Inspection Planning Primitives with Implicit Models

Jingyang You[1], Hanna Kurniawati[1] and Lashika Medagoda[2]

***Abstract*— The aging and increasing complexity of infrastructures make efficient inspection planning more critical in ensuring safety. Thanks to sampling-based motion planning, many inspection planners are fast. However, they often require huge memory. This is particularly true when the structure under inspection is large and complex, consisting of many struts and pillars of various geometry and sizes. Such structures can be represented efficiently using implicit models, such as neural Signed Distance Functions (SDFs). However, most primitive computations used in sampling-based inspection planner have been designed to work efficiently with explicit environment models, which in turn requires the planner to use explicit environment models or performs frequent transformations between implicit and explicit environment models during planning. This paper proposes a set of primitive computations, called Inspection Planning Primitives with Implicit Models (IPIM), that enable sampling-based inspection planners to entirely use neural SDFs representation during planning. Evaluation on three scenarios, including inspection of a complex real-world structure with over 92M triangular mesh faces, indicates that even a rudimentary sampling-based planner with IPIM can generate inspection trajectories of similar quality to those generated by the state-of-the-art planner, while using up to $70\times$ less memory than the state-of-the-art inspection planner.**

I. INTRODUCTION

Regular and frequent inspection of infrastructures is critical to ensure safety, while efficient inspection is important to minimize disruption. Autonomous robots have great potential to perform such regular, frequent, and fast inspections. However, inspection planning methods that enable robots to perform autonomous inspection still face difficulties to inspect large and complex structures, such as a distillation plant illustrated in Fig. 1, due to large memory requirements. This paper aims to alleviate such a difficulty.

Inspection Planning describes the process of finding a robot's trajectory, such that upon following the trajectory, the robot perceives all points on the surfaces of the structure being inspected without any collision. Among non-myopic inspection planning methods, sampling-based inspection planning algorithms are widely used.

In sampling-based inspection planning (e.g., [1], [2], [3]), the planner samples a representative set of valid configurations, and for each of these configurations, the planner maintains information about parts of the structures perceived by the robot if it were to scan the environment from the particular configuration. This additional information about the perceived observations implies that the memory requirements of the planner increases proportionally with the number of valid sampled configurations and complexity of the environments. Moreover, in general, the number of valid samples required to generate a good inspection trajectory also increases substantially with the size and complexity of the environment being inspected. As a result, inspection planners often have prohibitively large memory requirements. Recent implicit models, such as neural Signed Distance Functions (SDFs) [4], [5], are known for their memory efficiency and could alleviate the mentioned memory requirement problems.

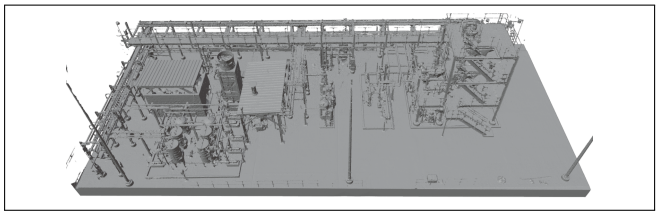

Fig. 1. A glycol distillation plant at San Jacinto College, on which we evaluate the performance of **IPIM**. We name it **Plant** in Section V-A.

However, existing sampling-based inspection planners cannot fully benefit from such an implicit representation because existing primitives for sampling-based inspection planning were designed for explicit environment models. These primitives are: **(1) Collision Check** to ensure collision-free inspection trajectory, **(2) Observation Simulation** to predict observations (e.g., images, point clouds) perceived from a sampled configuration, **(3) Observation Representation** to store the simulated observation during planning and **(4) Total Coverage Check** to calculate the coverage along a potential inspection trajectory. Many efficient methods for each of these primitives have been proposed, but they require the environment to be represented as an explicit model.

In this paper, we propose novel primitive computations, called Inspection Planning Primitives with Implicit Models (**IPIM**), to enable samping-based inspection planning to leverage the more compact implicit models during the entire planning process. Specifically, with **IPIM**, the most memory-consuming components of inspection planning, i.e., explicit global environment models and local observation representations, can be replaced by memory-efficient neural SDFs. Although SDFs are not new, primitives that enable sampling-based inspection planners to directly use neural SDF representation are novel and non-trivial.

We evaluated **IPIM** with a simple sampling-based inspection planner on three scenarios, including the inspection of a real-world distillation plant dataset with over 92M mesh shown in Fig. 1. The results indicate that **IPIM** can substantially reduce the memory requirement by up to $70\times$, compared to the state-of-the-art method.

[1] School of Computing, The Australian National University, Canberra ACT 2600, Australia {jingyang.you ; hanna.kurniawati}@anu.edu.au

[2] Abyss Solutions, 11-21 Underwood Rd, Homebush NSW 2140, Australia lashika@abysssolutions.com.au

II. Related Work

A. Sampling-based Inspection Planning

Inspection planning methods can be classified into myopic and non-myopic. Myopic methods generally rely on submodular characteristics of inspection planning problems. However, this characteristics is often false for complex cluttered environments where many areas are occluded, requiring elaborate inspection strategies that cannot be generated by myopic approaches.

The most scalable non-myopic approach is sampling-based inspection planning. These methods compute globally feasible / optimal inspection paths via sampling. For instance, Random Inspection Tree Algorithm (RITA) [1] is one of the first sampling-based methods that computes asymptotically optimal solution to inspection planning problems with non-holonomic motion constraints. To improve RITA's efficiency, Rapidly exploring Random Tree of Trees (RRTOT) [2] proposes to utilize inter-branch knowledge such that promising samples among different branches can be shared. The state-of-the-art sampling-based method today is Incremental Random Inspection-roadmap Search (IRIS) [3], [6], which first constructs a rapidly-exploring random graph (RRG) [7], then searches on the RRG for the optimal inspection path. Note, however, both RRTOT and IRIS require steering functions, which can be costly to compute, when the inspection is performed by non-holonomic robots.

All sampling-based inspection planners share the four primitives of inspection planning, as mentioned in Section I. Since they use explicit global environment models, they require huge memory when the structure to be inspected is large and complex, hindering their applicability in such inspection scenarios.

B. Neural Implicit Representation

Neural implicit representations encode 3D spatial information [5] compactly in the weights of a deep neural network. *Radiance Field* [8], [9], [10] and *Signed Distance Function* (SDF) [4], [5], [11] are the most widely used neural implicit representations. We focus on SDF for its high training and inference efficiency.

SDF is a continuous function that maps any given spatial point to a signed distance between the point and the boundary of an object, where the sign encodes whether the point is inside (negative) or outside (positive) of the object, and an SDF value of 0 implies the point is a surface point —that is, the point lies on the boundary of the object [4]. Finding the direct $\mathbb{R}^3 \to \mathbb{R}$ SDF mapping is challenging as deep neural nets are biased towards learning low-frequency functions [12], which is not the case in real-world 3D environments where high-frequency variations in structures are common. Various methods [13], [14], [15] have been proposed to efficiently encode raw coordinate inputs. These encodings enable training the SDF mapping to become nearly real-time, enabling their use in real-time robotics applications like visual SLAM [5], [11], [16], navigation [17], [18] and motion planning [19].

Despite its wide applicability in robotics, SDF is rarely used in inspection planning even though it can significantly reduce memory requirements because, inspection planning primitives have been designed for explicit environment models. Removing this difficulty, we propose **IPIM**, a set of primitives for inspection planning with neural SDFs.

III. Problem Definition

We follow a typical inspection planning problem formulation, e.g., [1]. Let $C_{free}, U, \mathcal{E}$ be the collision-free part of the robot's configuration space, the control space and the explicit model of the environment, respectively. Denote the set of surface points to be inspected in $\mathcal{E}$ as $S_{\mathcal{E}}$. Our goal is to find a collision-free trajectory $\gamma^* : \{0, 1, \cdots, T\} \to C_{free}$ induced by the time-parameterized control function $\tau : \{0, 1, \cdots, T\} \to U$, such that upon following the trajectory, the robot starting from $\gamma(0)$ can perceive each point in S from at least one configuration $\gamma(t), t \in \{0, 1, \cdots, T\}$, and the workspace length of the trajectory is minimized. We assume the robot's geometry and kinematics are known a priori. It is equipped with visibility sensors with depth information, such as depth cameras or LiDARs. The robot performs discrete sensing, taken only at the end of each control command $\tau(t), t \in [0, T]$.

IV. Implicit Sampling-based Inspection Planning

A. Overview

IPIM proposes efficient primitives for inspection planning when implicit neural SDF representation is used to represent the environment and observation. Neural SDF is recognized for its rapid training and compactness, making it well-suited for planning tasks that demand memory efficiency.

With **IPIM**, the known explicit environment model $\mathcal{E}$ is first converted into an implicit SDF model $f_{\mathcal{E}}$ represented by a neural network (Section IV-B). Without loss of generality, we suppose **IPIM** is used with a sampling-based inspection planner that builds a tree $\mathbb{T} = \{\mathbb{N}, \mathbb{E}\}$, where $\mathbb{N}$ and $\mathbb{E}$ are the set of nodes and edges in the tree. Under **IPIM**, each node $\mathbf{n} \in \mathbb{N}$ represents a 4-tuple $\langle \mathbf{q}, \mathbf{B}_{\mathbf{P_n}}, f_{\mathbf{P_n}}, \mathbf{cov(n)} \rangle$. The four elements in the tuple reflect how **IPIM** converts the four primitives of the planner to their implicit forms, namely:

1) **Collision Check:** Let $\mathbf{q} \in C_{free}$ be the sampled robot's configuration at node $\mathbf{n}$. Suppose the parent of $\mathbf{n}$ is $\mathbf{n}' \in \mathbb{N}$ and the configuration at $\mathbf{n}'$ is $\mathbf{q}' \in C_{free}$, then the collision check against trajectory $\overline{\mathbf{q}'\mathbf{q}}$ is performed with the implicit model $f_{\mathcal{E}}$.
2) **Observation Simulation:** Let $\mathbf{P_n}$ be the raw observation perceived by the robot's configuration at node $\mathbf{n}$ (e.g., a depth image). This observation is simulated using the SDF function $f_{\mathcal{E}}$. We also derive a point cloud from $\mathbf{P_n}$. *For brevity, we use the term bounding box of $\mathbf{P_n}$ to refer to the bounding box of the point cloud corresponding to the observation $\mathbf{P_n}$.* **IPIM** maintains the bounding box of $\mathbf{P_n}$, denoted as $\mathbf{B}_{\mathbf{P_n}}$. *In the following sections, we use $\mathbf{P}$ without the subscript $\mathbf{n}$ when referring to a general raw observation.*

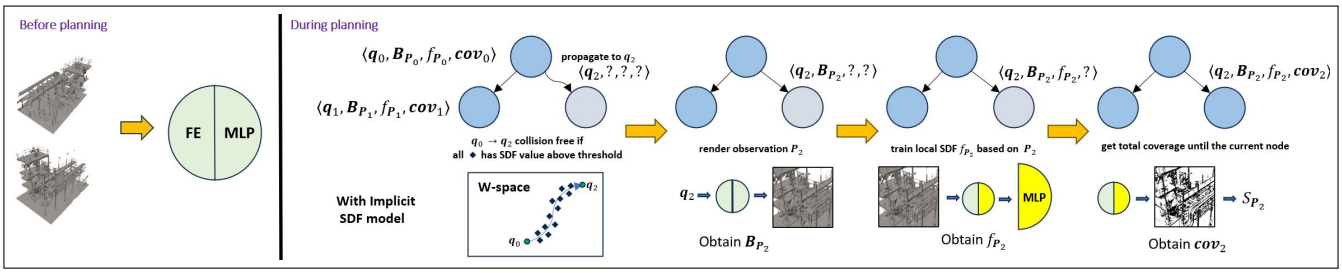

Fig. 2. Illustration of how **IPIM** is used with an inspection planner to reduce the memory cost. Before planning, the explicit environment $\mathcal{E}$ is converted to neural SDF $f_{\mathcal{E}}$ comprising a feature extractor (FE) and a Multi-Layer Perceptron (MLP). During planning, **IPIM** converts the four primitive computations of inspection planning with SDF to the implicit counterparts. Description about the four primitive computations is presented in Section I.

Algorithm 1 TP-IPIM (Explicit Env. $\mathcal{E}$, Planner **TP**)

1: Initialize: **TP**.Tree $\mathbb{T}$
2: Initialize: Implicit SDF model $f_{\mathcal{E}}$ ▷ Section IV-B
3: **while** PlanningTime=True **do**
4: $\mathbf{n}_{prt}, \mathbf{n}_{chd}, \overline{\mathbf{n}_{prt}\mathbf{n}_{chd}}$ = **Expand**($\mathbb{T}$)
5: **if CollisionFree**($\overline{\mathbf{n}_{prt}\mathbf{n}_{chd}}$, $f_{\mathcal{E}}$) **then** ▷ Section IV-C
6: $\mathbf{P}$ = **SimulateObs**($\mathbf{n}_{chd}$, $f_{\mathcal{E}}$) ▷ Section IV-D
7: $\mathbf{n}_{chd}.\mathbf{B}_{\mathbf{P}} = \mathbf{P}.\mathbf{B}_{\mathbf{P}}$
8: $f_{\mathbf{P}}$ = **ObsRepresent**($\mathbf{P}$, $f_{\mathcal{E}}$) ▷ Section IV-E
9: $S_{\mathbf{P}}$ = **MarchingCube**($f_{\mathbf{P}}$)
10: $\mathbf{n}_{chd}.f_{\mathbf{P}} = f_{\mathbf{P}}$
11: $\mathbf{n}_{chd}$.**cov** = **TotalCov**($\mathbf{n}_{chd}, S_{\mathbf{P}}$) ▷ Section IV-F
12: $\mathbb{T}$.**Add**($\overline{\mathbf{n}_{prt}\mathbf{n}_{chd}}, \mathbf{n}_{chd}$)

3) **Observation Representation:** The notation $f_{\mathbf{P_n}}$ denotes the tiny-size multi-layer perceptron that encodes the local SDF representation of the observation $\mathbf{P_n}$. The set of local surface points, denoted as $S_{\mathbf{P_n}}$, can then be *generated* from $\mathbf{B}_{\mathbf{P_n}}$ and $f_{\mathbf{P_n}}$ with marching cube [20].
4) **Total Coverage Check:** The notation **cov(n)** refers to the accumulated coverage in the path from the root of $\mathbb{T}$ until node $\mathbf{n}$ of $\mathbb{T}$. This coverage is calculated incrementally and implicitly as $\mathbb{T}$ being expanded.

The details of each of the primitives above are presented in Section IV-C–Section IV-F, respectively.

Algorithm 1 presents an overview of how a sampling-based inspection planner with tree representation can use **IPIM** and neural SDF model throughout its entire planning process. Fig. 2 provides a summary of the four primitives in **IPIM** and how they are applied in a tree-based sampling-based inspection planner.

B. SDF Neural Network Structure

IPIM replaces the memory-consuming explicit environment model $\mathcal{E}$ with an implicit neural SDF, denoted as $f_{\mathcal{E}}$, that approximates the ground truth SDF. To this end, it parameterizes the neural SDF function with additional parameters that are the weights of the neural network. Specifically:

$$f_{\mathcal{E}}(\mathbf{x}; \boldsymbol{\theta}_1, \boldsymbol{\theta}_2) = \text{MLP}\left(\text{FE}(\mathbf{x}; \boldsymbol{\theta}_1); \boldsymbol{\theta}_2\right) \tag{1}$$

where $\mathbf{x} \in \mathbb{R}^3$ is a 3D point, and $\boldsymbol{\theta}_1 \in \mathbb{R}^{d_1}, \boldsymbol{\theta}_2 \in \mathbb{R}^{d_2}$ are the weights of the neural network, where d_1, d_2 are hyperparameters determining the size of the weight. MLP stands for **M**ulti-**L**ayer **P**erceptron. Whilst the notation $\text{FE}(\mathbf{x}; \boldsymbol{\theta}_1) = \left[\text{HashGrid}(\mathbf{x}; \boldsymbol{\theta}_1)^T \ \text{OneBlob}(\mathbf{x})^T\right]^T$ refers to a **F**eature **E**xtractor that concatenates features obtained from HashGrid – multiresolution hash encoding [15], and OneBlob encoders [14]. The former learns spatial features from $\mathbf{x}$ and the latter describes $\mathbf{x}$ with multiple frequency bands, suitable for structures with complex geometry.

HashGrid disentangles the task of mapping into two subtasks: *(1) feature extraction* and *(2) mapping from extracted features to SDF values*. Task (1) requires many parameters as it contains high-level spatial information of the environment, while task (2) usually requires only MLPs of tiny sizes. Since $\mathcal{E}$ is a priori known, we can pre-train FE ($\boldsymbol{\theta}_1$) to complete task (1). During planning, all task (1) parameters are *shared* among all nodes in the tree, while the local observations in each node is fully controlled by tiny-sized parameters in downstream MLPs and parameterized by $\boldsymbol{\theta}_2$.

When training the network on the entire environment $\mathcal{E}$, we apply the mean squared loss function

$$\mathcal{L}_{\mathcal{E}}(\mathbf{X}, \mathbf{Y}; \boldsymbol{\theta}_1, \boldsymbol{\theta}_2) = \frac{1}{|\mathbf{X}|} \left\| f_{\mathcal{E}}(\mathbf{X}; \boldsymbol{\theta}_1, \boldsymbol{\theta}_2) - \mathbf{Y} \right\|_2^2 \tag{2}$$

where $\mathbf{X}, \mathbf{Y}$ are collections of 3D coordinates and corresponding SDF values obtained from the explicit representation of $\mathcal{E}$, respectively. $|\cdot|$ and $\|\cdot\|_2$ are the cardinality and the l_2 norm. We apply the truncated SDF (TSDF) [4] strategy, which ensures the magnitude of SDF value cannot be greater than a given threshold, to better approximate the depth values and reduce variance in points far from any surface. TSDF is applied to the ground truth SDF and to the learnt SDFs.

C. Collision Check with Implicit SDF

To check whether the robot at configuration $\mathbf{q}$ is in collision or not, we first represent the robot's geometry with a sufficiently dense set of control points [21] that encapsulates the robot, denoted by $\mathcal{C}(\mathbf{q})$. For any $\mathbf{p} \in \mathcal{C}(\mathbf{q})$ and threshold value $\xi > 0$, if $f_{\mathcal{E}}(\mathbf{p}) > \xi$, we know $\mathbf{p}$ is ξ units away from its closest surface. When all points in $\mathcal{C}(\mathbf{q})$ have SDF values greater than ξ, we can infer the robot at configuration $\mathbf{q}$ is collision free.

D. Generating Simulated Observation from SDF

IPIM provides a function to simulate observations from neural SDF environment model. Specifically, **IPIM** simulates observation depth image $\mathbf{P}$ from the neural SDF $f_{\mathcal{E}}$, instead

of the explicit environment $\mathcal{E}$. To this end, **IPIM** marches rays emitted from a simulated depth sensor. For each ray, **IPIM** finds the first surface point of $\mathcal{E}$ hit by the ray, i.e., the first point $\mathbf{p}$ along the ray that satisfies $f_{\mathcal{E}}(\mathbf{p}) = 0$. The simulated depth provided by the ray is then the distance the ray travels to reach $\mathbf{p}$.

E. Representing Simulated Observation

Instead of storing the simulated observation explicitly, **IPIM** represents them as local SDF, denoted as $f_{\mathbf{P}}$ and encoded as a tiny-sized local MLP that reuses the feature extractor FE of $f_{\mathcal{E}}$.

A key functionality of the simulated observation is to estimate the coverage of different paths. Suppose $\mathbf{P_n}$ is the depth image perceived from a robot's configuration at node $\mathbf{n}$ of the plannng tree $\mathbb{T}$. We define coverage of $\mathbf{n}$ as $\#\text{Covered}(\mathbf{P_n})/|S_{\mathcal{E}}|$, where $\#\text{Covered}(\mathbf{P_n})$ and $|S_{\mathcal{E}}|$ are the number of surface points of $\mathcal{E}$ visible in $\mathbf{P_n}$ and the total surface points, respectively. The set of surface points of $\mathcal{E}$ visible in $\mathbf{P_n}$ can be obtained by performing marching cube [20] on the SDF model $f_{\mathbf{P_n}}$. Modern libraries like *Pytorch3D* [22] support GPU-accelerated marching cube. These libraries enable marching cube operation to run fast, especially when $\mathcal{E}$ has a high resolution. Moreover, by using marching cube, the explicit $\mathbf{P}$ or equivalently the corresponding point cloud, can be discarded, and only the weights of $f_{\mathbf{P_n}}$, which is tiny, need to be maintained, thereby reducing memory usage. Since coverage is computed for each node of $\mathbb{T}$, the total memory **IPIM** saves is substantial.

Now the question is how to efficiently learn an accurate $f_{\mathbf{P}}$ from $\mathbf{P}$, so that the implicit $f_{\mathbf{P}}$ can be used faithfully just like the explicit $\mathbf{P}$. Such an accurate $f_{\mathbf{P}}$ also minimizes labeling unvisited areas as visited or vice versa, helping to compute an accurate estimate of paths' coverage. Training $f_{\mathbf{P}}$ from scratch generally requires more memory or time than storing $\mathbf{P}$ explicitly. However, since $\mathcal{E}$ is static and known a priori, any $\mathbf{P}$ must correspond to a certain part of $\mathcal{E}$. Since FE of $f_{\mathcal{E}}$ contains the spatial information of the entire $\mathcal{E}$, it also contains the area corresponding to any $\mathbf{P}$. Therefore, FE of $f_{\mathcal{E}}$ can be reused when encoding $\mathbf{P}$. The local SDF of $\mathbf{P}$ is hence $f_{\mathbf{P}} = \text{MLP}\left(\text{FE}(\mathbf{x}; \boldsymbol{\theta}_1); \boldsymbol{\theta}_{\mathbf{P}}\right)$, where the FE of $f_{\mathcal{E}}$ is reused, and the parameters of $\boldsymbol{\theta}_{\mathbf{P}}$ are re-trained and maintained.

Before defining the loss function used to train $f_{\mathbf{P}}$, let us first define our accuracy goals: (1) surface points visible from $\mathbf{P}$ should have SDF values close to 0, and (2) invisible (occluded / uncovered) surface points should have large TSDF magnitudes so that their corresponding vertices disappear in the mesh obtained from $f_{\mathbf{P}}$. The first objective can be solved by minimizing $\mathcal{L}_{\mathcal{E}}$, while (2) cannot, as the shared FE has dominantly more parameters than that of MLP. If only $\mathcal{L}_{\mathcal{E}}$ is minimized, FE *predicts* accurate SDF values on areas not visible in $\mathbf{P}$, which is undesirable. Instead, we want the SDF of areas not visible in local observation $\mathbf{P}$ to be large in magnitude, so that the marching cube *does not* generate surface points in areas not visible in $\mathbf{P}$.

Algorithm 2 ObsRepresent (Observation $\mathbf{P}$, TSDF $f_{\mathcal{E}}$)

Initialize: New multi-layer perceptron MLP
Initialize: TSDF model $f_{\mathbf{P}} = \text{MLP} \circ f_{\mathcal{E}}.\text{FE}$
Initialize: Freeze $\boldsymbol{\theta}_1$, weight of $f_{\mathcal{E}}.\text{FE}$
Initialize: Reset $\boldsymbol{\theta}_{\mathbf{P}}$, weight of MLP
Initialize: Ray origin, directions $\mathbf{O}$, $\mathbf{D}$ from $\mathbf{P}$
Initialize: Target depth $\mathbf{T}$ from $\mathbf{P}$, truncation $tr = f_{\mathcal{E}}.tr$
Sample $\mathbf{T}_{vis}$ in $[\mathbf{T} - tr, \mathbf{T}]$
Sample $\mathbf{T}_{occ}$ in $[\mathbf{T}, \mathbf{P}.\text{MaxDepth}]$
$\{\mathbf{X}_{vis}, \mathbf{X}_{occ}\} = \mathbf{O} + \{\mathbf{T}_{vis}, \mathbf{T}_{occ}\} \odot \mathbf{D}$
$\mathbf{Y}_{vis} = \mathbf{T} - \mathbf{T}_{vis}$; $\mathbf{Y}_{occ} = \min\{\mathbf{T} - \mathbf{T}_{occ}, -tr\}$
while Iter $<$ MaxIter **do**
    $\boldsymbol{\theta}_{\mathbf{P}} \leftarrow \boldsymbol{\theta}_{\mathbf{P}} - \alpha \nabla_{\boldsymbol{\theta}_{\mathbf{P}}} \mathcal{L}_{local}$
Return $f_{\mathbf{P}}$

Algorithm 3 TotalCov (Node N, Surface points $S_{\mathbf{P}_N}$)

Initialize: Predecessor Nodes $1, 2, \cdots, N$
Initialize: Bounding boxes $\{\mathbf{B}_{\mathbf{P}_i}\}_{i=1}^{N}$, Tolerance $\epsilon > 0$
for i **in** $\{1, 2, \cdots, N-1\}$ **do**
    if $\mathbf{B}_{\mathbf{P}_i} \cap \mathbf{B}_{\mathbf{P}_N} \neq \varnothing$ **then**
        for $\mathbf{p} \in S_{\mathbf{P}_N}$ **do**
            if $\mathbf{p} \in \mathbf{B}_{\mathbf{P}_i}$ **and** $|f_{\mathbf{P}_j}(\mathbf{p})| < \epsilon$ **then**
                Remove $\mathbf{p}$ from $S_{\mathbf{P}_N}$
Return $|S_{\mathbf{P}_N}|/|S_{\mathcal{E}}|$ + $(N-1)$.**cov**

Let us define a ray as $\mathbf{r} = \mathbf{o} + t \cdot \mathbf{d}$, where $\mathbf{o}, \mathbf{d}$ are the ray origin and direction, respectively, and t is the distance traveled along $\mathbf{d}$. From the depth information in $\mathbf{P}$, we know when the ray hits a surface, say at $t = t_0$, all subsequent points on the ray, i.e., at $t = t_0 + \delta$ where $\delta > 0$, are occluded. For learning the neural SDF, the occluded points have target TSDF values $\min\{tr, \delta\}$ where $tr > 0$ is the already set truncation value for TSDF. These occluded points will also be fed into the network to weaken the dominance of FE, letting MLP predict truncation values for occlusions. The loss function for learning local SDF is then:

$$\begin{aligned}\mathcal{L}_{local}(\mathbf{X}_{vis}, \mathbf{X}_{occ}, \mathbf{Y}_{vis}, \mathbf{Y}_{occ}; \boldsymbol{\theta}_1, \boldsymbol{\theta}_{\mathbf{P}}) \\ = \lambda_{vis} \cdot \mathcal{L}_{\mathcal{E}}(\mathbf{X}_{vis}, \mathbf{Y}_{vis}; \boldsymbol{\theta}_1, \boldsymbol{\theta}_{\mathbf{P}}) \\ + \lambda_{occ} \cdot \frac{1}{|\mathbf{X}_{occ}|} \left\| f_{\mathcal{E}}(\mathbf{X}_{occ}; \boldsymbol{\theta}_1, \boldsymbol{\theta}_{\mathbf{P}}) - \mathbf{Y}_{occ} \right\|_2^2 \quad (3)\end{aligned}$$

where λs are hyperparameters, vis, occ stand for visible and occluded. Note the weight of FE, $\boldsymbol{\theta}_1$, is frozen during learning MLP weights $\boldsymbol{\theta}_{\mathbf{P}}$. The pseudocode for learning $f_{\mathbf{P}}$ is shown in Algorithm 2. This strategy enables inspection planners to be much more efficient in representing locally observed environments.

F. Total Coverage Checking

Each path from the root to a leaf node in the planning tree $\mathbb{T}$ forms an inspection trajectory. The total coverage along trajectories can be used as termination criterion and planning heuristic. However, unlike explicit representations where the total coverage can be easily retrieved from set unions, trivially concatenating SDFs in nodes along trajectories do not result in another SDF that describes the total coverage.

To perform **Total Coverage Check** along a trajectory, **IPIM** requires the set of local surface points of the nodes in the trajectory. The local surface point $S_{\mathbf{P_n}}$ of node $\mathbf{n}$ can be obtained by performing marching cube on the learnt local SDF $f_{\mathbf{P_n}}$, assuming the bounding box of the observation $\mathbf{B}_{\mathbf{P_n}}$ is defined. To ensure $S_{\mathbf{P_n}}$ is a subset of global surface points, so that no extra surface points are generated, the bounding box $\mathbf{B}_{\mathbf{P_n}}$ must be snapped to the grid used by the marching cube on $f_{\mathcal{E}}$. In practice, one only needs to snap the two extreme points of $\mathbf{B}_{\mathbf{P_n}}$ to the grid. Note that using the above method, the set of local surface points of a node is only needed the first time coverage is computed for the particular node, and can be discarded immediately after, to keep the memory consumption low.

Suppose a trajectory has $N-1$ nodes with local SDFs $\{f_{\mathbf{P}_i}\}_{i=1}^{N-1}$ and total coverage C. When a new node N is added to the trajectory, we first identify the number K of newly visited surface points at the new node N — that is, surface points that do not intersect with observations in the predecessor nodes $\{1, 2, \cdots, N-1\}$. Then, the total coverage for the new trajectory, consisting of $\{\text{node-}1, \cdots, \text{node-}(N-1), \text{node-}N\}$, can be calculated incrementally as $C + K/|S_{\mathcal{E}}|$.

To compute the above coverage, we need an efficient method to check if a surface point in the newly added node, $\mathbf{p} \in S_{\mathbf{P}_N}$, intersects with an observation in the predecessor nodes $\{\mathbf{P}_i\}_{i=1}^{N-1}$. To this end, first, $\mathbf{B}_{\mathbf{P}_N}$ is checked against $\{\mathbf{B}_{\mathbf{P}_i}\}_{i=1}^{N-1}$. If $\mathbf{B}_{\mathbf{P}_N}$ and $\mathbf{B}_{\mathbf{P}_i}$ do not intersect, then $S_{\mathbf{P}_N} \cap S_{\mathbf{P}_i} = \varnothing$ and hence $\mathbf{p} \notin S_{\mathbf{P}_i}$. If the bounding box $\mathbf{B}_{\mathbf{P}_N}$ and $\mathbf{B}_{\mathbf{P}_i}$ intersects, we need to check if $\mathbf{p} \in S_{\mathbf{P}_i}$. This check can be done efficiently by first checking if $\mathbf{p}$ is in $\mathbf{B}_{\mathbf{P}_i}$. If it is, then $\mathbf{p} \in \mathbf{S}_{\mathbf{P}_i}$ if $|f_{\mathbf{P}_i}(\mathbf{p})| < \epsilon$ where $\epsilon > 0$ is a threshold to check if the SDF is close to 0. The number K of newly visited points is then the cardinality of the set $\{\mathbf{p} \in S_{\mathbf{P}_N} |\ \forall i \in [1, N-1], \mathbf{p} \notin \mathbf{S}_{\mathbf{P}_i}\}$, and the total coverage for the trajectory consisting of $\{\text{node-}1, \cdots, \text{node-}(N-1), \text{node-}N\}$ is $C + K/|S_{\mathcal{E}}|$. Algorithm 3 describes the method to fuse local SDFs to calculate the total coverage.

By converting all primitives of inspection planning to work directly with implicit environment models, **IPIM** enables the development of novel memory-efficient inspection planning.

V. Experiments

A. Experiment Settings

We evaluate our proposed method on three scenarios with screenshots of the mesh shown in Fig. 1 and Fig. 3:

1) **Bridge** has 7.4K mesh vertices and 9.4K mesh faces, taken from [3], [6]. We have reduced the scale of this environment by a factor of ten, so it has size 6m × 2m × 4m.
2) **Plant** refers to a glycol distillation plant at the San Jacinto College. The mesh model of the plant is obtained by processing raw scans collected from a Leica RTC360 Laser Scanner and a BLK ARC Lidar scanner. This scenario consists of 52.5M vertices and 92M faces, and is sized 44m × 19m × 15m.

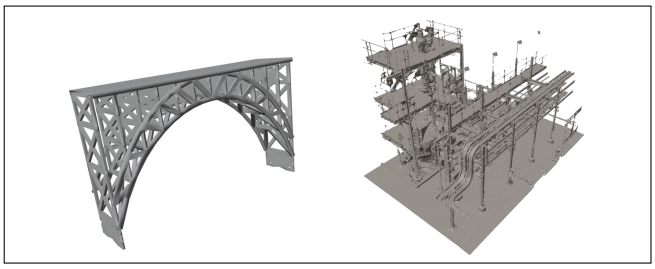

Fig. 3. From left to right: **Bridge** taken from [3], and **Plant-s** taken as a subset of **Plant** shown in Fig. 1.

3) **Plant-s** is a subset of **Plant** with size 15m × 10m × 10m, consisting of 10.4M vertices and 20.9M faces.

The robot used for inspection is a drone with 10cm side length. Its C-space has 5 dimensions, consisting of the position of the drone's center of mass, yaw, and pitch angle. The drone is equipped with a visibility sensor with limited range (1m for the **Bridge** scenario and 3m for the other scenarios) and 90° horizontal and vertical field-of-view.

IPIM works as a framework to be combined with an inspection planner. To demonstrate its performance, we propose a simple tree-based planner **TP** to be used with **IPIM**. **TP** itself is implemented on the CPU, while **IPIM** is accelerated by the GPU. The pseudocode of **TP** with **IPIM** is shown in Algorithm 1 with modification in the tree expansion, and additional prunings. When expanding the planning tree, inspired by [23], we bias **TP** towards nodes with higher coverage, so that $Pr(\mathbf{n} \text{ being sampled}) \propto 1/N_d(\mathbf{n}) + \alpha \mathbf{n}.\mathbf{cov}$ with constant $\alpha > 0$, $N_d(\mathbf{n})$ being the number of nodes whose configuration to $\mathbf{n}$ is less than d units. The expansion is performed single-threaded, such that only one new node is expanded every time. **TP** also prunes the tree every certain iterations (2000 in all our experiments). The pruning maintains only the branch with the highest coverage. To emphasise, the main reason of performing pruning is for **TP** to achieve a higher coverage. Without pruning, **TP** can be viewed as a RITA [1] inspection planner that fails to achieve a decent coverage in large complex scenarios, as suggested by [23] and tested by the authors. Note that **TP** is a simple example planner used to test the performance of **IPIM** and the pruning step will certainly not harm the per-node memory reduction of **IPIM** as **TP** and **IPIM** are completely separate and independent.

We compare the performance of four methods: **TP with IPIM (TP-IPIM)**, **TP without IPIM (TP)**, **IRIS [3]** — a state-of-the-art sampling-based inspection planner, and **IRIS-M**. **IRIS-M** is our modification **IRIS** where the visibility set computation is parallelized by *Open3D* [24] with 12 CPU threads. Without this modification, **IRIS** failed in **Plant-s** and **Plant**. We use the same library and the same number of CPU threads for **TP**. Note that although **TP** and **TP-IPIM** are tree-based while **IRIS** and **IRIS-M** are graph-based, they are still comparable as they all share the same four primitive computations of inspection planning.

TP-IPIM requires an implicit model of the environment, $f_{\mathcal{E}}$. To train $f_{\mathcal{E}}$, we uniformly sample points in both near-surface and far-surface areas. Afterwards, TSDF values of

TABLE I

COVERAGE, COST, AND MEMORY W.R.T. ENVIRONMENTS AND ALGORITHMS

Environment	**Bridge (30 minutes)**				**Plant-s (2 hours)**				**Plant (2 hours)**			
Planner	**IRIS**	**IRIS-M**	**TP**	**TP-IPIM**	**IRIS**	**IRIS-M**	**TP**	**TP-IPIM**	**IRIS**	**IRIS-M**	**TP**	**TP-IPIM**
Mem Base (MB)	0.8 ± 0.0	0.8 ± 0.0	$\mathbf{0.5 \pm 0.0}$	3.6 ± 0.0	N.A.	795 ± 10	930 ± 17	$\mathbf{32 \pm 0}$	N.A.	2178 ± 15	2179 ± 26	$\mathbf{31 \pm 0}$
Mem Total Limit	Each of **IRIS**, **IRIS-M**, and **TP**: **96 GB**; **TP-IPIM**: **1 GB**											
# Out of Mem	0/10	0/10	0/10	0/10	N.A.	6/10	**0/10**	**0/10**	N.A.	10/10	**0/10**	**0/10**
Coverage (%)	55 ± 1	55 ± 1	$\mathbf{88 \pm 0}$	54 ± 3	N.A.	56 ± 3	$\mathbf{84 \pm 1}$	54 ± 3	N.A.	21 ± 2	19 ± 2	$\mathbf{24 \pm 2}$
Cost	$\mathbf{348 \pm 6}$	$\mathbf{348 \pm 6}$	1965 ± 65	629 ± 29	N.A.	$\mathbf{159 \pm 2}$	361 ± 17	221 ± 13	N.A.	286 ± 14	$\mathbf{188 \pm 13}$	328 ± 20

those points can be evaluated from the explicit environment $\mathcal{E}$. The loss function used to encode $\mathcal{E}$, $\mathcal{L}_{\mathcal{E}}$, can then be optimized and $f_{\mathcal{E}}$ is obtained. **IRIS** and **IRIS-M** require manually setting the number of nodes in their RRGs where they search for the optimal inspection path. To set this parameter, we performed preliminary runs to find a suitable parameter for each scenario, which turned out to be 1000 nodes. To set other hyperparameters in **IRIS** and **IRIS-M**, we follow [3], [6].

We use 2.8GHz CPUs and a single RTX3090 GPU for all experiments. **TP-IPIM** and **TP** are implemented using Python for easier deep learning implementation using PyTorch [25]. We use Pytorch3D [22] to accelerate the marching cube algorithm of **TP-IPIM** with GPU. **IRIS** and **IRIS-M** are implemented in C++, following the official implementation. **IRIS** uses spherical sectors as their visibility sets. We change them to pyramids to align with the common assumption, so that all four methods to compare share the same geometry of visibility set. The code of both **IRIS** and **IRIS-M** are taken and modified from the official implementation of [3], [6]. C++ is more memory efficient than Python, giving **IRIS** and **IRIS-M** the upper hand.

Details related to training the environment model $f_{\mathcal{E}}$ and local SDFs $f_{\mathbf{P}}$ are as follows. $f_{\mathcal{E}}$ takes $\sim$ 5 minutes to train with a total of 426K parameters for **Bridge** and 6.9M parameters for **Plant-s** and **Plant**. During planning, $f_{\mathbf{P}}$ are 3-layer MLPs with 505 parameters for **Bridge** and 761 parameters for **Plant-s** and **Plant**. Each $f_{\mathbf{P}}$ is trained for 20 iterations in **Bridge** and 100 iterations in **Plant-s** and **Plant**, with learning rate 3e-3, using AdamW [26] optimiser. **TP-IPIM** plans approximately 8-9 nodes per second in the most complex **Plant** scenario, due to the small size of $f_{\mathbf{P}}$.

B. Performance Comparison

In this section, we compare the performance among **TP-IPIM**, **TP**, **IRIS** and **IRIS-M**, in terms of coverage, cost, and baseline memory and total memory consumption. Coverage is as defined in Section IV-E. The cost is defined as the length of the inspection trajectory, and the baseline memory is counted as the size of the explicit (**IRIS**, **IRIS-M**, **TP**) / implicit (**TP-IPIM**) global environment model, plus the size of all local observation representations in planning nodes. As discussed in Section I, these two components are necessary to most of the sampling-based inspection planning algorithms, and contribute most to the total memory consumption. We emphasize that even if we used GPU in **TP-IPIM**, the storage of planning nodes (e.g., MLP weights $f_{\mathbf{P}}$) is moved to CPU immediately after the GPU calculation is complete.

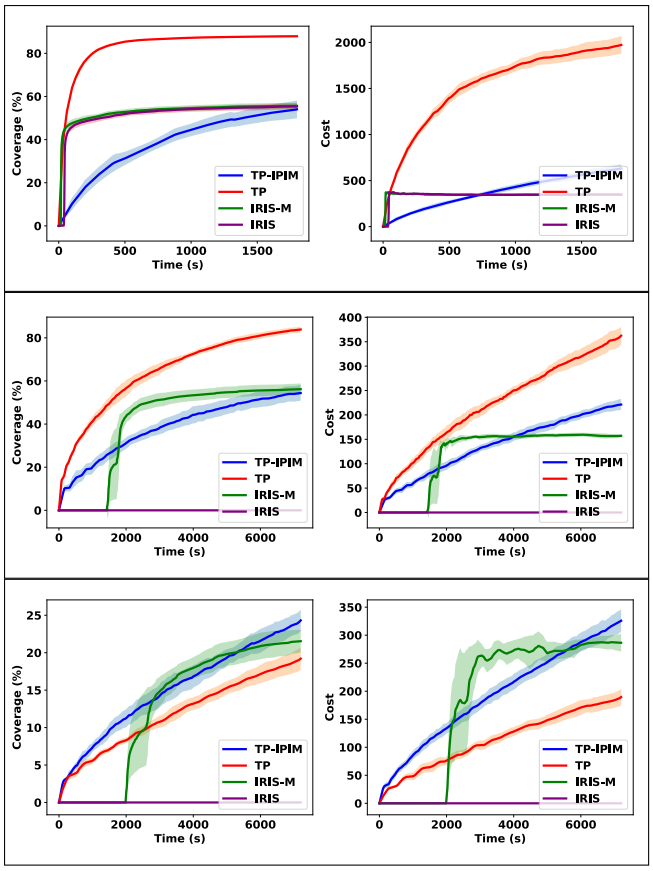

Fig. 4. From left to right: planning time v.s. inspection coverage, and planning time v.s. cost of the inspection path. From up to bottom: scenario **Bridge**, scenario **Plant-s** and scenario **Plant**.

All planners are run for 10 times in three scenarios. We set the planning time limit (which doesn't include the time of obtaining $f_{\mathcal{E}}$ and I/O) to be 30 minutes, 2 hours and 2 hours for **Bridge**, **Plant-s** and **Plant**, respectively. We set the total memory limit to be 96 GB for **IRIS**, **IRIS-M**, **TP** and only 1 GB for **TP-IPIM**, and record the number of out-of-memory runs. Detailed results are shown in Table. I and Figure. 4.

1) Comparison between TP and TP-IPIM: In **Bridge** and **Plant-s** scenarios, **TP** plans faster than **TP-IPIM**, as **TP-IPIM** trains local MLP online to represent the local observation. Though in **Plant**, **TP-IPIM** is faster than **TP**. With a complex environment, the explicit representation of local observations is slow even with parallelized computations using modern libraries. In **Plant**, which is the largest scenario, with **IPIM**, the same planner can be sped-up with substantially improved memory efficiency.

2) Comparison between TP-IPIM, IRIS and IRIS-M: We ran **IRIS** in both scenarios for 24 hours without getting

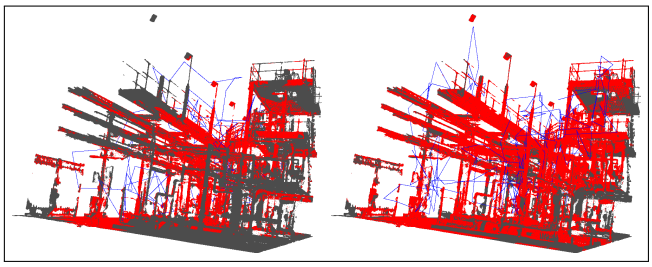

Fig. 5. Sample trajectories in **Plant-s** planned by **TP-IPIM**, with red covered, black uncovered and blue the inspection path. **TP-IPIM** plans within 2 hours (Up) and 3 hours (Bottom), with 55% and 62% coverage.

any results. Therefore, for the rest of the comparison, we use **IRIS-M**. In all scenarios, **TP-IPIM** achieves similar coverage results as **IRIS-M**. **IRIS-M** has asymptotically optimal guarantees, achieved by searching on an RRG. Despite its only 2 GB baseline memory cost, **IRIS-M** searches numerous number of inspection paths, resulting in huge total memory cost. As shown in Table. I, in **Plant-s** and **Plant**, out of 10 runs, **IRIS-M** ran out of memory for 6 and 10 runs with 96 GB total memory limit. In instances where **IRIS-M** exhausts its memory, we capture and save the best inspection path **IRIS-M** generated up to the point before the process terminated. On the other hand, **TP-IPIM** did not fail any of the runs, with only a 1 GB total memory limit. One might view the comparison unfair because **TP-IPIM** does not have any optimality guarantees. However, by just comparing the baseline memory consumptions, which always exists regardless of whether a search for optimality is performed, **IPIM** also brings in $\sim 25\times$ and $\sim 70\times$ memory efficiency in **Plant-s** and **Plant**.

Fig. 5 provides a visualization of the coverage of **TP-IPIM** for 2 hours and 3 hours planning time. The inspection path demonstrates that, as time increases, **IPIM** enables the planner to navigate through a cluttered area (see bottom right) to achieve better coverage.

VI. Summary

We present a set of primitive computations, called **IPIM**, to allow sampling-based inspection planning to efficiently use implicit environment models. Evaluation indicates that **IPIM** substantially reduces the memory cost in inspection planning of large, cluttered, and confined environments. Many avenues are possible for future work. For instance, can better efficiency be gained with other implicit environment models? And, how to incorporate uncertainty in the environment and perception?

Acknowledgment

We thank Abyss Solutions for collecting and providing the San Jacinto dataset. This work was supported by the ARC Research Hub in Intelligent Robotic Systems for Real-Time Asset Management (IH210100030).

References

[1] G. Papadopoulos, H. Kurniawati, and N. M. Patrikalakis, "Asymptotically optimal inspection planning using systems with differential constraints," in *ICRA*. IEEE, 2013, pp. 4126–4133.

[2] A. Bircher, K. Alexis, U. Schwesinger, S. Omari, M. Burri, and R. Siegwart, "An incremental sampling-based approach to inspection planning: the rapidly exploring random tree of trees," *Robotica*, vol. 35, no. 6, pp. 1327–1340, 2017.

[3] M. Fu, A. Kuntz, O. Salzman, and R. Alterovitz, "Asymptotically optimal inspection planning via efficient near-optimal search on sampled roadmaps," *IJRR*, p. 02783649231171646, 2023.

[4] J. J. Park, P. Florence, J. Straub, R. Newcombe, and S. Lovegrove, "Deepsdf: Learning continuous signed distance functions for shape representation," in *CVPR*, 2019, pp. 165–174.

[5] H. Wang, J. Wang, and L. Agapito, "Co-slam: Joint coordinate and sparse parametric encodings for neural real-time slam," in *CVPR*, 2023, pp. 13 293–13 302.

[6] M. Fu, O. Salzman, and R. Alterovitz, "Computationally-efficient roadmap-based inspection planning via incremental lazy search," in *ICRA*. IEEE, 2021, pp. 7449–7456.

[7] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *IJRR*, vol. 30, no. 7, pp. 846–894, 2011.

[8] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "Nerf: Representing scenes as neural radiance fields for view synthesis," *Communications of the ACM*, vol. 65, no. 1, pp. 99–106, 2021.

[9] S. Fridovich-Keil, A. Yu, M. Tancik, Q. Chen, B. Recht, and A. Kanazawa, "Plenoxels: Radiance fields without neural networks," in *CVPR*, 2022, pp. 5501–5510.

[10] A. Chen, Z. Xu, A. Geiger, J. Yu, and H. Su, "Tensorf: Tensorial radiance fields," in *ECCV*. Springer, 2022, pp. 333–350.

[11] E. Sucar, S. Liu, J. Ortiz, and A. J. Davison, "imap: Implicit mapping and positioning in real-time," in *ICCV*, 2021, pp. 6229–6238.

[12] N. Rahaman, A. Baratin, D. Arpit, F. Draxler, M. Lin, F. Hamprecht, Y. Bengio, and A. Courville, "On the spectral bias of neural networks," in *ICML*. PMLR, 2019, pp. 5301–5310.

[13] M. Tancik, P. Srinivasan, B. Mildenhall, S. Fridovich-Keil, N. Raghavan, U. Singhal, R. Ramamoorthi, J. Barron, and R. Ng, "Fourier features let networks learn high frequency functions in low dimensional domains," *Advances in neural information processing systems*, vol. 33, pp. 7537–7547, 2020.

[14] T. Müller, B. McWilliams, F. Rousselle, M. Gross, and J. Novák, "Neural importance sampling," *ACM Transactions on Graphics (ToG)*, vol. 38, no. 5, pp. 1–19, 2019.

[15] T. Müller, A. Evans, C. Schied, and A. Keller, "Instant neural graphics primitives with a multiresolution hash encoding," *ACM Transactions on Graphics (ToG)*, vol. 41, no. 4, pp. 1–15, 2022.

[16] Z. Zhu, S. Peng, V. Larsson, W. Xu, H. Bao, Z. Cui, M. R. Oswald, and M. Pollefeys, "Nice-slam: Neural implicit scalable encoding for slam," in *CVPR*, 2022, pp. 12 786–12 796.

[17] G. S. Camps, R. Dyro, M. Pavone, and M. Schwager, "Learning deep sdf maps online for robot navigation and exploration," 2022. [Online]. Available: https://arxiv.org/abs/2207.10782

[18] S. T. Bukhari, D. Lawson, and A. H. Qureshi, "Differentiable composite neural signed distance fields for robot navigation in dynamic indoor environments," *arXiv preprint arXiv:2502.02664*, 2025.

[19] Y. Li, X. Chi, A. Razmjoo, and S. Calinon, "Configuration space distance fields for manipulation planning," *arXiv preprint arXiv:2406.01137*, 2024.

[20] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," in *Seminal graphics: pioneering efforts that shaped the field*, 1998, pp. 347–353.

[21] V. Vasilopoulos, S. Garg, P. Piacenza, J. Huh, and V. Isler, "Ramp: Hierarchical reactive motion planning for manipulation tasks using implicit signed distance functions," in *IROS*, 2023, pp. 10 551–10 558.

[22] N. Ravi, J. Reizenstein, D. Novotny, T. Gordon, W.-Y. Lo, J. Johnson, and G. Gkioxari, "Accelerating 3d deep learning with pytorch3d," *arXiv:2007.08501*, 2020.

[23] P. Kafka, J. Faigl, and P. Váňa, "Random inspection tree algorithm in visual inspection with a realistic sensing model and differential constraints," in *ICRA*, 2016, pp. 2782–2787.

[24] Q.-Y. Zhou, J. Park, and V. Koltun, "Open3D: A modern library for 3D data processing," *arXiv:1801.09847*, 2018.

[25] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[26] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," 2019. [Online]. Available: https://arxiv.org/abs/1711.05101