import cv2import numpy as npclass Calibrator: def \_\_init\_\_(self, params, extrinsic, pcs): self.params\_ = params self.extrinsic\_ = extrinsic self.pcs\_ = pcs # Point cloud for different indexes def get\_final\_transformation(self): return self.extrinsic\_ def visual\_projection(self, T, save\_name, index): # Load the image img\_color = cv2.imread(self.params\_['img\_files'][index]) if img\_color is None: print(f"Cannot read {self.params\_['img\_files'][index]}") return # Undistort the image if necessary if self.params\_['intrinsic'].shape[1] == 3: img\_color = self.undistort\_img(img\_color, self.params\_['intrinsic'], self.params\_['dist']) lidar\_points = [] # Project point cloud onto image for src\_pt in self.pcs\_[index]: vec = np.array([src\_pt[0], src\_pt[1], src\_pt[2], 1.0]) if [self.pro](self.pro#self.pro)ject\_on\_image(vec, T): x, y = [self.pro](self.pro#self.pro)ject\_on\_image(vec, T) lidar\_point = (int(x), int(y)) lidar\_points.append(lidar\_point) # Draw lidar points on the image for point in lidar\_points: cv2.circle(img\_color, point, 3, (0, 255, 0), -1) # Save the image cv2.imwrite(save\_name, img\_color) print(f"Image saved: {save\_name}") def project\_on\_image(self, vec, T): # Projection logic to map 3D points to 2D image plane # Replace this with the actual logic based on your transformation and camera model vec\_transformed = T @ vec x, y = vec\_transformed[0] / vec\_transformed[2], vec\_transformed[1] / vec\_transformed[2] # Perspective division if 0 <= x < self.params\_['image\_width'] and 0 <= y < self.params\_['image\_height']: return int(x), int(y) return None def undistort\_img(self, img\_color, intrinsic, dist): # Undistort the image using OpenCV's undistort function h, w = img\_color.shape[:2] new\_intrinsic, \_ = cv2.getOptimalNewCameraMatrix(intrinsic, dist, (w, h), 1, (w, h)) undistorted\_img = cv2.undistort(img\_color, intrinsic, dist, None, new\_intrinsic) return undistorted\_imgimport numpy as npimport cv2class Calibrator: def \_\_init\_\_(self, params, IMG\_W, IMG\_H): self.params\_ = params self.IMG\_W = IMG\_W self.IMG\_H = IMG\_H def project\_on\_image(self, vec, T, margin=0): # Apply transformation matrix T and project the point if self.params\_['intrinsic'].shape[1] == 4: vec2 = self.params\_['intrinsic'] @ T @ vec else: cam\_point = T @ vec cam\_vec = cam\_point[:3] # Take x, y, z from the transformed point vec2 = self.params\_['intrinsic'] @ cam\_vec # Check if the point is in front of the camera if vec2[2] <= 0: return False, None, None # Normalize to get image coordinates x = int(np.round(vec2[0] / vec2[2])) y = int(np.round(vec2[1] / vec2[2])) # Check if the projected point falls within the image bounds, considering the margin if -margin <= x < self.IMG\_W + margin and -margin <= y < self.IMG\_H + margin: return True, x, y return False, None, Noneimport numpy as npclass Calibrator: def \_\_init\_\_(self, params, extrinsic, pcs, IMG\_W, IMG\_H): self.params\_ = params self.extrinsic\_ = extrinsic self.pcs\_ = pcs # Point clouds self.IMG\_W = IMG\_W self.IMG\_H = IMG\_H self.POINT\_PER\_PIXEL = 0 def cal\_ratio(self): lidar\_points = [] point\_num = 0 assert self.params\_['point\_range\_top'] < self.params\_['point\_range\_bottom'], \ "Top range should be less than bottom range" top = int(self.params\_['point\_range\_top'] \* self.IMG\_H) bottom = int(self.params\_['point\_range\_bottom'] \* self.IMG\_H) # Iterate through all the points in the first point cloud (pcs\_[0]) for src\_pt in self.pcs\_[0]: vec = np.array([src\_pt[0], src\_pt[1], src\_pt[2], 1.0]) success, x, y = [self.pro](self.pro#self.pro)ject\_on\_image(vec, self.extrinsic\_) if success and top < y < bottom: point\_num += 1 # Calculate the area within the defined range area = (self.params\_['point\_range\_bottom'] - self.params\_['point\_range\_top']) \* self.IMG\_H \* self.IMG\_W # Compute the point per pixel ratio self.POINT\_PER\_PIXEL = point\_num / area print(f"Estimated point number per pixel: {self.POINT\_PER\_PIXEL}") def project\_on\_image(self, vec, T, margin=0): # Projection logic (reuse the method defined earlier) if self.params\_['intrinsic'].shape[1] == 4: vec2 = self.params\_['intrinsic'] @ T @ vec else: cam\_point = T @ vec cam\_vec = cam\_point[:3] vec2 = self.params\_['intrinsic'] @ cam\_vec if vec2[2] <= 0: return False, None, None x = int(np.round(vec2[0] / vec2[2])) y = int(np.round(vec2[1] / vec2[2])) if -margin <= x < self.IMG\_W + margin and -margin <= y < self.IMG\_H + margin: return True, x, y return False, None, Noneimport cv2import numpy as np# Initialize the color bar matrixcolor\_bar = np.zeros((1, 13 \* 3 \* 3, 3), dtype=np.uint8)def create\_color\_bar(): # Hue, Saturation, and Value arrays H = [180, 120, 60, 160, 100, 40, 150, 90, 30, 140, 80, 20, 10] S = [255, 100, 30] V = [255, 180, 90] # Create an empty matrix for the HSV color bar color = np.zeros((1, 13 \* 3 \* 3, 3), dtype=np.uint8) h = 0 s = 0 v = 0 # Fill the HSV color bar for ba in range(13 \* 3 \* 3): h = ba % 13 s = (ba // 13) % 3 v = (ba // (13 \* 3)) % 3 color[0, ba, 0] = H[h] # Hue color[0, ba, 1] = S[s] # Saturation color[0, ba, 2] = V[v] # Value # Convert from HSV to BGR global color\_bar color\_bar = cv2.cvtColor(color, cv2.COLOR\_HSV2BGR)# Call the function to create the color barcreate\_color\_bar()# Optionally, display the color bar to verifycv2.imshow("Color Bar", color\_bar)cv2.waitKey(0)cv2.destroyAllWindows()import cv2import numpy as npimport pcl # Assuming pcl is available or using Open3D as an alternativeclass Calibrator: def \_\_init\_\_(self, json\_params): self.params\_ = json\_params self.extrinsic\_ = json\_params['extrinsic'] self.masks\_ = [] self.mask\_point\_num\_ = [] self.n\_mask\_ = [] self.pcs\_ = [] print("----------Start processing data----------") # Load image img = cv2.imread(self.params\_['img\_files'][0]) if img is None: print(f"Can not read {self.params\_['img\_files'][0]}") exit(1) self.IMG\_H, self.IMG\_W = img.shape[:2] # Process each data file for i in range(self.params\_['N\_FILE']): print(f"Processing data {i + 1}:") # Load mask file mask\_point\_num = [] masks = np.zeros((self.IMG\_H, self.IMG\_W, 4), dtype=np.uint8) DataLoader.load\_mask\_file(self.params\_['mask\_dirs'][i], self.params\_['intrinsic'], self.params\_['dist'], masks, mask\_point\_num) self.masks\_.append(masks) self.mask\_point\_num\_.append(mask\_point\_num) self.n\_mask\_.append(len(mask\_point\_num)) # Load lidar file pc\_origin = pcl.PointCloud() # Create an empty point cloud pc = pcl.PointCloud() # Create a new point cloud for processed data DataLoader.load\_lidar\_file(self.params\_['lidar\_files'][i], pc\_origin) # Process the point cloud [self.pro](self.pro#self.pro)cess\_pointcloud(pc\_origin, pc) self.pcs\_.append(pc) # Create the color bar self.create\_color\_bar() def process\_pointcloud(self, pc\_origin, pc): # Processing point cloud logic (similar to the ProcessPointcloud method previously converted) pass def create\_color\_bar(self): # Create color bar logic (similar to the Create\_ColorBar method previously converted) passclass DataLoader: @staticmethod def load\_mask\_file(mask\_dir, intrinsic, dist, masks, mask\_point\_num): # Logic to load and process the mask files goes here pass @staticmethod def load\_lidar\_file(lidar\_file, pc\_origin): # Logic to load lidar files into the point cloud goes here passimport numpy as npimport pclimport pcl.pcl\_visualizationfrom pcl import VoxelGrid, NormalEstimationfrom pcl import PointCloud, PointIndicesfrom pcl import octreeclass Calibrator: def process\_pointcloud(self, pc\_origin, pc): # Pre-filter points by initial extrinsic pc\_filtered = PointCloud() # Create a new point cloud margin = 300 intensity\_max = 1 vars = [] if self.params\_['search\_range\_rot'] > 10 or self.params\_['search\_range\_trans'] > 1: print("Search range too large. Don't support!") Util.gen\_vars(1, np.deg2rad(self.params\_['search\_range\_rot']), 1, self.params\_['search\_range\_trans'], vars) # Filter the points based on the projection onto the image for src\_pt in pc\_origin: if not np.isfinite(src\_pt[0]) or not np.isfinite(src\_pt[1]) or not np.isfinite(src\_pt[2]): continue vec = np.array([src\_pt[0], src\_pt[1], src\_pt[2], 1]) x, y = 0, 0 for var in vars: extrinsic = np.dot(self.extrinsic\_, Util.get\_delta\_t(var)) if [self.pro](self.pro#self.pro)ject\_on\_image(vec, extrinsic, margin): intensity\_max = max(intensity\_max, src\_pt[3]) pc\_filtered.append(src\_pt) break print(f"Point cloud num: {len(pc\_filtered)}") # Down-sample if needed if self.params\_['is\_down\_sample']: cloud\_downsampled = PointCloud() cloud\_tmp = PointCloud() cloud\_tmp\_ds = PointCloud() kLeafSize = 0.1 filter\_map = VoxelGrid() filter\_map.set\_leaf\_size(kLeafSize, kLeafSize, kLeafSize) cloud\_downsampled.clear() # Using octree for downsampling octree\_map = octree.OctreePointCloudSearch(kLeafSize \* 1250) octree\_map.set\_input\_cloud(pc\_filtered) octree\_map.add\_points\_from\_input\_cloud() for it in octree\_map.leaf\_iterator(): ids = it.get\_leaf\_container().get\_point\_indices\_vector() cloud\_tmp.clear() for id in ids: cloud\_tmp.append(pc\_filtered[id]) filter\_map.set\_input\_cloud(cloud\_tmp) filter\_map.filter(cloud\_tmp\_ds) cloud\_downsampled += cloud\_tmp\_ds pc\_filtered = cloud\_downsampled print(f"Points num after downsample: {len(pc\_filtered)}") # Segment point cloud normals = PointCloud() seg\_indices = [] n\_seg = self.segment\_pc(pc\_filtered, normals, seg\_indices) self.n\_seg\_.append(n\_seg) print(f"Extract {n\_seg} segments from point cloud.") # Construct a new type point cloud with normals and segmentation for i in range(len(pc\_filtered)): pt = PointXYZINS() pt.x = pc\_filtered[i][0] pt.y = pc\_filtered[i][1] pt.z = pc\_filtered[i][2] [pt.int](pt.int#pt.int)ensity = pc\_filtered[i][3] / intensity\_max pt.normal\_x = normals[i].normal\_x pt.normal\_y = normals[i].normal\_y pt.normal\_z = normals[i].normal\_z pt.curvature = normals[i].curvature self.curvature\_max\_ = max(self.curvature\_max\_, pt.curvature) pt.segment = -1 pc.append(pt) # Assign segment labels for i in range(n\_seg): for index in seg\_indices[i].indices: pc[index].segment = i def project\_on\_image(self, vec, extrinsic, margin): # Placeholder for the ProjectOnImage method logic pass def segment\_pc(self, pc\_filtered, normals, seg\_indices): # Placeholder for the point cloud segmentation logic passclass PointXYZINS: def \_\_init\_\_(self): self.x = 0 self.y = 0 self.z = 0 [self.int](self.int#self.int)ensity = 0 self.normal\_x = 0 self.normal\_y = 0 self.normal\_z = 0 self.curvature = 0 self.segment = -1import pclfrom pcl import SACSegmentationFromNormals, NormalEstimation, ModelCoefficientsfrom pcl import PointIndices, EuclideanClusterExtraction, ExtractIndicesclass Calibrator: def segment\_pc(self, cloud, normals, seg\_indices): # Compute normals norm\_est = NormalEstimation() tree = cloud.make\_kdtree() norm\_est.set\_SearchMethod(tree) norm\_est.set\_KSearch(40) # Set K-Search or Radius Search norm\_est.setInputCloud(cloud) norm\_[est.com](est.com#est.com)pute(normals) # Plane segmentation coefficients = ModelCoefficients() indices\_plane = PointIndices() seg = SACSegmentationFromNormals() seg.set\_OptimizeCoefficients(True) seg.set\_ModelType(pcl.SACMODEL\_NORMAL\_PLANE) seg.set\_NormalDistanceWeight(0.2) seg.set\_MethodType(pcl.SAC\_RANSAC) seg.set\_MaxIterations(3000) seg.set\_DistanceThreshold(0.2) seg.setInputCloud(cloud) seg.setInputNormals(normals) seg.segment(indices\_plane, coefficients) # Extract plane points extract = ExtractIndices() extract.setInputCloud(cloud) indices\_notplane = PointIndices() cloud\_out = pcl.PointCloud() plane\_size = len(indices\_plane.indices) indices\_plane\_all = PointIndices() seg\_point\_num = [] while plane\_size > self.params\_['min\_plane\_point\_num']: print(f"Plane points: {plane\_size}") seg\_indices.append(indices\_plane) seg\_point\_num.append(plane\_size) indices\_plane\_all.indices.extend(indices\_plane.indices) extract.setIndices(indices\_plane\_all) extract.filter(cloud\_out) extract.getRemovedIndices(indices\_notplane) seg.setIndices(indices\_notplane) seg.segment(indices\_plane, coefficients) plane\_size = len(indices\_plane.indices) print(f"Plane points < {self.params\_['min\_plane\_point\_num']}, stop extracting plane.") # Euclidean cluster extraction ec = EuclideanClusterExtraction() eu\_cluster\_indices = [] ec.set\_ClusterTolerance(self.params\_['cluster\_tolerance']) ec.set\_MaxClusterSize(10000) ec.set\_MinClusterSize(50) ec.set\_SearchMethod(tree) ec.setInputCloud(cloud) ec.setIndices(indices\_notplane) ec.extract(eu\_cluster\_indices) print(f"Euclidean cluster number: {len(eu\_cluster\_indices)}") seg\_indices.extend(eu\_cluster\_indices) for cluster in eu\_cluster\_indices: seg\_point\_num.append(len(cluster.indices)) self.seg\_point\_num\_.append(seg\_point\_num) return len(seg\_indices)# Placeholder for params and cloud input datacalibrator = Calibrator()import numpy as npimport cv2class Calibrator: def \_\_init\_\_(self, params): self.params = params self.extrinsic = params['extrinsic'] self.max\_score = 0 self.best\_var = np.zeros(6) def CalRatio(self): # Placeholder for ratio calculation (e.g., LiDAR points per pixel) pass def VisualProjectionSegment(self, extrinsic, filename, flag): # Placeholder for visualizing the segmented projection of LiDAR points on image pass def VisualProjection(self, extrinsic, filename, flag): # Placeholder for visualizing projection of LiDAR points on image pass def CalScore(self, extrinsic): # Placeholder for calculating the score (alignment quality) return np.random.rand() # Just for demo, should return a calculated score def PrintCurrentError(self): # Placeholder for printing current error between refined extrinsic and ground truth pass def RandomSearch(self, search\_num, trans\_init, rot\_init, thread\_id): # Simulate a random search for optimization for i in range(search\_num): var = np.random.uniform(-1, 1, 6) \* [rot\_init, rot\_init, rot\_init, trans\_init, trans\_init, trans\_init] score = self.CalScore(self.extrinsic \* self.GetDeltaT(var)) if score > self.max\_score: self.max\_score = score self.best\_var = var def RandomSearchThread(self, search\_num, trans\_init, rot\_init): # For multithreading (optional) self.RandomSearch(search\_num, trans\_init, rot\_init, 0) def GetDeltaT(self, var): # Placeholder for generating a transformation matrix from the variable 'var' T = np.eye(4) # Simulate transformation matrix calculation from var (6D vector for rot & trans) T[:3, :3] = np.eye(3) # Simulate rotation part T[:3, 3] = var[3:6] # Simulate translation part return T def Calibrate(self): # Step 1: Calculate ratio self.CalRatio() print("----------Start calibration----------") # Step 2: Visualize initial projections self.VisualProjectionSegment(self.extrinsic, "init\_proj\_seg.png", 0) self.VisualProjection(self.extrinsic, "init\_proj.png", 0) if self.params['is\_gt\_available']: self.VisualProjectionSegment(self.params['extrinsic\_gt'], "gt\_proj\_seg.png", 0) self.VisualProjection(self.params['extrinsic\_gt'], "gt\_proj.png", 0) # Step 3: Calculate initial score self.max\_score = self.CalScore(self.extrinsic) rot\_init = self.params['search\_range\_rot'] + 0.5 trans\_init = self.params['search\_range\_trans'] + 0.05 # Step 4: Random search for optimization while rot\_init > 0.3: self.best\_var = np.zeros(6) if self.params['num\_thread'] != 0: self.RandomSearchThread(self.params['search\_num'], trans\_init, np.deg2rad(rot\_init)) else: self.RandomSearch(self.params['search\_num'], trans\_init, np.deg2rad(rot\_init), 0) # Update extrinsic matrix self.extrinsic = np.dot(self.extrinsic, self.GetDeltaT(self.best\_var)) # Reduce the search space rot\_init /= 2 trans\_init /= 1.5 print("----------Calibration complete----------") # Step 5: Print final error if ground truth is available if self.params['is\_gt\_available']: self.PrintCurrentError() # Step 6: Visualize refined projections self.VisualProjectionSegment(self.extrinsic, "refined\_proj\_seg.png", 0) self.VisualProjection(self.extrinsic, "refined\_proj.png", 0)# Example usage:params = { 'extrinsic': np.eye(4), # Initial extrinsic matrix 'extrinsic\_gt': np.eye(4), # Ground truth extrinsic 'search\_range\_rot': 5.0, 'search\_range\_trans': 0.1, 'search\_num': 100, 'is\_gt\_available': True, 'num\_thread': 0 # Change to non-zero for multi-threading}calibrator = Calibrator(params)calibrator.Calibrate()