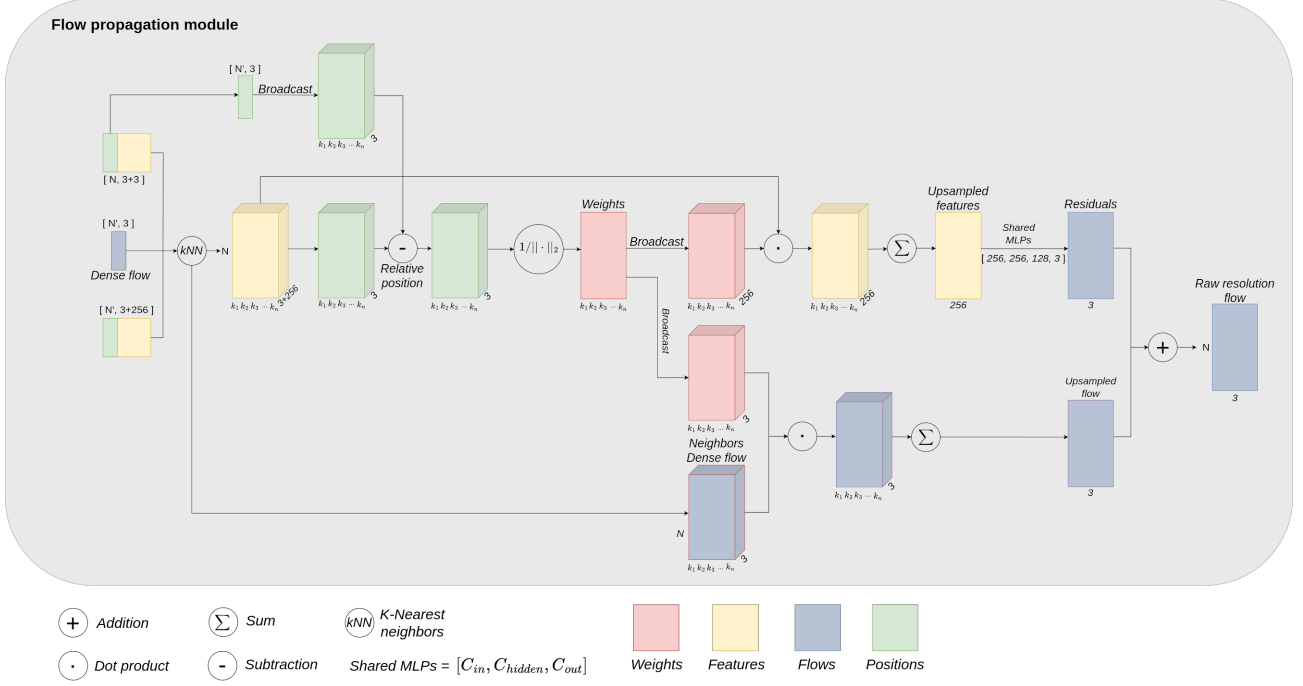


Figure 3: **Flow propagation layer from FH-Net [19]**. The upsampled flow is obtained through a 3-NN interpolation and added to the residual flow obtained from the dense neighboring features passed to an MLP.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl