Supplementary Material: Localization-Guided Foreground Augmentation in Autonomous Driving

Jiawei Yong[1*] Deyuan Qu[2] Qi Chen[2] Kentaro Oguchi[2] Shintaro Fukushima[1]
[1]Toyota Motor Corporation, Japan [2]Toyota Motor North America, USA
jiawei_yong@mail.toyota.co.jp, s_fukushima@mail.toyota.co.jp
deyuan.qu@toyota.com, qi.chen@toyota.com, kentaro.oguchi@toyota.com

1. Overview

This supplementary material provides detailed implementation for the Localization-Guided Foreground Augmentation (LG-FA) framework described in the main paper. The code is organized to correspond with the methodology sections:

- **Section 2**: Coordinate transformation utilities supporting the global vector map construction and frame alignment described in Section 3.1 of the main paper.
- **Section 3**: Implementation of Section 3.2 (Localization and Line Completion), including the class-constrained localization algorithm (Algorithm 1) with SE(2) pose estimation, point-to-segment alignment, and coarse-to-fine scheduling.
- **Section 4**: Implementation of Section 3.3 (Augmented Foreground Perception), covering ego pose reprojection and foreground overlay.
- **Section 5**: Evaluation metrics from Section 4 (Experiments), including Chamfer Distance, Discrete Fréchet Distance, Scale Error, and Completion Rate visualization.
- **Section 6**: Ablation study baseline construction (pose-only baseline for Figure 6).
- **Section 7**: Software dependencies and environment specifications.

Code availability. The complete source code can be made publicly available upon request or if required.

2. Coordinate Transformation

This section provides coordinate transformation utilities that support the global vector map construction (Section 3.1) and frame-level vectorized map processing. These functions enable conversion between ego vehicle frame, camera frame, global scene frame, and image coordinates as described in Eq. (2) of the main paper.

*Corresponding author.

2.1. Ego Pose Extraction

The ego pose is extracted from nuScenes sample data tokens to enable transformation between per-frame ego coordinates and the unified global scene frame.

```python
import numpy as np
from pyquaternion import Quaternion

def get_ego_pose_from_cam_sd(nusc, cam_sd_token):
    """
    Extract ego pose (translation and rotation) from a
    nuScenes camera sample_data token.

    Used for transforming per-frame polylines to the
    unified global scene frame (Eq. 2 in main paper).

    Args:
        nusc: NuScenes database object
        cam_sd_token: Camera sample_data token string

    Returns:
        t_ego: (3,) translation vector [x, y, z]
        R_ego: (3, 3) rotation matrix
    """
    cam_sd = nusc.get('sample_data', cam_sd_token)
    ego_pose = nusc.get('ego_pose',
     cam_sd['ego_pose_token'])

    t_ego = np.array(ego_pose['translation'])
    q = Quaternion(ego_pose['rotation'])
    R_ego = q.rotation_matrix

    return t_ego, R_ego
```

Listing 1. Extract ego pose from nuScenes camera sample data token.

2.2. Homogeneous Transformation Matrix

```python
def build_T(R3x3, t3):
    """
    Build a 4x4 homogeneous transformation matrix from
    rotation matrix and translation vector.

    Args:
        R3x3: (3, 3) rotation matrix
        t3: (3,) or (3, 1) translation vector

    Returns:
        T: (4, 4) homogeneous transformation matrix
```

```
    """
    T = np.eye(4, dtype=np.float64)
    T[:3, :3] = R3x3
    T[:3, 3] = np.asarray(t3).flatten()
    return T
```

Listing 2. Build 4x4 homogeneous transformation matrix.

2.3. Ego-to-Camera Transformation

```
def ego_to_cam_matrix(nusc, cam_sd_token):
    """
    Compute the transformation matrix from ego vehicle
    frame to camera frame. Used in Section 3.3 for
    reprojecting the augmented foreground perception
    to the front-camera view.

    Args:
        nusc: NuScenes database object
        cam_sd_token: Camera sample_data token

    Returns:
        T_e2c: (4, 4) ego-to-camera transformation
        K: (3, 3) camera intrinsic matrix
    """
    cam_sd = nusc.get('sample_data', cam_sd_token)
    cs = nusc.get('calibrated_sensor',
                  cam_sd['calibrated_sensor_token'])

    # Sensor extrinsics (ego -> sensor)
    t_cs = np.array(cs['translation'])
    q_cs = Quaternion(cs['rotation'])
    R_cs = q_cs.rotation_matrix

    # T_ego->camera
    T_e2c = build_T(R_cs.T, -R_cs.T @ t_cs)

    # Camera intrinsics
    K = np.array(cs['camera_intrinsic'])

    return T_e2c, K
```

Listing 3. Compute ego-to-camera transformation matrix for augmented foreground projection.

2.4. BEV to Image Projection

```
def project_xy_to_img(xy_ego, T_e2c, K, img_h, img_w):
    """
    Project 2D BEV points (in ego frame) to image
     pixel
    coordinates. Used for overlaying the
     line-completed
    vector map onto the front-camera view (Figures 3
     and 5).

    Args:
        xy_ego: (N, 2) BEV points [x, y] in ego frame
        T_e2c: (4, 4) ego-to-camera transformation
        K: (3, 3) camera intrinsic matrix
        img_h: Image height in pixels
        img_w: Image width in pixels

    Returns:
        uv: (M, 2) valid pixel coordinates [u, v]
        mask: (N,) boolean mask for valid projections
    """
    N = xy_ego.shape[0]
    # Assume z=0 for BEV points (ground plane)
    pts_ego = np.hstack([xy_ego,
                         np.zeros((N, 1)),
                         np.ones((N, 1))])  # (N, 4)

    # Transform to camera frame
    pts_cam = (T_e2c @ pts_ego.T).T[:, :3]  # (N, 3)

    # Filter points behind camera
    valid_depth = pts_cam[:, 2] > 0.1

    # Project to image plane
    pts_img = (K @ pts_cam.T).T  # (N, 3)
    uv = pts_img[:, :2] / pts_img[:, 2:3]  # (N, 2)

    # Check image bounds
    valid_u = (uv[:, 0] >= 0) & (uv[:, 0] < img_w)
    valid_v = (uv[:, 1] >= 0) & (uv[:, 1] < img_h)

    mask = valid_depth & valid_u & valid_v

    return uv[mask], mask
```

Listing 4. Project BEV map elements to image coordinates for visualization.

2.5. Inter-Frame Vector Transformation

```
def transform_vectors(vectors_dict_xy, T_src2dst):
    """
    Transform polyline vectors from source ego frame
     to
    destination ego frame. Implements the
     transformation
    in Eq. (2): Gamma_t^c = T_t o Gamma_{t,i}^c

    Args:
        vectors_dict_xy: Dict mapping class_id to
     list of
                         polylines, each (K, 2) array
        T_src2dst: (4, 4) transformation from source
     to
                   destination frame

    Returns:
        transformed: Dict with same structure,
     transformed
    """
    R = T_src2dst[:2, :2]
    t = T_src2dst[:2, 3]

    transformed = {}
    for cls_id, polylines in vectors_dict_xy.items():
        transformed[cls_id] = []
        for poly in polylines:
            # Apply 2D rigid transformation
            poly_dst = (R @ poly.T).T + t
            transformed[cls_id].append(poly_dst)

    return transformed
```

Listing 5. Transform polyline vectors between ego frames (Eq. 2).

3. Localization and Line Completion

This section implements the localization and line completion module described in Section 3.2 of the main paper. The core algorithm is the class-constrained 2D localization (Algorithm 1) that estimates ego pose via bidirectional point-to-segment alignment with Huber robust loss.

3.1. SE(2) Pose Representation

We define the 2D pose parameterization $\theta = (t_x, t_y, \phi)$ as described in the main paper, where (t_x, t_y) is the translation and ϕ is the yaw angle.

```python
from dataclasses import dataclass
import numpy as np

@dataclass
class Pose2D:
    """
    2D pose in SE(2) as defined in Section 3.2.
    theta = (t_x, t_y, phi) with planar rotation R(phi)
    and translation t = [t_x, t_y]^T
    """
    tx: float = 0.0
    ty: float = 0.0
    yaw: float = 0.0  # phi in radians

def rotz(yaw):
    """
    Create 3x3 rotation matrix R(phi) for rotation
    about Z-axis (Eq. in Section 3.2).

    Args:
        yaw: Rotation angle phi in radians

    Returns:
        R: (3, 3) rotation matrix
    """
    c, s = np.cos(yaw), np.sin(yaw)
    return np.array([
        [c, -s, 0],
        [s,  c, 0],
        [0,  0, 1]
    ], dtype=np.float64)

def pose_to_T4(p: Pose2D):
    """
    Convert Pose2D to 4x4 homogeneous transformation.
    Implements: X'(theta) = {R(phi)x + t | x in X^c}

    Args:
        p: Pose2D object

    Returns:
        T: (4, 4) transformation matrix
    """
    T = np.eye(4, dtype=np.float64)
    c, s = np.cos(p.yaw), np.sin(p.yaw)
    T[0, 0], T[0, 1] = c, -s
    T[1, 0], T[1, 1] = s, c
    T[0, 3], T[1, 3] = p.tx, p.ty
    return T

def apply_T4_points(pts, T):
    """
    Apply 4x4 transformation to 2D points.

    Args:
        pts: (N, 2) points in source frame
        T: (4, 4) transformation matrix

    Returns:
        pts_out: (N, 2) transformed points
    """
    N = pts.shape[0]
    pts_h = np.hstack([pts,
                       np.zeros((N, 1)),
                       np.ones((N, 1))])
    pts_out = (T @ pts_h.T).T[:, :2]
    return pts_out
```

Listing 6. SE(2) pose representation and utilities.

3.2. Pose Error Computation

```python
def pose_residual_errors(T_est, T_gt):
    """
    Compute translation and heading errors between
    estimated and ground truth poses. Used for
    localization evaluation in Table 2.

    Args:
        T_est: (4, 4) estimated transformation
        T_gt: (4, 4) ground truth transformation

    Returns:
        trans_err: Translation error (meters)
        head_err: Heading error (degrees)
    """
    # Translation error
    t_est = T_est[:2, 3]
    t_gt = T_gt[:2, 3]
    trans_err = np.linalg.norm(t_est - t_gt)

    # Heading error
    yaw_est = np.arctan2(T_est[1, 0], T_est[0, 0])
    yaw_gt = np.arctan2(T_gt[1, 0], T_gt[0, 0])
    head_err = np.abs(yaw_est - yaw_gt)
    head_err = np.minimum(head_err,
                          2 * np.pi - head_err)
    head_err = np.degrees(head_err)

    return trans_err, head_err
```

Listing 7. Compute pose residual errors for evaluation (Table 2).

3.3. Point-to-Segment Distance

The forward term d_f in Eq. (5) measures point-to-segment discrepancy from transformed current-frame sampled points to line segments of the global vector map.

```python
def proj_point_to_seg(p, a, b):
    """
    Project point p onto line segment [a, b].
    Computes d_seg(p, s) in Eq. (5).

    Args:
        p: (2,) query point
        a: (2,) segment start point
        b: (2,) segment end point

    Returns:
        proj: (2,) projected point on segment
        dist: Distance from p to proj (d_seg)
        t: Parameter in [0, 1] along segment
    """
    ab = b - a
    ap = p - a

    ab_sq = np.dot(ab, ab)
    if ab_sq < 1e-12:
        # Degenerate segment
        return a.copy(), np.linalg.norm(ap), 0.0

    t = np.clip(np.dot(ap, ab) / ab_sq, 0.0, 1.0)
    proj = a + t * ab
    dist = np.linalg.norm(p - proj)

    return proj, dist, t

def build_segments_from_poly(poly):
    """
    Convert polyline to list of segments for the
    class-wise segment set S^c in Algorithm 1.

    Args:
        poly: (K, 2) polyline vertices

    Returns:
        segments: List of ((2,), (2,)) segment tuples
    """
    segments = []
    for i in range(len(poly) - 1):
```

```
        segments.append((poly[i], poly[i + 1]))
    return segments
```

Listing 8. Point-to-segment projection for forward term (Eq. 5).

3.4. Class-Constrained Localization (Algorithm 1)

This implements Algorithm 1 from the main paper: iterative class-constrained alternating procedure with bidirectional point-to-segment alignment.

```
from scipy.spatial import cKDTree

def class_constrained_delta_p2seg(
    X_by_c,          # Dict[int, (N_c, 2)] source points
    p2seg_index,     # Dict[int, cKDTree] target segments
    gates,           # Dict[int, float] class thresholds r_c
    weights,         # Dict[int, float] class weights w_c
    huber_k=1.0,     # Robust loss parameter rho
    n_iters=10,      # Max iterations K
    init_pose=None   # Initial pose theta_0
):
    """
    Class-constrained iterative point-to-segment alignment
    implementing Algorithm 1 from the main paper.

    Minimizes the bidirectional objective (Eq. 7):
    theta = argmin sum_c w_c [d_f(X^c, S^c; theta) +
                              d_b(Y^c, X^c; theta)]

    Key components from Algorithm 1:
    - BuildCorr: Builds correspondences P (line 3)
    - RobustWeights: Computes weights w with Huber rho
    - WeightedProcrustes: Solves weighted 2D alignment

    Args:
        X_by_c: Current-frame sampled points by class
                c in {ped crossing, divider, boundary}
        p2seg_index: KDTree for segment search (S^c)
        gates: Per-class gating thresholds r_c
        weights: Per-class importance weights w_c
        huber_k: Huber loss threshold (rho in Eq. 5)
        n_iters: Maximum iterations K
        init_pose: Initial pose theta_0 (GNSS)

    Returns:
        pose: Optimized Pose2D (theta)
        info: Dict with convergence info
    """
    if init_pose is None:
        pose = Pose2D()
    else:
        pose = Pose2D(init_pose.tx, init_pose.ty,
                      init_pose.yaw)

    cls_ids = sorted(X_by_c.keys())

    for it in range(n_iters):
        # Current transformation
        T_curr = pose_to_T4(pose)

        # Accumulate weighted normal equations
        # (WeightedProcrustes step)
        JtWJ = np.zeros((3, 3), dtype=np.float64)
        JtWr = np.zeros(3, dtype=np.float64)
        total_weight = 0.0

        for cid in cls_ids:
            if cid not in X_by_c or cid not in
     p2seg_index:
                continue

            pts_src = X_by_c[cid]
            if len(pts_src) == 0:
                continue

            # Transform source points: X'(theta)
            pts_tf = apply_T4_points(pts_src, T_curr)

            gate = gates.get(cid, 2.0)  # r_c
            w_cls = weights.get(cid, 1.0)  # w_c

            # BuildCorr: Find correspondences
            for i, pt in enumerate(pts_tf):
                # Class-wise correspondence search
                dists, idxs = p2seg_index[cid].query(
                    pt, k=3, distance_upper_bound=gate
                )

                if dists[0] == np.inf:
                    continue  # Skip if no match
    within r_c

                d = dists[0]

                # RobustWeights: Huber weighting (rho)
                if d < huber_k:
                    w_huber = 1.0
                else:
                    w_huber = huber_k / d

                w = w_cls * w_huber

                # Jacobian for WeightedProcrustes
                x, y = pts_src[i]
                c, s = np.cos(pose.yaw),
    np.sin(pose.yaw)

                J = np.array([
                    [1, 0, -s * x - c * y],
                    [0, 1,  c * x - s * y]
                ])

                r = np.array([d, 0])

                JtWJ += w * (J.T @ J)
                JtWr += w * (J.T @ r)
                total_weight += w

        # Check for empty correspondences (line 4-6)
        if total_weight < 1e-6:
            break

        # Solve normal equations (WeightedProcrustes)
        try:
            delta = np.linalg.solve(
                JtWJ + 1e-6 * np.eye(3), -JtWr
            )
        except np.linalg.LinAlgError:
            break

        # Update pose: theta <- theta_new
        pose.tx += delta[0]
        pose.ty += delta[1]
        pose.yaw += delta[2]

        # Check convergence (PoseDiff, line 9-13)
        if np.linalg.norm(delta) < 1e-5:
            break

    info = {'iterations': it + 1,
            'total_weight': total_weight}
    return pose, info
```

Listing 9. Class-constrained iterative alignment (Algorithm 1).

3.5. Coarse-to-Fine Schedule

As described in Section 3.2, we use a coarse-to-fine schedule: a coarse stage using only road boundary geometry with a relaxed gate, followed by an all-class refinement stage with tighter class-wise gates.

```python
def hierarchical_delta_p2seg(X_by_c, p2seg_index, cfg):
    """
    Two-stage coarse-to-fine alignment as described in Section 3.2 of the main paper.

    Stage 1 (Coarse): Road boundary only, relaxed gate (4.0m), 8 iterations
    Stage 2 (Fine): All classes with tight thresholds (1.5/1.5/1.8m), 15 iterations

    Class IDs: 0=ped crossing, 1=divider, 2=boundary

    Args:
        X_by_c: Source points by class
        p2seg_index: Target segment KDTree indices
        cfg: Configuration dict with schedule parameters

    Returns:
        pose: Final optimized Pose2D
        info: Dict with stage-wise info
    """
    # Default configuration matching Section 4.1
    coarse_gate = cfg.get('coarse_gate', 4.0)
    coarse_iters = cfg.get('coarse_iters', 8)
    fine_gates = cfg.get('fine_gates', {0: 1.5, 1: 1.5, 2: 1.8})
    fine_iters = cfg.get('fine_iters', 15)
    huber_k = cfg.get('huber_k', 1.0)

    # Class weights (boundary more reliable)
    weights = {0: 1.0, 1: 1.0, 2: 1.2}

    # Stage 1: Coarse alignment using boundary only
    X_boundary = {2: X_by_c.get(2, np.zeros((0, 2)))}
    idx_boundary = {2: p2seg_index.get(2)}

    if idx_boundary[2] is not None:
        gates_coarse = {2: coarse_gate}
        pose_coarse, info1 = class_constrained_delta_p2seg(
            X_boundary, idx_boundary,
            gates_coarse, weights,
            huber_k=huber_k,
            n_iters=coarse_iters
        )
    else:
        pose_coarse = Pose2D()
        info1 = {}

    # Stage 2: Fine alignment using all classes
    pose_fine, info2 = class_constrained_delta_p2seg(
        X_by_c, p2seg_index,
        fine_gates, weights,
        huber_k=huber_k,
        n_iters=fine_iters,
        init_pose=pose_coarse
    )

    info = {
        'coarse': info1,
        'fine': info2,
        'coarse_pose': pose_coarse,
        'final_pose': pose_fine
    }

    return pose_fine, info
```

Listing 10. Hierarchical coarse-to-fine alignment schedule.

4. Augmented Foreground Perception

This section implements the augmented foreground perception described in Section 3.3 of the main paper. After localization, LG-FA reprojects the current-frame ego vehicle and detected objects into the coordinate frame of the completed map, forming the augmented foreground perception shown in Figure 3.

4.1. Foreground Reprojection

```python
def reproject_to_map_frame(
    ego_pose_refined,     # Refined pose from Section 3.2
    detections,           # Detected objects in ego frame
    map_polylines         # Line-completed vector map V_t^c
):
    """
    Reproject the current-frame ego vehicle and detected
    objects into the coordinate frame of the completed
    map (Section 3.3).

    This creates the augmented foreground perception where
    ego position and objects share a common global frame
    with the completed lane dividers, road boundaries,
    and pedestrian crossings.

    Args:
        ego_pose_refined: Pose2D from localization
        detections: List of detection dicts with
                    'bbox', 'class', 'position'
        map_polylines: Dict[cls_id, List[(K,2)]]

    Returns:
        augmented: Dict containing:
            - 'ego_in_map': Ego position in map frame
            - 'detections_in_map': Transformed detections
            - 'map_elements': The completed vector map
    """
    T_ego2map = pose_to_T4(ego_pose_refined)

    # Ego position in map frame
    ego_origin = np.array([[0.0, 0.0]])
    ego_in_map = apply_T4_points(ego_origin, T_ego2map)[0]

    # Transform detections to map frame
    detections_in_map = []
    for det in detections:
        pos_ego = np.array([det['position'][:2]])
        pos_map = apply_T4_points(pos_ego, T_ego2map)[0]

        det_transformed = {
            'class': det['class'],
            'position': pos_map,
            'bbox': det.get('bbox', None)
        }
        detections_in_map.append(det_transformed)

    augmented = {
        'ego_in_map': ego_in_map,
        'detections_in_map': detections_in_map,
        'map_elements': map_polylines
    }

    return augmented
```

Listing 11. Reproject ego vehicle and detections onto the completed map (Section 3.3).

4.2. Augmented Perception Overlay

```python
import cv2

def overlay_augmented_foreground(
    image,                # Front camera image
    augmented,            # Output from reproject_to_map_frame
    T_map2cam,            # Map frame to camera transform
    K,                    # Camera intrinsics
    colors=None           # Color scheme for classes
):
    """
    Overlay the augmented foreground perception onto the
    front-camera view as shown in Figure 3 and Figure 5.

    Red: Lane dividers
    Green: Road boundaries
    Blue: Pedestrian crossings
    Yellow: Detected objects

    Args:
        image: (H, W, 3) front camera image
        augmented: Augmented perception dict
        T_map2cam: (4, 4) map to camera transform
        K: (3, 3) camera intrinsic matrix
        colors: Optional custom color dict

    Returns:
        overlay: (H, W, 3) image with overlays
    """
    if colors is None:
        colors = {
            0: (255, 0, 0),     # Blue - ped crossing
            1: (0, 0, 255),     # Red - divider
            2: (0, 255, 0),     # Green - boundary
            'det': (0, 255, 255)  # Yellow - detections
        }

    overlay = image.copy()
    h, w = image.shape[:2]

    # Draw map elements (completed polylines)
    for cls_id, polylines in augmented['map_elements'].items():
        color = colors.get(cls_id, (255, 255, 255))

        for poly in polylines:
            uv, mask = project_xy_to_img(
                poly, T_map2cam, K, h, w
            )
            if len(uv) >= 2:
                pts = uv.astype(np.int32)
                cv2.polylines(overlay, [pts], False,
                              color, thickness=2)

    # Draw detected objects
    det_color = colors['det']
    for det in augmented['detections_in_map']:
        pos = np.array([det['position']])
        uv, mask = project_xy_to_img(
            pos, T_map2cam, K, h, w
        )
        if mask[0]:
            pt = tuple(uv[0].astype(int))
            cv2.circle(overlay, pt, 5, det_color, -1)

    # Draw ego position
    ego_pos = np.array([augmented['ego_in_map']])
    uv_ego, mask_ego = project_xy_to_img(
        ego_pos, T_map2cam, K, h, w
    )
    if mask_ego[0]:
        pt = tuple(uv_ego[0].astype(int))
        cv2.drawMarker(overlay, pt, (255, 255, 0),
                       cv2.MARKER_DIAMOND, 10, 2)

    return overlay
```

Listing 12. Overlay augmented foreground onto front-camera view (Figures 3 and 5).

5. Evaluation Metrics

This section implements the evaluation metrics described in Section 4 (Experiments) of the main paper, including Chamfer Distance and Discrete Fréchet Distance for map quality evaluation (Table 1), Scale Error following ORB-SLAM3, and Completion Rate for the ablation study (Figure 6).

5.1. Chamfer Distance

Chamfer Distance measures the average bidirectional point-wise deviation between predicted and ground-truth polylines (Section 4.1).

```python
from scipy.spatial import cKDTree

def chamfer_distance(xy_a, xy_b):
    """
    Compute symmetric Chamfer Distance between two
    point sets. Used for map quality evaluation in
    Table 1 of the main paper.

    CD(A, B) = mean(d(a, B)) + mean(d(b, A))

    "Representative of the overall geometric accuracy
    for the majority of scenes" - Section 4.1

    Args:
        xy_a: (N, 2) predicted point set
        xy_b: (M, 2) ground truth point set

    Returns:
        cd: Chamfer Distance (meters)
    """
    if len(xy_a) == 0 or len(xy_b) == 0:
        return float('inf')

    tree_a = cKDTree(xy_a)
    tree_b = cKDTree(xy_b)

    # A to B
    dist_a2b, _ = tree_b.query(xy_a, k=1)
    # B to A
    dist_b2a, _ = tree_a.query(xy_b, k=1)

    cd = np.mean(dist_a2b) + np.mean(dist_b2a)
    return cd
```

Listing 13. Symmetric Chamfer Distance for map quality evaluation (Table 1).

5.2. Discrete Fréchet Distance

Fréchet Distance is sensitive to worst-case deviations along a polyline, capturing endpoint mismatches and gap-

bridging mistakes (Section 4.1).

```python
def discrete_frechet(P, Q):
    """
    Compute Discrete Frechet Distance between two
    polylines using dynamic programming.

    "Sensitive to worst-case deviations along a
     polyline
    and can be dominated by a small number of endpoint
    mismatches, gap-bridging mistakes, or ordering
    inconsistencies" - Section 4.1

    Args:
        P: (N, 2) predicted polyline
        Q: (M, 2) ground truth polyline

    Returns:
        dfd: Discrete Frechet Distance (meters)
    """
    n, m = len(P), len(Q)
    if n == 0 or m == 0:
        return float('inf')

    # Compute pairwise distances
    D = np.zeros((n, m), dtype=np.float64)
    for i in range(n):
        for j in range(m):
            D[i, j] = np.linalg.norm(P[i] - Q[j])

    # Dynamic programming table
    dp = np.full((n, m), np.inf, dtype=np.float64)
    dp[0, 0] = D[0, 0]

    # Fill first row
    for j in range(1, m):
        dp[0, j] = max(dp[0, j-1], D[0, j])

    # Fill first column
    for i in range(1, n):
        dp[i, 0] = max(dp[i-1, 0], D[i, 0])

    # Fill rest of table
    for i in range(1, n):
        for j in range(1, m):
            dp[i, j] = max(
                min(dp[i-1, j], dp[i, j-1], dp[i-1,
     j-1]),
                D[i, j]
            )

    return dp[n-1, m-1]
```

Listing 14. Discrete Fréchet Distance via dynamic programming (Table 1).

5.3. Scale Error

Scale Error is computed following ORB-SLAM3, measuring the scale factor after similarity alignment between predicted and ground truth maps.

```python
def umeyama_scale_2d(src, dst):
    """
    Estimate optimal scale factor using Umeyama
     alignment.
    Following ORB-SLAM3 for scale error computation.

    "Global Scale Error in percent, defined as |1 -
     s| x 100,
    following ORB-SLAM3, where s is the estimated
     scale
    factor after similarity alignment" - Section 4.1

    Args:
        src: (N, 2) source/predicted points
        dst: (N, 2) destination/ground truth points

    Returns:
        scale: Estimated scale factor s
        scale_error: |1 - s| * 100 as percentage
    """
    assert len(src) == len(dst) and len(src) >= 2

    # Center the point sets
    mu_src = np.mean(src, axis=0)
    mu_dst = np.mean(dst, axis=0)

    src_c = src - mu_src
    dst_c = dst - mu_dst

    # Compute variances
    var_src = np.sum(src_c ** 2) / len(src)

    # Compute covariance
    cov = (dst_c.T @ src_c) / len(src)

    # SVD
    U, S, Vt = np.linalg.svd(cov)

    # Scale factor
    scale = np.sum(S) / var_src if var_src > 1e-9
     else 1.0

    # Scale error as percentage
    scale_error = abs(1.0 - scale) * 100.0

    return scale, scale_error
```

Listing 15. Scale error estimation following ORB-SLAM3 (Section 4.1).

5.4. Completion Rate Visualization

This implements the completion rate metric and CVPR-style visualization used in Figure 6 of the main paper for the ablation study comparing LG-FA against the pose-only baseline.

```python
#!/usr/bin/env python3
"""
CVPR-style completion rate bar chart for LG-FA
    ablation.
Generates Figure 6 comparing w/ and w/o LG-FA.

"Completion rate, defined as the ratio of detected
geometric elements relative to the maximum achievable
coverage, averaged across all validation scenes"
- Section 4.4
"""
import os
import re
import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path

def read_points_txt_counts(path):
    """
    Read point counts per class from prediction file.

    Args:
        path: Path to pred_points.txt file

    Returns:
        cnt: Dict mapping class_id to point count
    """
    cnt = {}
    with open(path, 'r') as f:
        for ln in f:
            ln = ln.strip()
            if not ln or ln.startswith('#'):
                continue
```

```
            parts = re.split(r'[,\s]+', ln)
            if len(parts) < 4:
                continue
            try:
                cid = int(float(parts[3]))
                cnt[cid] = cnt.get(cid, 0) + 1
            except:
                continue
    return cnt

def compute_completion_rates(baseline_root, lgfa_root):
    """
    Compute per-class completion rates across scenes.

    Args:
        baseline_root: Path to baseline predictions
        lgfa_root: Path to LG-FA predictions

    Returns:
        r_baseline: List of baseline rates per class
        r_lgfa: List of LG-FA rates per class
    """
    cls_ids = [0, 1, 2]  # crossing, divider, boundary
    scenes = sorted([d for d in os.listdir(baseline_root)
                     if d.startswith('scene-')])
    common = [s for s in scenes
              if (lgfa_root/s/'pred_points.txt').exists()]

    ratios = {0: [], 1: [], 2: []}

    for scene in common:
        b_path = baseline_root / scene / 'pred_points.txt'
        l_path = lgfa_root / scene / 'pred_points.txt'

        b_cnt = read_points_txt_counts(b_path)
        l_cnt = read_points_txt_counts(l_path)

        for cid in cls_ids:
            b_pts = b_cnt.get(cid, 0)
            l_pts = l_cnt.get(cid, 0)

            if b_pts > 0 or l_pts > 0:
                max_pts = max(b_pts, l_pts)
                b_rate = b_pts / max_pts * 100
                l_rate = l_pts / max_pts * 100
                ratios[cid].append((b_rate, l_rate))

    r_baseline, r_lgfa = [], []
    for cid in cls_ids:
        if ratios[cid]:
            r_baseline.append(
                np.mean([r[0] for r in ratios[cid]]))
            r_lgfa.append(
                np.mean([r[1] for r in ratios[cid]]))
        else:
            r_baseline.append(0)
            r_lgfa.append(0)

    return r_baseline, r_lgfa

def plot_completion_chart(y_wo, y_w, output_path):
    """
    Generate CVPR-style bar chart (Figure 6).

    Args:
        y_wo: Array of rates without LG-FA
        y_w: Array of rates with LG-FA
        output_path: Path to save figure
    """
    plt.rcParams.update({
        'font.family': 'serif',
        'font.serif': ['Times New Roman', 'DejaVu Serif'],
        'font.size': 18,
        'axes.labelsize': 20,
        'axes.titlesize': 20,
        'xtick.labelsize': 14,
        'ytick.labelsize': 16,
        'legend.fontsize': 16,
        'figure.dpi': 150,
        'savefig.dpi': 300,
        'axes.linewidth': 1.2,
        'axes.spines.top': False,
        'axes.spines.right': False,
    })

    LABELS = ['Ped.\ncrossing', 'Divider',
              'Boundary', 'Avg\n(all)']

    fig, ax = plt.subplots(figsize=(10, 6))

    x = np.arange(len(LABELS))
    width = 0.36

    color_wo = '#4878A8'   # Steel blue
    color_w = '#E8893C'    # Orange

    ax.bar(x - width/2, y_wo, width,
           label='w/o LG-FA', color=color_wo,
           edgecolor='white', linewidth=0.8)
    ax.bar(x + width/2, y_w, width,
           label='w/ LG-FA', color=color_w,
           edgecolor='white', linewidth=0.8)

    ax.set_ylabel('Completion Rate (%)',
                  fontsize=20, fontweight='medium')
    ax.set_ylim(0, 125)
    ax.set_yticks([0, 20, 40, 60, 80, 100])
    ax.yaxis.grid(True, linestyle='--', alpha=0.4)

    ax.set_xticks(x)
    ax.set_xticklabels(LABELS, fontsize=14)

    # Separator between per-class and summary
    ax.axvline(2.5, ls='--', lw=1.5, c='gray',
               alpha=0.6)
    ax.text(1.0, 122, 'Per-class', ha='center',
            fontsize=16, fontweight='bold',
            color='#333')
    ax.text(3.0, 122, 'Summary', ha='center',
            fontsize=16, fontweight='bold',
            color='#333')

    ax.legend(loc='upper center',
              bbox_to_anchor=(0.5, -0.12),
              ncol=2, frameon=False, fontsize=16)

    # Delta annotations with brackets
    for i in range(len(LABELS)):
        delta = y_w[i] - y_wo[i]
        top = max(y_wo[i], y_w[i]) + 2

        xl, xr = x[i] - width/2, x[i] + width/2
        bracket_h = 3
        ax.plot([xl, xl, xr, xr],
                [top, top+bracket_h, top+bracket_h, top],
                lw=1.2, c='#333',
                solid_capstyle='round')

        ax.text(x[i], top + bracket_h + 1.5,
                f'+{delta:.1f}%', ha='center',
                fontsize=14, fontweight='bold',
                color='#2E7D32')

    ax.set_xlim(-0.6, len(LABELS) - 0.4)
    plt.tight_layout(rect=[0, 0.06, 1, 0.96])

    plt.savefig(output_path, bbox_inches='tight',
                dpi=300)
    plt.close()
```

Listing 16. Completion rate computation and visualization (Figure 6).

6. Ablation Study: Pose-Only Baseline

This section implements the pose-only baseline used for ablation comparison in Figure 6 and Section 4.4 of the main paper. This baseline aggregates per-frame predictions using raw nuScenes ego poses without map-based alignment or line completion.

6.1. Vector Denormalization

```python
def denorm_if_needed(vecs_xy, roi_size, origin):
    """
    Denormalize vectors from [0, 1] to metric coordinates.

    Args:
        vecs_xy: List of (K, 2) polylines
        roi_size: (width, height) of ROI in meters
        origin: (x, y) origin offset in meters

    Returns:
        vecs_metric: List of polylines in metric coords
    """
    if len(vecs_xy) == 0:
        return vecs_xy

    all_pts = np.vstack(vecs_xy)
    if np.all(all_pts >= -0.1) and np.all(all_pts <= 1.1):
        vecs_metric = []
        for vec in vecs_xy:
            vec_m = vec * np.array(roi_size) + np.array(origin)
            vecs_metric.append(vec_m)
        return vecs_metric

    return vecs_xy
```

Listing 17. Denormalize vectors from normalized coordinates.

6.2. Pose-Only Baseline Construction

```python
def build_pose_only_baseline(
    nusc,
    preds,
    scene_name,
    roi_size=(60.0, 30.0),
    origin=(-30.0, -15.0)
):
    """
    Aggregate per-frame predictions using raw ego poses
    only - NO map-based alignment or line completion.

    "The baseline builds the global vector map by directly
    transforming and merging per-frame predictions using
    the raw nuScenes ego poses, i.e., without alignment
    (map-based pose refinement) and without line
    completion." - Section 4.4

    Args:
        nusc: NuScenes database object
        preds: Dict mapping frame tokens to predictions
        scene_name: Scene name string
        roi_size: ROI dimensions in meters
        origin: ROI origin offset

    Returns:
        aggregated: Dict[cls_id, List[(K, 2)]] merged
                    polylines in anchor frame
    """
    scene = [s for s in nusc.scene
             if s['name'] == scene_name][0]

    sample_tokens = []
    sample_token = scene['first_sample_token']
    while sample_token:
        sample_tokens.append(sample_token)
        sample = nusc.get('sample', sample_token)
        sample_token = sample['next']

    if len(sample_tokens) == 0:
        return {}

    # Use last frame as anchor
    anchor_token = sample_tokens[-1]
    anchor_sample = nusc.get('sample', anchor_token)
    anchor_cam_token = anchor_sample['data']['CAM_FRONT']
    t_anchor, R_anchor = get_ego_pose_from_cam_sd(
        nusc, anchor_cam_token
    )
    T_anchor = build_T(R_anchor, t_anchor)
    T_anchor_inv = np.linalg.inv(T_anchor)

    aggregated = {0: [], 1: [], 2: []}

    for sample_token in sample_tokens:
        if sample_token not in preds:
            continue

        sample = nusc.get('sample', sample_token)
        cam_token = sample['data']['CAM_FRONT']

        t_ego, R_ego = get_ego_pose_from_cam_sd(
            nusc, cam_token
        )
        T_ego = build_T(R_ego, t_ego)

        # Raw pose transformation (no refinement)
        T_frame2anchor = T_anchor_inv @ T_ego

        frame_preds = preds[sample_token]

        for cls_id, vecs in frame_preds.items():
            if cls_id not in aggregated:
                continue

            vecs_m = denorm_if_needed(vecs, roi_size, origin)

            vecs_anchor = transform_vectors(
                {cls_id: vecs_m}, T_frame2anchor
            )[cls_id]

            aggregated[cls_id].extend(vecs_anchor)

    return aggregated
```

Listing 18. Build pose-only baseline for ablation (Figure 6, Section 4.4).

7. Dependencies

The implementation requires the following software dependencies:

Notes:

- SciPy is required for `cKDTree` used in efficient nearest-neighbor queries for class-constrained correspondence search (Algorithm 1).

Table 1. Software dependencies and minimum versions.

Package	Version
Python	≥ 3.8
NumPy	≥ 1.20
SciPy	≥ 1.7
OpenCV	≥ 4.5
Pillow	≥ 8.0
nuScenes-devkit	≥ 1.1
pyquaternion	≥ 0.9
Matplotlib	≥ 3.4

- Matplotlib is required for the completion rate visualization (Figure 6).
- The nuScenes-devkit is used for dataset access and ego pose extraction.

Installation:

```
pip install numpy>=1.20 scipy>=1.7 opencv-python>=4.5
pip install Pillow>=8.0 pyquaternion>=0.9
pip install nuscenes-devkit>=1.1 matplotlib>=3.4
```

Listing 19. Install dependencies via pip.