5

Dense Map Representations

Victor Reijgwart, Jens Behley, Teresa Vidal-Calleja, Helen Oleynikova,
Lionel Ott, Cyrill Stachniss and Ayoung Kim

We now shift our focus to a different aspect of SLAM, specifically its mapping component. The mapping problem is approached with the assumption that the robot's pose is known, and the objective is to construct a dense map of its surroundings. Indeed, typical approaches first solve for the robot trajectory using the SLAM back-end —as discussed in the previous chapters— and then reconstruct a dense map given the trajectory. In this chapter, we illustrate the details of the dense map representation, focusing on the map elements, data structures, and methods.

Early mapping approaches were predominantly based on sparse, landmark-based solutions as discussed in Chapter 1 that extract only a few salient features from the environment. However, the increase in compute capabilities paired with the advent of accurate 3D range sensors, such as mechanical 3D LiDARs or RGB-D cameras that provide detailed 3D range measurements at high frequencies, led to an increasing research interest in dense map representations. Dense maps are crucial for downstream tasks that require a detailed understanding of the environment, such as planning, navigation, manipulation and precise localization. This chapter explains how these dense methods leverage the full spectrum of range sensor data to refine and update comprehensive maps.

The chapter begins by presenting key sensor types that facilitate dense mapping, primarily focusing on range sensors that produce detailed range measurements. The chapter continues with an introduction to fundamental representation elements and data structures in Section 5.2, that we then tailor to specific applications in Section 5.4. Contrasting with sparse landmark-based mapping, the choice of a dense map representation hinges on the sensor types used and the intended downstream applications. Key factors influencing the selection of the representation type are summarized in Section 5.5.

5.1 Range Sensing Preliminaries

Before we delve into dense map representations, we briefly summarize a key sensing modality, range sensors, often used for SLAM, providing the necessary context for the following discussion. Such sensors produce range measurements to the objects

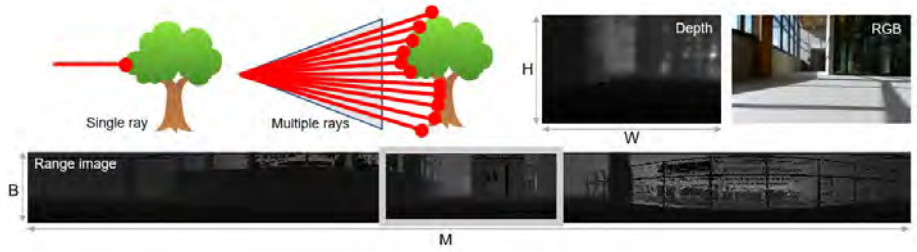


Figure 5.1 Single ray and multi-ray types for LiDAR sensors. Sample data from real-world is visualized to show RGB, depth, and LiDAR range image.

in the environment, including LiDAR sensors, time-of-flight (TOF) cameras, RGB-D cameras, and stereo cameras. Here, we concentrate on the most commonly used LiDAR sensors and RGB-D cameras that are predominately used in outdoor and indoor environments for SLAM and dense mapping.

5.1.1 Sensor Measurement Model

Let us start with a brief summary of the sensing mechanism and associated measurement model. In the case of LiDAR sensors,[1] the range measurements are generated using laser beams that are emitted, reflected by the environment, and then detected [941]. By measuring the time t_{emit} when the laser beam is emitted and the time of detection t_{detect}, we can derive the range r using the speed of light c as follows:

$$r = \frac{c\,(t_{\text{detect}} - t_{\text{emit}})}{2}. \tag{5.1}$$

A single ray measurement can be enhanced into a two-dimensional (2D) or three-dimensional (3D) collection of points by employing an array of rays that move in a designated pattern, such as a 360-degree rotation or a specific shape. The collection of points generated by the sensor is referred to as point clouds, which serve as the fundamental element for creating maps. The LiDAR measurements can also be represented using a range image $\boldsymbol{R} \in \mathbb{R}^{B \times M}$, where the range of each of the B beams is stored for a single complete turn of the sensor, *i.e.,* a complete 360° rotation. Thus, we have M measurements in the horizontal field of view of the sensor for each of the B beams.

Another commonly used range sensor in robotics applications are RGB-D cameras, such as Microsoft's Kinect and Azure Kinect DK, and Intel's RealSense. RGB-D cameras provide besides the RGB image $\boldsymbol{I}_{\text{RGB}} \in \mathbb{R}^{3 \times H \times W}$ of height H and width W, a depth map $\boldsymbol{I}_D \in \mathbb{R}^{H \times W}$ of the same dimension, where each pixel location contains the depth or range. To generate the depth map $\boldsymbol{I}_D$, early RGB-D cameras

[1] We illustrate this chapter using simple 2D and mechanically rotating 3D LiDARs, while other solid-state and flash LiDARs will be introduced in Chapter 8

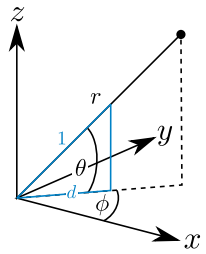
use a structured infrared (IR) light source with a known pattern that is projected onto the environment. The distance of individual pixels is then determined from the distortion of the known pattern. As sunlight usually interferes with this sensing mechanism, such RGB-D cameras using projected light are mainly used in indoor environments. Fortunately, newer generations of RGB-D sensors introduce an IR texture projector or TOF less affected by interferences, supporting outdoor applications.

5.1.2 Conversion to Point Cloud

Using the intrinsics of a range sensor (*e.g.,* a LiDAR or RGB-D camera), we can convert a range image $\boldsymbol{R}$ or a depth map $\boldsymbol{I}_D$ into a point cloud $\mathsf{P} = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_N\}$, where points $\boldsymbol{p}_i \in \mathbb{R}^3$ are expressed in the local coordinate frame of the sensor. The point is the most fundamental unit in the map representation and will be discussed further in this chapter.

For conversion from LiDAR, the sensors provide an intrinsic calibration for each beam $(\phi_{i,j}, \theta_{i,j})$, where $1 \leq i < B$ and $1 \leq j < M$, consisting of the azimuthal angle $\phi_{i,j} \in [0, 2\pi]$ and polar/inclination angle $\theta_{i,j} \in [-\pi, \pi]$ as depicted in Figure 5.2. Using these known angles of each beam, we can convert a range measurement $r_{i,j}$ at $\boldsymbol{R}_{i,j}$ into a three-dimensional point $\boldsymbol{p} = (x, y, z)$ as follows:

Figure 5.2



$$x = r_{i,j} \cos(\theta_{i,j}) \cos(\phi_{i,j}) \tag{5.2}$$

$$y = r_{i,j} \cos(\theta_{i,j}) \sin(\phi_{i,j}) \tag{5.3}$$

$$z = r_{i,j} \sin(\theta_{i,j}) \tag{5.4}$$

For an RGB-D camera, we commonly use a pinhole camera model to convert the ranges $r_{u,v} = \boldsymbol{R}_{u,v}$ at pixel location (u, v) into a three-dimensional coordinate. For this, we use the intrinsics of the camera $\boldsymbol{K} \in \Re^{3\times4}$ to convert a homogeneous coordinate $\boldsymbol{x} = (u, v, r_{u,v})$ in the image coordinate into a point $\boldsymbol{p}$ in the camera coordinate:

$$\boldsymbol{p} = \boldsymbol{K}^{-1}\boldsymbol{x}. \tag{5.5}$$

The resulting point cloud is said to be *organized* or *unorganized* depending on how the points are structured. When converted from a depth map, the point cloud is organized, and each point location is structured with respect to the pixel location of the associate depth map. This can be exploited to compute neighboring points by a simple indexing. On the other hand, the point cloud generated by a LiDAR sensor is more complicated. For example, the generated point cloud is organized in the static 3D mechanical LiDAR. However, the organization of the point cloud no longer

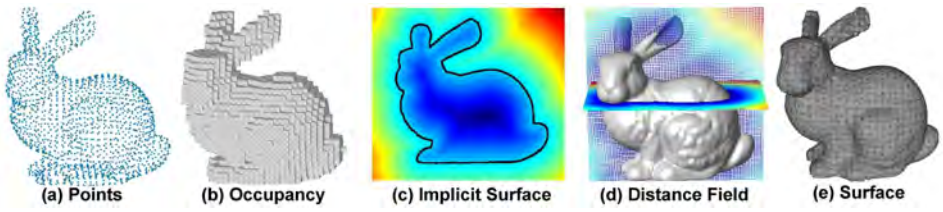


Figure 5.3 Examples of common dense representations.

holds in the case of non-repeated pattern solid-state, flash LiDARs, or mechanically rotating LiDAR under motion distortion. In many cases, the unorganized points are distorted due to the movement of the sensor and measurement neighborhood in the range image is not necessarily correlated to spatial neighborhood. Therefore, an estimate of the motion of the sensor while completing a sweep, *e.g.,* inertial measurement unit (IMU) measurements or odometry information, together with the information of the per-beam time is necessary to undistort an LiDAR point cloud to account for the motion of the sensor [1137, 264, 1226]. For more details on motion distortion and compensation, refer to Chapter 8.

5.2 Foundations of Mapping

A map, generated with sensors' information and data processing approaches, is a symbolic structure that models the environment [1095, 142]. One thing to be noted here is that the map representation can be diverse, and many different representations exist for the same spatial information (as in Figure 5.3). The choice and accuracy of the scene representation strongly impact the performance of the task at hand, and thus the representation should be determined by the use case. For instance, motion estimation and localization in robotics favor sparse representation, such as 3D points features [793, 903] in order to exploit these features for consistent robot pose estimation. On the other hand, a key objective of scene reconstruction is an accurate, dense, and high-resolution map, for example, for inspection purposes [514, 828, 928]. Similarly, path planning tasks require dense information such as obstacle occupancy or closest distance to collision for obstacles avoidance [780, 312, 1114]. Overall, this chapter examines the following three questions.

Q1. What quantity do we need to estimate for dense mapping? The most commonly used quantities to represent the environment are *occupancy* and *distance*. Occupancy is a key property in mapping for distinguishing between free and occupied space. Distance estimation provides a more robot-centric interpretation of free and occupied space by measuring the range to nearby surfaces or objects.

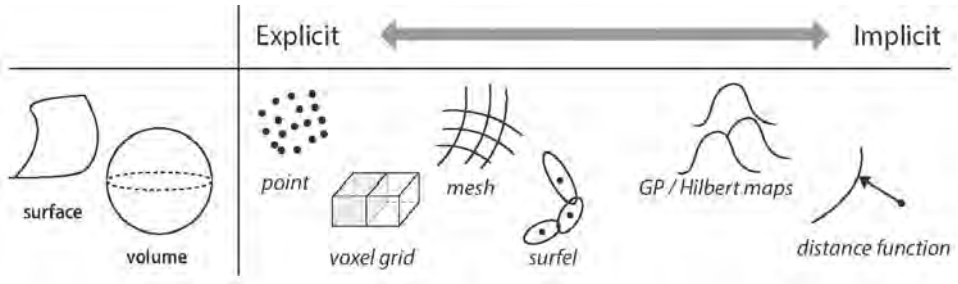


Figure 5.4 The representation can be either explicit or implicit. The illustration is simplified for clarity. In reality, the abstraction is not clearly separable and can often be applied in a combined manner.

Q2. How should the world be represented? This is the question of what space abstraction we use for representation. Broadly, representation can be either explicit or implicit. Explicit space abstractions are further classified based on the type of geometry they utilize, while implicit representation can be categorized based on their choice of functions. The list of abstraction types are illustrated in Figure 5.4.

Q3. What data structure and storage should be used? The chosen representation should be stored in memory for later use. The data structure and storage method should be selected based on the specific application and intended usage.

In the literature, a wide variety of approaches exist for generating a dense representation of the scene using range sensors, varying in terms of their estimated quantities, space abstraction, storage structure, continuity, and application areas.

Beginning the discussion on different representations, we first explore the primary quantities estimated from range measurements. The focus is on understanding the key quantities predominantly estimated in the mapping phase. We will provide a concise overview of the basic definitions of each quantity, elaborating their significance and the specific contexts in which they play a crucial role.

5.2.1 Occupancy Maps

Since their introduction over three decades ago by Elfes and Moravec [312, 780], occupancy grids have been widely used. Their simplicity and computational efficiency have made occupancy grids[2] popular when mapping indoor (and even outdoor) environments. In the simplest scenario, the estimated quantity is the *probability* of a cell being occupied. In this case, the occupancy of the cell is modeled as a probability of that cell containing an obstacle, with occupancy equal to 1 for occu-

[2] Occupancy mapping can be conducted without grid-based methods (e.g., GPOM [822]). In this chapter, we will focus on grid-based mapping for simplicity.

pied cells and 0 for cells deemed empty. Essentially occupancy mapping is a binary classification problem to predict the binary class probability of each cell.

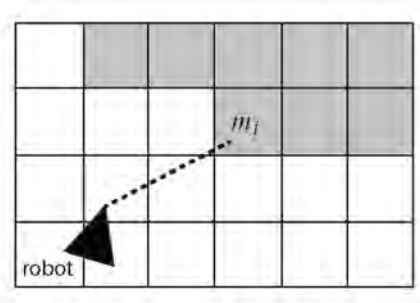


Figure 5.5 A simple ray-casting example in 2D. Given a range measurement at a certain (computed or estimated) azimuth, the return of the range measurement indicates the existence of an obstacle.

Given a set of sensor measurements $\boldsymbol{z}_{1:t}$ and a set of sensor poses $\boldsymbol{x}_{1:t}$, the probability of being occupied for each cell in the map $\boldsymbol{m}$ is modeled as $p(\boldsymbol{m}|\boldsymbol{z}_{1:t}, \boldsymbol{x}_{1:t})$. A sample map is shown in Figure 5.5. Assuming that each cell m_k is independent and that the measurements are conditionally independent, the update of occupancy can be formulated efficiently using the well-known log-odds form [780],

$$l(m_k|\boldsymbol{z}_{1:t}) = l(m_k|\boldsymbol{z}_{1:t} - 1) + l(m_k|z_t)\,, \tag{5.6}$$

where $l(\cdot|\cdot) = \log(o(\cdot|\cdot))$, and $o(\cdot|\cdot)$ is the odds form:

$$o(m_k|\boldsymbol{z}_{1:t}) = \frac{p(m_k|\boldsymbol{z}_{1:t})}{1 - p(m_k|\boldsymbol{z}_{1:t})}\,. \tag{5.7}$$

The main advantage of occupancy mapping is shown in (5.6), where only the previous occupancy value and the inverse sensor model $l(m_k|\boldsymbol{z}_t)$ are needed to update the probability through a simple addition. Despite these benefits, occupancy grids rely on very strong assumptions of the environment to be efficient. Notably, the assumption that the likelihood of occupancy in one cell is independent of other cells disregard spatial correlations that can be important to infer occupancy in unobserved nearby regions. Additionally, traditional occupancy grids require the discretization of the environment be defined a priori which makes the spatial resolution constant throughout the map.

5.2.2 Distance Fields

Another way of representing the geometry surrounding the robot is not through *probabilities* of occupancy, but rather by describing the boundary between free and occupied space. In an ideal world, we could describe the shape and location of this

boundary using an analytical function in three variables (x, y and z) which reaches 0 whenever a point $\boldsymbol{p} = (x, y, z)$ lies on the surface. In other words, the function's zero crossings correspond to the surface itself. Such a function is known as an *implicit surface*. By convention, the function's sign is negative when $\boldsymbol{p}$ lies *inside* an object and positive *outside*. A simple example in Figure 5.6 illustrates how this implicit function values are determined.

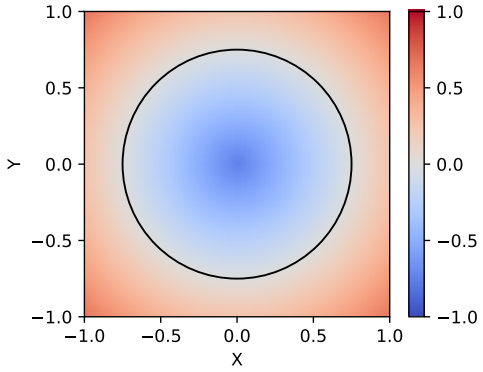


Figure 5.6 A solid 2D disk represented as an explicit surface (black outline) and implicit surface (colored field). The implicit surface function is negative inside the object (blue) and positive outside (red). Note how the function's zero crossings correspond to the surface itself.

There are many advantages to continuous implicit surface representations. Given that there is no fixed resolution, they can represent objects of arbitrary shapes at any level of detail. Furthermore, they make it possible to check if a given point in space is inside or outside an obstacle by simply evaluating the function's sign.

Different types of functions can be employed to model implicit surfaces, with the Euclidean Signed Distance Function (ESDF) being a prevalent option. At any query point, the ESDF expresses the distance to the nearest surface (indicated by the magnitude of the ESDF) and whether the point is inside an obstacle (indicated by the sign of the ESDF). ESDF representations are commonly used by accelerated geometric algorithms for tasks such as collision checking. Furthermore, high-quality proximity gradients can be derived from the ESDF for optimization-based motion planning and shape registration.

The ESDF is computed by finding the (Euclidean) closest surface point for each point in the map. For a known surface, this can efficiently be done using techniques such as Fast Marching. However, for surface estimation, the *projective* signed distance is more commonly used as it can efficiently be computed from measurements and is better suited for filtering. Given a measurement ray going through a query point, the projective distance is defined as the distance from the beam's endpoint to the query point *along the ray*. This eliminates the need to search the closest point

explicitly. Although the projective distance overestimates the Euclidean distance, its zero crossings (the estimated surface) remain correct. The standard approach of estimating implicit surfaces, proposed by Curless and Levoy [242] and popularized in the field of robotics by KinectFusion [807], combines the projective signed distances for all measurements using a simple weighted average. To reduce the impact of overestimates, the projective signed distance function is typically clamped to a fixed range, named the truncation band, in which case it is called the Truncated Signed Distance Function (TSDF). Note that ESDFs can efficiently be computed from TSDFs and occupancy maps, as will be described in Section 5.4.4.

5.2.3 Occupancy Maps or Distance Fields?

Being volumetric methods, a shared aspect of these estimated quantities in occupancy and implicit representations is that they model the geometry by estimating a quantity of interest everywhere in the observed volume. However, each representation fundamentally prioritizes different things. Which option is best, therefore, depends on the application. We will briefly summarize two key differences.

Directness in modeling: Given that measurement rays *directly* tell us which parts of space are free, occupied or unobserved, maps based on occupancy probabilities can be updated using fewer heuristics and assumptions. In contrast, implicit surfaces typically model the distance to the surface. This can be computed exactly for a known surface, but not from partial measurements. For surface estimation, they therefore rely on distance proxies such as the previously introduced TSDF.

Smoothness: Implicit surface maps are inherently smoother than occupancy maps, which model a binary property. The smoothness of implicit surfaces has many benefits. Most importantly, it makes them differentiable. The resulting proximity gradients are valuable for many applications. Smoothness also reduces the approximation errors resulting from discretization and makes it possible to obtain good, sub-pixel resolution estimates through interpolation. However, since discontinuities cannot be represented smoothly, implicit surfaces tend to miss thin obstacles.

5.3 Map Representations

5.3.1 Explicitness of Target Spatial Structures

As summarized in Figure 5.4, the representation can be classified based on their explicitness and target space. In 3D mapping, representing volume is straightforward; however, surfaces are equally important in robotic mapping for enabling downstream tasks. We can consider four major categorization: explicit surface, implicit surface, explicit volume, and implicit volume representations.

For surfaces, we can either *explicitly* or *implicitly* represent a surface. Defined as a 2D manifold, explicit type of representation aims to characterize the space

in terms of their boundary of the objects in the scene. The simplest abstraction that can represent the boundary is directly the point cloud produced by the range sensors. Another general representation of the surface is the polygon mesh (Section 5.4.3), which comprises vertices, edges, and faces. These meshes have the ability to encode the directed surfaces of a volume by forming connected closed polygons, more commonly triangles. Surfels (Section 5.4.2) are also popular abstraction widely used in mapping. Surface representations are a key for any visualization application, but also are used for rendering simulated environments, augmented reality or for computed aided design and 3D printing.

Similar strategies are employed in 3D volume modeling. Naive point-based representations are commonly used in LiDAR SLAM. Additionally, occupancy or distance-based voxels (Section 5.4.4) are popular choices for explicit representation. When storing volumetric maps, careful consideration of data storage is necessary to minimize computational costs. Implicit representations for volumetric mapping are also utilized, typically through functions. GP (Section 5.4.5) and Hilbert maps (Section 5.4.6) are well-known examples of implicit representations.

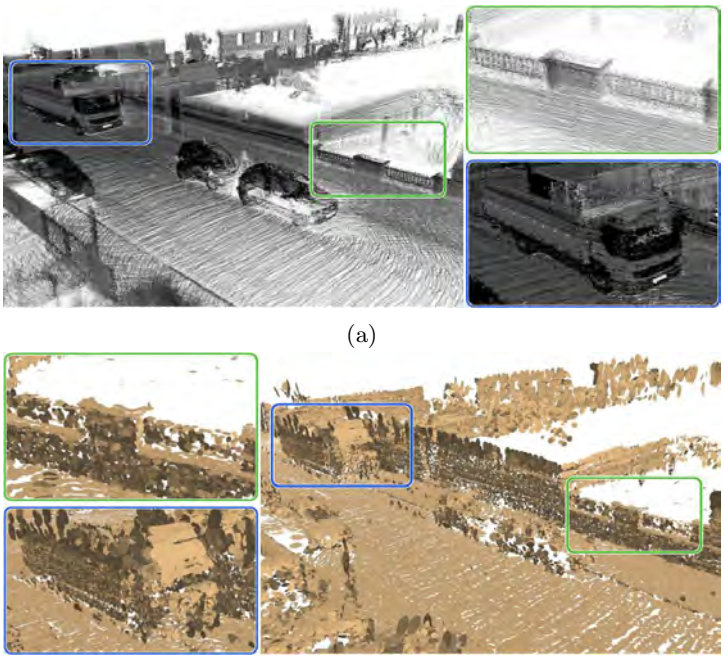
5.3.2 Types of Spatial Abstractions

5.3.2.1 Points

Given range measurements, a straight-forward dense map representation is using clouds. For generation, we accumulate the point clouds P_t recorded at time t in the local coordinate frame $\mathcal{F}^t$ using the estimated global pose $\boldsymbol{T}_t^1$ in a global map point cloud P_M. Also common practice is to assume our global coordinate frame of P_M is given in the coordinate frame of the first point cloud $\mathcal{F}^1$.

Since simply accumulating point clouds $\mathsf{P}_1, \ldots, \mathsf{P}_t$ does not scale to larger environments, a common strategy is to discard redundant measurements of the same spatial location. To this end, most methods [1264, 264, 1137] use efficient nearest neighbor search, such as voxel grids or hierarchical tree-based representations (see Section 5.3.3), to subsample and store the point clouds, *e.g.,*, store only a limited number of points per voxel [264, 1137] or only specific points that meet a certain criterion are stored [1264]. Additionally, a representation of only keyframes where only a few point clouds are explicitly stored is possible, but this requires to determine when a keyframe or submap needs to be generated.

Being the most elemental representation form, a point cloud map can be converted into other representations, *e.g.,* a mesh via Poisson surface reconstruction [1135] or a Signed Distance Function (SDF) via marching cubes [1136]. Unfortunately, this is feasible only with additional data, such as the viewpoint of a point's measurement. Yet, this information may be lost when merging multiple measurements into a point cloud map. Therefore, assumptions about the surface's direction are often required to discern inside or outside regions.



(a)

(b)

Figure 5.7 Qualitative comparison of maps generated by accumulating point clouds and surfels from a sequence of LiDAR scans from the KITTI dataset [381] Sequence 07. (a) Point cloud map. The brightness of points indicates the remission of the LiDAR measurements. (b) Corresponding surfel map based on circular disks. The complete map with all accumulated point clouds use 2.95 GB, while the corresponding surfel map by SuMa [73] uses only 160 MB.

5.3.2.2 Surfels

While point cloud maps directly represent the measurements, the stored points do not contain surface information or can represent from which direction a point has been measured. With surfels [877], we can encode such information by adding directional information to a point. Surfels are commonly represented via circular or elliptic discs [557, 73, 1181, 110, 109], or more generally ellipsoids [1046, 1047, 290, 1313] modeled with a a Gaussian. So-called splatting [1313, 109] allows to integrate texture information but also blend overlapping surfels into coherent renderings of a specific viewpoint.

A commonly employed circular surfel representing a circular disk is defined by a location $\boldsymbol{p} \in \mathbb{R}^3$, a normal direction $\boldsymbol{n} \in \mathbb{R}^3$, and a radius $r \in \mathbb{R}$. As rendering primitive such surface patches can be efficiently rendered using the capabilities of modern graphics processing units (GPUs), which can be exploited to efficiently

render arbitrary views. This accelerates point-to-surfel associations and leads to the substantial memory reduction.

Figure 5.7 qualitatively compares a point cloud-based and a surfel-based map representation. A dense point cloud can accurately represent the environment with a high level of detail, but at the cost of memory. In contrast, while losing fine details as multiple measurements get aggregated into a single surfel, significant memory usage can be reduced while preserving the main structural details of larger surfaces.

Closely related to the explicit geometric representation of surfaces via surfels, *i.e.,* small circular surface patches, is the representation via a normal distributions transform (NDT) [89, 1041]. Using NDT, the space is subdivided into voxels and the points inside a voxel are approximated via a normal distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, having estimated mean $\boldsymbol{\mu}$ and covariance from the enclosed points $\boldsymbol{\Sigma}$. The eigenvalues $\lambda_1 < \lambda_2 < \lambda_3$ and corresponding eigenvectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3$ of the covariance can be used to estimate the surface properties inside a voxel. For planar surfaces ($\lambda_1 \ll \lambda_2$), the eigenvector $\boldsymbol{v}_1$ of the smallest eigenvalue λ_1 corresponds to the surface normal. Thus, for planar surfaces the NDT represents a surfel, and can also more accurately represent point distributions that cannot be approximated via a surfel. In that sense, the NDT is a hybrid representation that is explicit due to the space division into a voxel grid, but also implicit due to the representation of voxels via a normal distribution which continuously represents the space inside a voxel.

5.3.2.3 Meshes

While describing local surface properties, both point clouds and surfel maps are still relatively sparse as they do not model the surface's connectivity. One way to get a more complete understanding is to use meshes, which describe the surface as a set of points that are connected to form a collection of polygons. This, in turn, makes it possible to represent watertight surfaces, query and interpolate new surface points, and efficiently iterate along a connected surface.

In meshing terminology, each polygon is referred to as a *face*, and each corner point as a *vertex*. The most common types are triangle meshes, where each face is bounded by three vertices, and quad meshes, whose faces are bound by four vertices. Note that a polygon, or face, can always be broken down into an equivalent set of triangles; hence, triangle meshes are not only the simplest but also the most general.

A mesh is a very flexible and memory-efficient representation because the number of faces and vertices can be directly adapted to the surface complexity and required detail. For example, a plane of any size can be represented with just two triangular faces and four vertices. Furthermore, meshes are well-suited for parallel processing and rendering. Meshes are often used in applications that overlap with computer graphics, such as rendering, surface analysis, manipulation, and deformation, and more generally in applications involving digital models, simulation, or surface-based algorithms, such as path planning for ground robots.

5.3.2.4 Voxels

Point clouds and meshes are well suited to represent properties of the environment that are defined along surfaces. However, certain estimated quantities, including occupancy and Signed Distance, are defined throughout the entire volume. One straightforward way to store and process volumetric properties is to discretize them over a regular grid. Discretized occupancy and Signed Distance maps are called occupancy grids and Signed Distance Fields, respectively.

Generalizing the concept of 2D pixels, the cells in a 3D grid are referred to as voxels. Given a grid's regular structure, each voxel can easily be assigned a unique index and stored in a data structure. Note that a voxel is merely a container or, more formally, a space partition. The significance of its contents varies from one method to another. In a classic occupancy grid, a voxel's value represents the likelihood that any point in the voxel is occupied. Hence, the occupancy at an arbitrary point in the map is equal to the value of the voxel that contains the point. However, a voxel's value does not have to represent a constant little cube. For example, voxels in a Signed Distance Field estimate the signed distance at each voxel's center. To retrieve the signed distance at an arbitrary point, one would therefore query the voxels that neighbor the point and obtain the point's value using interpolation. Finally, some applications even use (sparse) voxel grids to store and efficiently query non-volumetric properties, such as points or surface colors.

5.3.2.5 Continuous Functions

Functions are a key abstraction for mapping in a continuous manner. The problem of mapping in this case is reduced to fitting a parametric or non-parametric function, *i.e.,* solving a regression problem. Most of the above-mentioned space abstractions require the discretization of the environment to be defined *a priori*, which usually makes the spatial resolution constant throughout the map. Continuous functions, however, parametric or non-parametric, give more flexibility allowing the resolution to be recomputed and also provide interpolation capabilities to fill up data gaps.

Some parametric functions such as infinite lines in 2D [1110] and planes in 3D [542, 382, 1110] require making strict assumptions about the environment and limit the representation of the scene. However, these representations are efficient in terms of memory consumption and computational complexity. Control points-based functions (*e.g.,* B-splines [937]) or non-parametric (*e.g.,* GP-based) have the ability to model the environment with fewer assumptions, still in a continuous manner. From occupancy [822], implicit surface [1183, 643], distance fields [1192], and surface itself [1126], GP-based representations are a popular choice to represent the environment —despite their high computational complexity— because of their probabilistic nature, which enables uncertainty quantification and inference over both observed and unseen areas [386].

A key advantage of the continuous functions for mapping is that if they are chosen

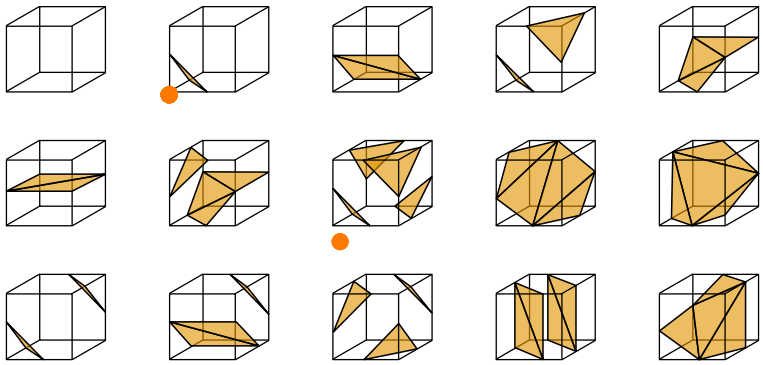


Figure 5.8 The algorithm works by projecting the cubes into the implicit surface, querying the sign of the values at the corners of the cubes, and looking up which one of the 15 configurations these values map to. (Image credit: Ryoshoru, "*Marching cubes*", licensed under CC BY 4.0. Source: https://commons.wikimedia.org/wiki/File:MarchingCubesEdit.svg)

to be at least once differentiable they will be able to provide gradients. Gradient information can be key for localisation to compute surface normals [1193], loop closure to compute terrain features [385], for data fusion [643] and planning [1193] applications.

5.3.2.6 Conversions

The abstractions listed above are not always used exclusively; they are often converted from one form to another or employed simultaneously in multiple forms.

For instance, explicit geometry, such as points and meshes, can be transformed into an implicit surface. One flexible method to compute the signed distance at any point in space is through a *closest point lookup*, which can be performed against any explicit geometry and on-demand, only when and where needed. Alternatively, the signed distance can be computed across all points on a regular grid using *wavefront propagation*, which can efficiently be implemented via the fast marching method [996].

Conversely, it is common to convert implicit surfaces into mesh representations. The original technique for converting distance fields into meshes is known as Marching Cubes [703]. The algorithm divides the implicit surface into a grid of fixed-size *cubes*, which it processes independently. Each cube generates a set of triangle elements based on the implicit surface's values at its eight corners (Figure 5.8). The positions of their vertices are then refined through linear interpolation. Meshes, including those from BIM or CAD models, can, in turn, be sampled to create points or surfels.

Occasionally, a discrete representation must be converted into a continuous one. This is typically achieved by solving an optimization problem over the parameters

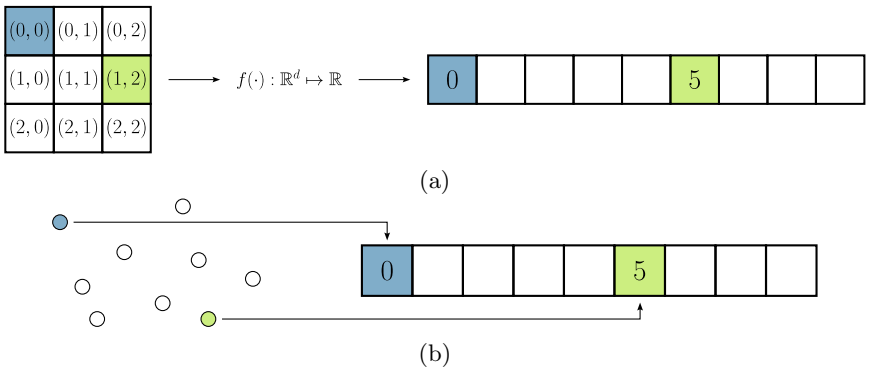


Figure 5.9 (a) Mapping of logical grid coordinates to an underlying naive array-based storage through a function $f(\cdot)$ that uniquely maps coordinates to linear indices. (b) Storage of unordered data directly into an array structure.

of the continuous representation, minimizing the fitting error with respect to the discrete data.

5.3.3 Data Structures and Storage

Previously introduced abstractions all need to be stored in memory. In this section, we explore how various abstractions are stored in memory by examining the choice of data structures along with their advantages and disadvantages.

5.3.3.1 Naive Data Storage

For many representations a simple dynamically resizeable array is a reasonable starting point. For data with a pre-defined spatial partitioning two things are needed, the type of data to store and a conversion function from a spatial coordinate to an index coordinate. This is often used when building maps representing occupancy or signed distance values. For irregular data, only the type of data to store is needed, for example point clouds or surfels. The naive storage of ordered data using a mapping function and unordered pointcloud data is illustrated in Figure 5.9.

The benefit of this naive approach is that it is simple and provides fast random access. The trade-off is, that large amounts of memory can be required for such a representation. Also while read and modify operations are fast, changing the spatial dimension of the representation can be very costly as the content of the entire data structure needs to be copied.

5.3.3.2 Hash Map

A natural extension to address the limitations of the naive storage method described above is to use a hash table. This approach divides the map into shards, applies

a *hash function* to convert the coordinates of each shard into a single value, and stores the sharded data in a table indexed by this hash value. The shards are typically chosen to represent map subregions with well-established coordinates, such as cubes in a regular grid. These cubes may correspond to individual voxels or fixed-size groups of voxels, referred to as voxel *blocks*. Alternatively, they can also store other elements like points, surfels, or mesh fragments contained in their respective subregions.

A hash table retains the fast $\mathcal{O}(1)$ look-up time of a fixed array while allowing the map to grow dynamically without reallocation. Three key considerations must be addressed when using a hash table for dense map storage:

1 **Granularity of the sharding:** Smaller shards improve sparsity by allocating data only where necessary. However, the number of shards should not grow too large, as this reduces the hash table's insertion performance and memory efficiency. This trade-off is particularly relevant when hash maps are used to store properties that only exist along the surface.
2 **Hash function:** An ideal hash function distributes keys evenly across the table, even when the data is spatially adjacent, as is often the case in mapping scenarios.
3 **Collision resolution:** The method for handling hash collisions, whether through linear chaining (where each entry contains a linked list) or open addressing, significantly affects the performance of the hash table.

In most cases, hash tables offer a good balance of fast access and efficient insertion and deletion of data. However, they may require initial tuning to perform well for a given application.

5.3.3.3 Tree-based Data Structures

Another option to efficiently store spatial data while only occupying memory for relevant parts of the environment is to use hierarchical, tree-based representations. Just like hash tables, trees generally enable efficient access and insertions. However, their unique strength is their hierarchical structure, which can be used to efficiently store multi-resolution data and speed up spatial operations such as nearest neighbor search. The most prominent tree variants are kD-trees [79], bounding volume hierarchies (BVH) [231], and octrees [759].

Among them, the octree efficiently searches neighbors with the capability to integrate novel measurements incrementally. The octree is a tree representing each node by a so-called *octant* that refers to a subspace. An octant is defined by a center $\boldsymbol{c} \in \mathbb{R}^3$ and an extent $e \in \mathbb{R}$, corresponding to an axis-aligned bounding-box. Each octant has potentially 8 child octants of extent $\frac{1}{2}e$, as depicted in Figure 5.10. Common practice is to store points only in the leaf octants (*i.e.,* octants without children) and determine subsets of points at inner octants of the tree structure by tree traversal.

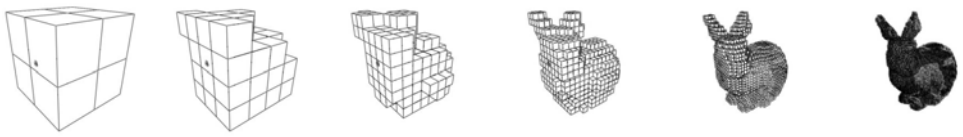


Figure 5.10 Example of an octree and its octants at different levels of the tree hierarchy. Each level of an octree subdivides the space in more fine-grained octants. Note that deeper levels of the octree only represent the occupied space.

To construct an octree, we iteratively divide space into octants within an axis-aligned bounding box encompassing a point cloud P. Each division splits P into subsets $\mathsf{P}_1, \ldots, \mathsf{P}_8$, corresponding to 8 child octants of half extent $\frac{1}{2}e$. Non-empty subsets P_i form child octants with center $\boldsymbol{c}$ and extent $\frac{1}{2}e$, stopping at a specific octant size or a minimal point count. Once constructed, updates and insertions can efficiently be performed by traversing the tree structure and adding inner nodes as needed. When new data is inserted that falls outside of the tree's root octant, the tree can be extended by creating a new root node and assigning the new data and the old root node to its children.

In contrast to voxel grids, an octree represents only data-containing subspaces, enabling efficient storage of occupied space. However, this memory efficiency requires tree traversal to access specific leaf octants, potentially leading to increased runtime to locate points. Additionally, the tree structure itself must be explicitly represented, incurring extra memory overhead. Several recent approaches address these memory overheads [313, 826, 77].

5.3.3.4 Hybrid Data Structures

To balance memory requirements and runtime for data access, several data structures combine the advantages of different data structures in specific ways, leading to hybrid representations. In this tradeoff, we accept less efficient memory usage but enable more efficient memory access.

For example, hashed voxel grids [814] combine the strengths of dense voxel grids and hash tables by splitting the environment into fixed-sized, dense blocks (e.g. $8 \times 8 \times 8$ voxels), which are in turn stored in a hash table. Thanks to the hash table's flexibility, blocks only need to be allocated in locations that contain meaningful information (e.g. near the surface). At the same time, using a plain 3D array to store the voxels inside each block ensures that operations remain simple, efficient, and even suitable to GPU acceleration.

Another option is to combine hash tables with trees. In a similar vein to hashed voxel grids, the VDB [3] data structure [798, 797] splits the space into hashed blocks, but stores a hierarchical tree inside each block. This data structure provides all the

[3] VDB refers to sparse volumetric data and stands for several different thing as Voxel Data Base or Volumetric Data Blocks. Here we follow the terminology used in [798].

Section	Space Abstraction Type	Representing Map Entities
5.4.1	Points	Surface
5.4.2	Surfels	Surface
5.4.3	Mesh	Surface (connected)
5.4.4	Voxels	Occupancy or Implicit surface
5.4.5 - 5.4.6	Continuous function	Occupancy or Implicit surface

Table 5.1 *Summary of presented mapping methods.*

benefits of hierarchical trees, including multi-resolution representation and efficient nearest neighbor lookups. However, since each block has a fixed size, the maximum tree height is constant regardless of the size of the environment. Lookups and insertions can therefore be performed in constant time, and significantly faster than when using pure trees.

5.4 Constructing Maps: Methods and Practices

So far, we have explored the target quantities to estimate and the various space abstractions available for mapping. In this section, we will examine in detail the methods used to construct these map elements. The approaches are categorized by their main space abstraction, as shown in Table 5.1. Note that some of the methods use additional space abstractions to improve performance, for example, by grouping points into voxels for more efficient storage and faster queries.

5.4.1 Points

As mentioned in Section 5.3.2.1, naively storing points by accumulating the measured point clouds will not scale to large-scale environments and will lead to redundantly represented measurements. Therefore, most approaches [1264, 264, 1137] adopt a point-based representation in combination with a voxel grid or octree to represent the dense map. Moreover, the selection of a data structure is driven by the requirement for efficient nearest neighbor searches, essential for conducting scan registration through iterative closest point (ICP), where point correspondences must be iteratively established.

In order to handle large-scale environments, some methods, such as the well-known LiDAR SLAM LOAM [1264], filter the raw point clouds to extract corner and surface points thereby significantly reducing the point cloud size. A voxel grid is applied to store only a subset of points in the map representation, pruning redundant measurements. Stemming from the point-based voxelization used in LOAM, several follow-up approaches [1004, 1150, 845, 673, 901, 1005] refine the extraction of points [1004, 1150, 845], improve the optimization pipeline [845, 673], or integrate information of an IMU [901, 1005].

Another branch of methods handles the amount of point cloud data differently to avoid reliance on a capable feature extraction approach. Regularly sampling the point clouds via a voxel grid [264] significantly reduces the number of points per LiDAR scan and removes potentially redundant information. The key insight is here that points in the voxel grid are not aggregated and averaged, but original measurements are retained. Following these insights, Dellenbach *et al.* [264] and Vizzo *et al.* [1137] use this strategy to downsample an input point cloud, only storing a restricted number of points inside a voxel grid map.

Overall, as also mentioned in Section 5.3.2.1, the (hashed) voxel grid serves dual purposes: it abstracts space by storing a limited number of point measurements per voxel, and it facilitates accelerated nearest neighbor search through direct indexing of neighboring voxels.

5.4.2 Surfels

For surfels, similar strategies can be applied as for point clouds, but notably Stückler *et al.* [1046] and follow-up work by Dröschel *et al.* [290] use an octree to represent surfels at multiple levels in the octree hierarchy for data association. The so-called multi-resolution surfel maps indirectly represent the surfels via accumulated mean and covariance statistics, like a NDT.

In contrast, Whelan *et al.* [1181] store surfels as a simple list and exploit efficient rendering techniques to produce a projection for data association for RGB-D SLAM in indoor environments. In this case, surfels are explicit geometric primitives and, therefore, need to be directly handled to update the surfel properties (*i.e.*,, size and direction) accordingly [557]. A key contribution of Whelan *et al.* is leveraging a map deformation that directly deforms the surfels instead of relying on a pose graph optimization, which enables the use of the measurements represented by the surfels to deform the map on a loop closure detection. A similar strategy for map deformation of surfels was employed by Park *et al.* [852].

Similarly, Behley *et al.* [73] target outdoor environments, which makes it necessary to represent the surfels via multiple submaps of $100\,\mathrm{m} \times 100\,\mathrm{m}$ spatial extent that can be off-loaded from GPU memory. In contrast to ElasticFusiuon [1181], the approach relies on pose graph optimization but exploits that surfels can be freely positioned and ties surfels to poses enabling a straight-forward deformation of the map with pose-graph-optimized poses, which was also adopted by other approaches [1156].

5.4.3 Meshes

As introduced ealier, meshes offer an expressive, flexible way to represent connected surfaces. Mesh generation methods can be split into two families of approaches.

The first family directly converts the measured points into a mesh. In contrast, the second family splits the problem into two steps: reconstructing an implicit surface, followed by iso-surface extraction to get the final mesh (see Section 5.3.2.6).

Methods in the first family typically work *directly* by computing the Delaunay triangulation of the input point set and identifying the subset of Delaunay triangles that lie on the surface. A detailed overview of such methods is provided in [163]. When building directly from points, the mesh implicitly adapts itself to the sampling density. This can be an advantage, as it provides adaptive resolution, but it also means these methods are more sensitive to sampling irregularities and holes. In practice, direct meshing methods are chosen when the entire surface can be sampled densely with a very accurate depth sensor, for example, using surveying equipment.

The second family of approaches uses an implicit surface as an *intermediate* step, to simplify the process of fusing and filtering the data before extracting the final surface mesh. One intuitive way to generate the implicit surface from data is to estimate the distance to the surface at each point on a regular grid. As described in Section 5.2.2, the implicit surface's sign must also be set according to whether each point is inside or outside an object. This information is often determined based on estimated surface normals, which can for example be obtained by applying Principal Component Analysis (PCA) over a small surrounding area. However, as indicated in [484], such methods may yield implicit surfaces that are discontinuous. Tackling this issue, Carr et al. [161] model the implicit surface using a collection of Radial Basis Functions (RBFs) and fit these to the input points by solving a global optimization problem. The resulting implicit surfaces are smooth by construction and faithfully fill holes based on the global context. Unfortunately, solving the underlying large, dense optimization problem is computationally expensive. Shen et al. [1008] overcome this limitation by locally approximating the input points using moving least squares (MLS). Going one step further, Poisson Surface Reconstruction [553] fits the implicit function to the normals of the measured points by solving a partial differential equation (PDE), resulting in a sparse, computationally tractable optimization problem that is particularly robust to noise.

In robotics applications, constructing a mesh from a live sensor stream is often desirable. One way to make surface reconstruction efficient enough to run in real-time is to use incremental updates. TSDF-based surface reconstruction is particularly popular in practice given its inherently incremental nature and general simplicity. This method falls under the second family of approaches and estimates the implicit surface by averaging projective distances. Since the cost of updating the TSDF, or implicit surface in general, overshadows the cost of the mesh extraction, real-time methods primarily focus on optimizing the former.

5.4.4 Voxels

Voxel-based methods are among the most commonly used volumetric representations in 3D reconstruction and robotics. Instead of covering a swath of existing literature chronologically, this section will focus on concepts commonly encountered in practice and organize them according to three fundamental decision criteria: the chosen estimated quantity, data structure, and scalability considerations.

5.4.4.1 Methods by their Estimated Quantity

The first choice in a voxel-based mapping framework is which quantity to estimate, with the most common options being *occupancy* (see Section 5.2.1) or a *distance* metric (see Section 5.2.2). The previous discussion in Section 5.2.3 can be used to decide between the two.

Since the introduction of the original continuous probabilistic occupancy measurement model for sonar [780], simplified piecewise-constant models have been developed to reduce computational costs [486]. This shift was influenced by the advent of LiDAR technology and the growing interest in transitioning from 2D to 3D maps. More recently, Loop et al. [699] presented a continuous probabilistic model that, instead of inflating objects, converges to an occupancy probability of 0.5 along objects' surfaces. Occupancy estimation, popular for collision avoidance due to its superior recall, is limited by its discontinuous nature and uninformative gradients compared to distance-based methods (see Section 5.2.3).

For distance metrics, we must not only estimate the positive part of the distance field but also extrapolate negative distances behind the surface since the surface is represented by the signed distance field's zero-crossings. To limit the accuracy impact of fusing imperfect positive and negative distances estimates (see Section 5.2.2), the updates are typically clamped to a small *truncation band* around the surface boundary. However, distance-based methods remain prone to erasing geometry. For example, when thin objects are observed from opposing sides, averaging the observed positive and hallucinated negative distances makes the zero-crossings flip around or disappear. Some works have analyzed the effect of the truncation band and weight drop-offs on the quality of the final reconstruction [140]. Fundamentally, the problem can be reduced but not eliminated. Overall, the surfaces estimated by TSDFs outperform occupancy methods along smooth surfaces at the cost of lower *recall* on thin objects.

The distance information provided by TSDFs is inherently valuable. However, instead of being conservative, TSDFs strictly overestimate the *Euclidean* distance. To address this safety concern, voxblox [828] popularized incrementally building ESDFs. Voxblox fuses the sensor data into a TSDF and then updates its ESDF using a brushfire algorithm [636]. Subsequently, FIESTA [433] proposed a hybrid approach that incrementally updates an ESDF map from an occupancy map instead.

5.4.4.2 Methods by Data Structure

The simplest data structure for volumetric mapping is a static 3D array. As shown by KinectFusion [807], this data structure yields good results for small and fixed-size scenes. However, many applications require the ability to dynamically expand the map at runtime, while only allocating voxels where needed to save memory.

To address these concerns, Niessner proposed a voxel-block hashing scheme [814], which groups the voxels into blocks (e.g., $8 \times 8 \times 8$ voxels) that are stored in a hash-map. This data structure was quickly adopted for TSDFs, providing constant-time ($\mathcal{O}(1)$) lookups and dynamic insertions. Of course, it can also be used to store occupancy probabilities, as shown by FIESTA [433]. Compared to hashing voxels individually, grouping them in blocks offers an adjustable trade-off between the hash table's size and the granularity at which voxels are allocated.

Naturally, voxels can also be stored using tree structures. Octomap [486] first popularized using an *octree* to store occupancy probabilities and has been the *de facto* standard for volumetric mapping for many years. A significant advantage of using trees is that they inherently support multi-resolution, while a major limitation is that encoding the tree's structure introduces a significant memory overhead, and that the cell lookup time is proportional to the tree's height. Most recent approaches address this limitation by leveraging hybrid data structures. Supereight [1129], for example, proposes to use a standard (dynamic) octree for the first levels and static octrees for the last few levels. These static octrees can be seen as octrees stored using a fixed-sized array. This removes the memory overhead of encoding parent-child relationships with pointers, at the cost of reducing granularity since static octrees are allocated as a block. The VDB [798] data structure was first introduced for the visual effects (VFX) industry and subsequently used by several volumetric mapping frameworks [729, 1136]. As discussed in Section 5.3.3.4, it combines block-hashing with trees to obtain the best of both worlds: good memory efficiency, hash-like constant time lookups and insertions, and tree-like multi-resolution.

A practical consideration is that downstream tasks often demand storing additional information, such as colors, semantics [407, 947] or an ESDF [828, 433] alongside the occupancy probabilities or TSDF. Although virtually any data structure can be extended to support additional channels, the required implementation effort scales with how complicated the underlying data structure is. This further motivates using simple data structures (e.g. voxel-block hashing) or flexible, third-party libraries.

5.4.4.3 Methods by Measurement Integration Algorithm

The algorithm used to update the map based on depth measurements is referred to as the measurement integrator. It updates the estimated quantity for each observed voxel by applying the measurement model. The two main approaches used to integrate measurements are ray-tracing and projection-based methods.

For each measured point, ray tracing integrators cast a ray from the sensor to the point and update all the voxels intersected by the ray. An advantage of this approach is that it is very general, and only requires that the position of the sensor's origin is known. However, voxels may be hit by multiple rays, especially if they are near the sensor. This leads to duplicated efforts, and handling the resulting race conditions in parallel implementations creates implementation and performance overheads.

In contrast, projection-based methods directly iterate over the observed voxels and look up the ray(s) needed to compute their update by projecting each voxel into sensor coordinates. Iterating over the map instead of the rays inherently avoids race conditions. Projection-based methods are, therefore, prevalent in multi-threaded and GPU-accelerated volumetric mapping frameworks. The predictable access pattern resulting from directly iterating over the map also reduces memory bottlenecks. Yet, a major disadvantage is the need for explicit knowledge of the sensor's full pose and projection model. This method is also harder to use with disorganized point clouds, including the clouds obtained after applying LiDAR motion-undistortion.

5.4.4.4 Methods by Scalability

Memory and computational costs are two of the main bottlenecks in volumetric mapping. For fixed-resolution methods, the memory and computational complexities grow linearly with the map's total volume and cubically with the chosen resolution. Reducing these complexities is of significant research interest, as it is necessary to create detailed maps that scale beyond small, restricted volumes.

Early works in volumetric mapping mainly focused on reducing memory usage. For example, Octomap [486] proposes to use its octree's inner nodes to store their children's max or average occupancy. By recursively pruning out leaf nodes whose estimated quantities are close to their parent, constant areas in the map are automatically represented with fewer, lower resolution nodes. This adaptation to the environment's geometry is very effective in practice since environments predominantly consist of free space. Furthermore, storing min, max, or average values in the octree's inner nodes could be valuable for downstream tasks, as it enables map queries at lower resolutions and the use of hierarchical algorithms for tasks such as fast collision checking or exploration planning. Yet, a core limitation of Octomap is that it integrates all measurements at the highest resolution, meaning that the scaling of its computational complexity remains cubic.

Multi-resolution can also be leveraged to reduce the computational cost of measurement updates. Given that measurement rays are emitted at fixed angles, resulting in fewer rays hitting distant geometry, it seems logical to lower the update resolution as the distance increases. This can be achieved through multi-resolution ray-tracing [294] or multi-resolution projective integration [1129]. Supereight2 [355] reduces the computational complexity further by adjusting the update resolution to the entropy of the measurement updates. Such methods significantly enhance the update performance, yet a remaining challenge is that the map's different resolution

levels still have to be synchronized explicitly. One way to eliminate this synchronization requirement is to encode only the differences between each resolution level, instead of storing absolute values in each octree node. This can formally be done by applying wavelet decomposition. Wavelet-encoded maps can efficiently be queried at any resolution at any time. Using this property, wavemap [929] reduces the computational complexity even further by updating the map in a coarse-to-fine manner. In addition to adjusting the update resolution to the measurement entropy, it also skips uninformative updates, such as when the occupancy for an area in the map has converged to being free, and all measurements agree.

5.4.5 GPs

As mentioned in Section 5.3.2.5, formulating the mapping problem as a regression problem is desired to obtain a continuous representation. Moreover, if the aim is to limit the number of assumptions about the environment, solving a non-linear regression problem with non-parametric methods is ideal. GP [916] is a stochastic, non-parametric, non-linear regression approach. It allows estimating the value of an unknown function at an arbitrary query point given noisy and sparse measurements at other points. We already learned in Chapter 2.2 how GP can be used for continuous time trajectory representation. As will be apparent, GP are also an appealing solution for mapping continuous quantities, and they have been extensively used in the robotics literature to model continuously spatial phenomena with depth sensors [1126, 822, 385, 582].

The information in GP models is contained in its mean $\boldsymbol{m}(\boldsymbol{x})$ and kernel functions $\boldsymbol{\mathcal{K}}(\boldsymbol{x}, \boldsymbol{x}')$ and model the estimated continuous quantity as

$$f(\boldsymbol{x}) \sim \mathcal{GP}\left(\boldsymbol{m}(\boldsymbol{x}), \boldsymbol{\mathcal{K}}(\boldsymbol{x}, \boldsymbol{x}')\right). \tag{5.8}$$

Let $\mathsf{X} = \{\boldsymbol{x}_j \in \mathbb{R}^D\}$ be a set of locations with measurements $\boldsymbol{y}$, with $y_j = f(\boldsymbol{x}_j) + \epsilon_j$ of the estimated quantity taken at the locations $\boldsymbol{x}_j$. For J number of training pair $(\boldsymbol{x}_j, y_j)$, we assume the noise ϵ_j to be *i.i.d* following Gaussian $\epsilon_j \sim \mathcal{N}(\mathbf{0}, \sigma_j^2)$. Given a set of testing locations $\mathsf{X}^* = \{\boldsymbol{x}_n^* \in \mathbb{R}^D \mid n = 0, \ldots, N\}$, we can express the joint distribution of the function values and the observed target values as,

$$\begin{bmatrix} \boldsymbol{y} \\ \boldsymbol{f}_* \end{bmatrix} = \mathcal{N}\left(\mathbf{1}, \begin{bmatrix} \boldsymbol{K}(\mathsf{X}, \mathsf{X}) + \sigma_j^2 \mathbf{I} & \boldsymbol{K}(\mathsf{X}, \mathsf{X}^*) \\ \boldsymbol{K}(\mathsf{X}^*, \mathsf{X}) & \boldsymbol{K}(\mathsf{X}^*, \mathsf{X}^*) \end{bmatrix}\right), \tag{5.9}$$

where $\boldsymbol{K} = \left[\boldsymbol{\mathcal{K}}(\boldsymbol{x}_i, \boldsymbol{x}_j)\right]_{ij}$. Thus the conditional distribution of $(\boldsymbol{f}_* \mid \mathsf{X}, \boldsymbol{y}, \mathsf{X}^*) \sim \mathcal{N}\left(\overline{\boldsymbol{f}}_*, \operatorname{cov}(\boldsymbol{f}_*)\right)$, with the mean equation is given by,

$$\overline{\boldsymbol{f}_*} = \boldsymbol{K}(\mathsf{X}^*, \mathsf{X})\left[\boldsymbol{K}(\mathsf{X}, \mathsf{X}) + \sigma_j^2 \mathbf{I}\right]^{-1} \boldsymbol{y}, \tag{5.10}$$

and the covariance equation is,

$$\text{cov}\left(\boldsymbol{f}_*\right) = \boldsymbol{K}\left(\mathsf{X}^*, \mathsf{X}^*\right) - \boldsymbol{K}\left(\mathsf{X}^*, \mathsf{X}\right)\left[\boldsymbol{K}(\mathsf{X}, \mathsf{X}) + \sigma_j^2 \mathbf{I}\right]^{-1} \boldsymbol{K}\left(\mathsf{X}, \mathsf{X}^*\right). \tag{5.11}$$

Here, (5.10) and (5.11) are the predictive equations for the estimated quantitative.

GPs have proven particularly powerful to represent spatially correlated data, hence overcoming the traditional assumption of independence between cells, characteristic of the occupancy grid method for mapping environments. Gaussian Process Occupancy Map (GPOM)s [822] collects sensor observations and the corresponding labels (free or occupied) as training data; the map cells comprise testing locations, which are related to the training data as shown in (5.9). After the regression is performed using (5.10) and (5.11), the cell's probability of occupancy is obtained by "squashing" regression outputs into occupancy probabilities using binary classification functions.

In its original formulation GPOM is a batch mapping technique with cubic computational complexity ($\mathcal{O}(J^3 + J^2 N)$). Approaches that aim to tacking this computational complexity especially for incremental GP map building have been proposed following this work, for example [582, 583, 1155, 386].

A key advantage of mapping with GP-based functions is that the estimated quantity can be linearly operated [974] and still produce a GP as an output. Given that derivatives and, therefore gradients, are linear operations, the differentiation output of the estimated quantity is probabilistic. A continuous representation of the uncertainty in the environment can be used to highlight unexplored regions and optimize a robot's search plan [386, 684]. The continuity property of the GP map can improve the flexibility of a planner by inferring directly on collected sensor data without being limited by the resolution of a grid/voxel cell.

5.4.5.1 Gaussian Process Implicit Surface

Implicit surfaces can also be represented by a GP. Gaussian process implicit surface (GPIS) techniques [1183, 743, 684, 513] use a GP approach to estimate a probabilistic and continuous representation of the implicit surface given noisy measurements. Furthermore, GPIS can be also used to estimate not only the surface but also the distance field in a continuous manner [584, 1040, 643, 1192].

In the GPIS formulation, let us consider the distance field $\boldsymbol{d}$ to be estimated from the distance to the nearest surface d_i given the the points on the surface and its corresponding gradient $\nabla \boldsymbol{d}$ computed through linear operators [974]. Then $\boldsymbol{d}$ with $\nabla \boldsymbol{d}$ can be modelled by the joint GP with zero mean (given that at the surface the distance is zero):

$$\begin{bmatrix} \boldsymbol{d} \\ \nabla \boldsymbol{d} \end{bmatrix} \sim \mathcal{GP}(\mathbf{0}, \boldsymbol{K}(\mathsf{X}, \mathsf{X}')). \tag{5.12}$$

GPIS approaches have the ability to estimate a continuous implicit surface and the normal of the surface through the gradient, both with uncertainty. Some works

have considered the use of parametric function priors to capture given shapes more accurately [743, 513]. Other approaches aimed to estimate not only the implicit surface but the full distance field. Given the nature of the vanilla version of the GPIS formulation, the distance is well approximated near the measurements, *i.e.,* on the surface, but falls back to the mean, which in this case is zero, faraway from the surface. To estimate the full distance field in a continuous and probabilistic formulation further away from the surface, works have considered applying a non-linear operation to a GPIS-like formulation [1192, 1193, 641].

All these works have to deal with the computational complexity of the GP-based formulation, but as an exchange, a continuous, generative, and probabilistic representation of the environment, given only point clouds can be achieved.

5.4.6 Hilbert Maps

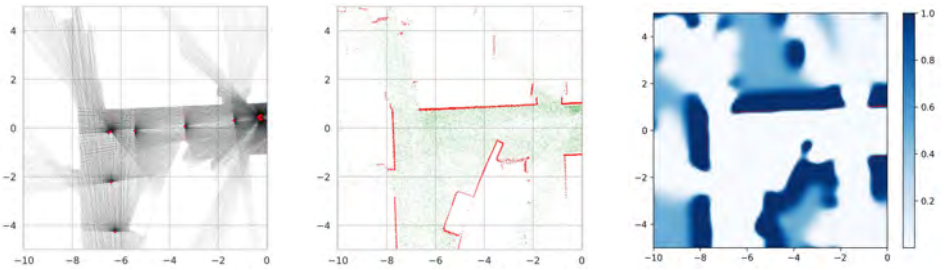
Hilbert Map (HM)s [911] are in many ways similar to GPOMs [822]. Both are continuous probabilistic models that do not discretize the space, unlike voxel-based methods, and in contrast to point-based methods are capable of interpolating missing data. As stated, the major challenge in GPOM is high computational expense. Thus the design goals of Hilbert maps were the following: (i) process data continuously in an online manner, (ii) model dependence between observations, and (iii) incorporate measurement uncertainty.

To achieve these goals, training a logistic regressor with stochastic gradient descent in a projected feature space is often leveraged. The classifier and optimizer combination enables online model updates using large amounts of data while the feature projection permits representing intricate spatial details with such a simple classifier.

The feature projection serves the same idea as the kernel in a GP, but instead of a full covariance we use an approximation. There are many options for this, including Nystroem [1182], Random Fourier Features [908], and Sparse Kernel [762], which is what we will be using. To goal of the sparse kernel is to limit the range at which observations have an influence which improves convergence and computational efficiency. The outcome is a kernel that drops to exactly 0 at a specific distance.

This kernel allows us to project points in 2D or 3D space into significantly higher dimensions by placing inducing kernels at regular intervals over the space to be mapped. Furthermore, this enables computing high-dimensional feature space representations of input data to be trained the logistic regression classifier using mini-batch stochastic gradient descent. Lastly, training is done by sampling free space points along the range measurement, while adding the return as an obstacle point.

One challenge faced by HMs is the expressivity of the used kernel. A radial basis function (RBF), as used in Figure 5.11, is a circle or a sphere and their values need to be combined to reconstruct intricate details of the environment. Therefore there



(a) (b) (c)

Figure 5.11 (a) The observations by a robot using a 2D LiDAR sensor, which is turned into a dataset (b) Free (green) and occupied (red) points. These are then used to train a Hilbert Map as seen in (c).

is a tradeoff in the form of number of kernels and their lengthscale affecting the computational cost, reconstruction detail, and interpolation ability.

5.4.7 Deep Learning in Mapping

With the recent interest of the computer vision and robotics community in novel view synthesis using neural radiance fields (NeRFs) [770], which provided compelling results for image generation via a simple multi-layer perceptron (MLP), several approaches investigated the usage of neural representations to estimate a SDF. Learning to predict a SDF at arbitrary spatial location leads to a continuous representation that can be turned into meshes at arbitrary resolutions, but also can lead to more complete representation due to the interpolation capabilities of the learned function.

Similar to implict representations, the representation is learned from input data and approximated to provide a continuous function that can be queried at arbitary locations. While often these neural representations are learned offline with given poses, there has been recently also an interest in incremental approaches [1052, 1288] and approaches that estimate poses on-the-fly using a neural representation [1052, 973].

In particular, the approach of Sucar *et al.* [1052] uses a neural network to predict the SDF value of an arbitrary point in the scene based on RGB-D frames. Follow-up approaches extended this approach by separating the spatial representation of the features via voxel grids [869, 1304], octrees [1288], points [973, 269], etc. from the neural representation. In these approaches, small but descriptive features are stored in a spatial representation and used to determine with a small, neural network the SDF value of an arbitrary point in the scene. This allows to decouple the learned function from the spatial representation, which makes it possible to rely on small

neural decoders to turn features into signed distance values, but also being effective for large-scale scenes, such as outdoor environments.

The area of deep learning-based mapping, reconstruction, and SLAM is currently rapidly evolving and integrating ideas from classical representation, such as surfel splatting [556, 747], to achieve remarkable results in terms of reconstruction quality, but also capabilities. In particular, the ability to render novel views and generate new data at arbitary positions could be potentially exploited for robot learning without relying on simulated environments. For example, NeRF and Gaussian splatting [1313, 109] have gained considerable popularity, demonstrating significant potential in various SLAM-related works. These will be further detailed in Chapter 14.

5.5 Usage Considerations

All map representations trade off distinctive, often complementary, strengths and weaknesses. When choosing a map representation for a given application or robotic system, it is therefore important to carefully consider how the map will be used in all downstream tasks. Further factors to consider are the operating environment and available sensors. We will start by discussing environmental factors, which motivate several clear-cut choices, followed by more nuanced task-dependent considerations. Finally, we conclude this chapter with a brief discussion on usage considerations related to the existing methods presented in Section 5.4.

5.5.1 Environmental Aspects

Operating environments can be categorized as either *structured* or *unstructured*. In tightly controlled spaces, such as automated factories, custom map representations – tailored to the robot's task and specific objects it will encounter – typically outperform general dense representations in terms of efficiency and accuracy. In contrast, the dense representations covered in this chapter can model objects of arbitrary shapes and work in any environment. When operating in changing or partially unknown environments, it is often important for robots to be able to distinguish observed free space from unobserved space. This information allows path planners to avoid unsafe motions through unobserved space, which could be occupied, and can also be used for exploration planning. Explicit surface representations, including points, surfels, and meshes, generally cannot distinguish between free and unobserved space, while all occupancy-based methods do. Implicit surface-based methods can also provide this distinction, though very often for more reconstruction-focused applications, this information is discarded farther from the surface to save computational and memory costs.

Another consideration is *scalability*. Explicit representations tend to be more memory efficient than implicit representations, as they only describe the surface

itself and their fidelity can easily be adapted to the detail required for each part of the scene. When free-space information is required, multi-resolution approaches can offer significant improvements over fixed-resolution voxelized representations in terms of accuracy, memory, and their ability to capture very thin objects.

One final consideration is whether the environment has a significant amount of *dynamic objects* and the degree to which these should be modeled. From the perspective of map representations, most existing approaches can be grouped into one of three categories. The first set of approaches does not consider dynamics and directly fuses all measurements into one of the representations introduced in this chapter. In practice, this might already suffice when using implicit representations, since their free-space updates typically do a good job at erasing leftovers of objects after they moved. The second category of approaches tries to only integrate the environment's static elements into the map, by explicitly detecting and discarding all measurements corresponding to dynamic objects. This approach is particularly popular when using explicit maps, where leftovers are more tedious to remove, and generally makes it possible to generate clean maps even in highly dynamic spaces. The last set of solutions not only represents the background but also the moving elements in the scene. Note that this is commonly done using hybrid representations, mixing fundamental geometric representations introduced in this chapter with bespoke representations at the object level. For a detailed discussion on SLAM in dynamic environments, including concrete methods to implement the above and more advanced approaches, we refer the reader to chapter 15.

5.5.2 Downstream Task Types

In addition to the environment, it is equally important to consider what map information is necessary for the robot's required tasks. While any given operation can typically be performed on all representations, the efficiency and implementation complexity tend to vary greatly. The biggest difference lies in whether the operation is performed along the surface or in Cartesian space. As shown in Table 5.2, implicit representations generally allow for simple, efficient filtering of properties that are expressed in Cartesian coordinates, such as occupancy. In contrast, explicit representations are well suited to filter properties that are expressed along the surface, such as visual textures. This explains why explicit representations are generally more sensitive to the quality of the depth measurements, but can create very detailed, visually appealing 3D reconstructions. On the other hand, implicit methods are well suited for fusing noisy depth measurements, such as RGB-D camera data.

In terms of queries, explicit representations make it possible to directly iterate over the surface. This explains their popularity in rendering and graphics applications, and for tasks such as coverage path planning. However, they require additional steps, such as nearest neighbor lookups, to answer queries in Cartesian coor-

	Efficient in	
Operation	Explicit representation	Implicit representation
Filter measurements	Along the surface (texture,...)	In Cartesian space (occupancy,...)
Query and iterate	In surface coordinates (coverage planning,...)	In Cartesian coordinates (collision checking,...)
Modify surface	Geometry (deformation,...)	Topology (merge, cut, simplify,...)

Table 5.2 *Complementary strengths and weaknesses of explicit and implicit surface representations.*

dinates. The exact opposite is true for implicit representations, which are therefore commonly used for collision checking tasks.

Finally, explicit representations allow for efficient modifications of the surface's geometry, including deformations. In practice, maps are often constructed by integrating depth measurements using pose estimates from an imperfect, drifting odometry system. Over time, the accumulated pose errors also lead to inconsistencies in the dense map. Just like in SLAM systems, these errors can be eliminated by deforming the dense map when detecting loop closures. Although both explicit and implicit surfaces can be deformed, this operation is inherently simpler and more efficient when using an explicit representation. In contrast, using an implicit representation simplifies and improves the efficiency of operations affecting the surface's topology, or connectivity. Implicit representations are therefore often used to merge surface estimates, combine or subtract object shapes, and simplify surfaces.

It is important to remember that different representations can also be used in tandem to leverage their respective strengths. One good example of a hybrid approach is TSDF-based meshing (Section 5.4.3), where noisy depth measurements are first conveniently filtered using an implicit surface representation (TSDF) which is then converted to an explicit representation (mesh) using Marching Cubes. When deciding whether the advantages of hybrid representations outweigh the overhead they introduce, it is worth considering how the conversions can be limited to only happen locally and infrequently.

5.5.3 Summary of Mapping Methods

We now conclude our discussion by summarizing the key differences between the existing methods presented in this chapter. Starting with the explicit representations, using a collection of points to describe the surface is simple and requires the fewest assumptions, but it is also the least informative. Beyond infinitesimal points, surfels represent the surface's properties over small neighborhoods, or patches. Finally, meshes explicitly represent the surface's connectivity and allow its properties

to smoothly be interpolated. However, estimating the surface's connectivity requires the most assumptions and comes at a significant computational cost.

In terms of implicit representations, a particular advantage of implicit surfaces over occupancy maps is that they offer fast, high-quality distance information and gradients which are beneficial for optimization-based planning. However, filtering occupancy estimates requires less assumptions and, for voxel-based methods, occupancy maps are better at capturing thin obstacles. In cases with particularly noisy or sparse depth measurements, non-voxelized implicit representations, based on GPs and Hilbert Maps, provide particularly good uncertainty estimates. As they explicitly consider the geometry's spatial correlations, they are generally also better at interpolating partially observed surfaces.

One rapidly advancing research area is that of learning-based methods. In terms of learning-based implicit representations, NeRFs have been shown to enable promising new capabilities, particularly for semantic modeling and spatial reasoning. More recently, Gaussian splatting [560] – an explicit learning-based representation bearing similarities to surfels – lead to an increasing interest into approaches using splatting [556, 747, 1256]. Researchers are actively working on improving the computational and memory footprint of these approaches, testing what new skills they can enable, and exploring how they can be integrated into complete robotic systems. Looking ahead, we suspect that learning-based methods can increase the generality and expressiveness of dense representation, while improving their ability to handle noisy measurements, incomplete observations and dynamic objects through learned priors.

Acknowledgment

The authors thank Lan Wu for her support in preparing this chapter.