# Robot Motion Planning

## 1 Introduction

The main problem is to find a path from one position or configuration to another, without colliding with the environment. Planning algorithms can be rated regarding different aspects.

Performance metrics for path planning algorithms:

- path length
- execution speed
- planning time
- safety (distance to obstacles)
- robustness to disturbances
- probability of success

To avoid long calculations in online processes, often an offline pre-processing is done and a road-map is calculated. When the manipulator is high-complex it is not possible to calculate the road-map beforehand and thus probabilistic approaches are used. Those perform sampling and create the road-maps with uncertain data.

## 2 Configuration Space

### 2.1 Basics

**Workspace:** Reachable space of the end-effector.
**Task space:** Corresponds to the space where the robot's task can be naturally expressed.
**Configuration:** The smallest number of real-valued coordinates to fully describe configurations of a robot. For example, a mobile robot moves in directions $x, y$ and can turn its yaw heading $\theta$ - a configuration of this robot could be $\boldsymbol{c} = (1, 3, 30°)^{\mathrm{T}}$.
**Configuration space (*C-space*):** The space of all possible configurations, that a robot can have in a certain workspace.
**Dimension of the C-space/DoF:** Minimum number of parameters needed to specify the configuration. Because there can also be angles in the topology of the C-space, it is usually not a Cartesian space (e.g. $2\pi = 4\pi$, cyclic repetition along angle axis).
**Common robot models:** Single points, single rigid bodies and multiple rigid bodies with joints.
**Non-holonomic robot:** Robot, that cannot move freely on the configuration space manifold because it is constrained in some way. For example, a car cannot drive sideways.
**Path:** Continuous curve in configuration space $C$ connecting two configurations $q$ and $q'$: $\tau : s \in [0, 1] \rightarrow \tau(s) \in C$ s.t. $\tau(0) = q$ and $\tau(1) = q'$
**Trajectory:** A trajectory is a path parametrized by time: $\tau : t \in [0, T] \rightarrow \tau(t) \in C$. Possible constraints: Smoothness, minimum length, minimum time, ...

### 2.2 Calculate DoF

**Calculate DoF of a single body:** DoF = total DoF of 3 fixed points - number of constraints (of the 3 fixed points, e.g. distances between points).
*Alternative:* Find DoF of one point, go the next one and apply constraints, go to the third point and apply constraints.

**Calculate DoF for general robots:**

- Open chains: Add the DoF of each joint
- Closed chains:
  $DoF = N(k-1) - \sum_{i=1}^{n}(N - f_i) = N(k-n-1) + \sum_{i=1}^{n} f_i$
  – $N = 6$ for 3D, $N = 3$ for 2D
  – $k =$ number of links (including ground link)
  – $n =$ number of joints
  – $f_i =$ DoF of the $i^{th}$ joint

### 2.3 Parametrizations of configurations

Configuration: $\boldsymbol{q} = (position, orientation) = (x, y, z, \cdot)$. There exist different ways how to parametrize orientations:

- **Parametrization of orientations by rotation matrix:**
  $R = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}$ with $\det(R) = 1$ and $R^T R = I$ (or $RR^T = I$)
  Columns of $R$: Coordinate axes of new coordinate System
- **Parametrization of orientations by Euler angles:**
  Different sequences of rotation axes exist, e.g. "Roll, Pitch, Yaw" around $x$, $y$ and $z$:

  $$R_{xyz}(\alpha, \beta, \gamma) = R_z(\gamma) R_y(\beta) R_x(\alpha) = \\ = \begin{pmatrix} c\gamma & -s\gamma & 0 \\ s\gamma & c\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} c\beta & 0 & s\beta \\ 0 & 1 & 0 \\ -s\beta & 0 & c\beta \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & c\alpha & -s\alpha \\ 0 & s\alpha & c\alpha \end{pmatrix}$$

  Problem: Singularities
- **Parametrization of orientations by unit quaternions:**
  $\boldsymbol{u} = (cos(\theta/2), n_x sin(\theta/2), n_y sin(\theta/2), n_z sin(\theta/2))$
  with $\|\boldsymbol{n}\| = 1$ ($\boldsymbol{n}$: rotation vector, $\theta$: rotation angle)
  Advantages: Compact, no singularities, natural orientation representation

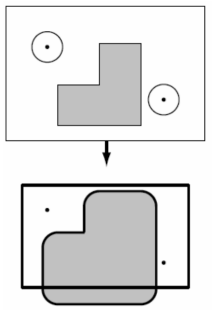
### 2.4 Configuration space

**C-space $C$:** $F \cup C_{obs}$
**Free space $F$:** All collision-free configurations (open subset of $C$).
**Obstacle space $C_{obs}$:** Set of configurations which lead to a collision with workspace obstacles (closed subset of $C$).
**Semi-free space:** A configuration $q$ is semi-free if the robot placed $q$ touches the boundary, but not the interior of obstacles. $F \cup \partial C_{obs}$
**Determine free space:** Discretize the configuration space and check for each grid cell if the corresponding configuration leads to a collision in the workspace (brute-force approach).


TU München - Fakultät für Informatik - Robotics, Cognition, Intelligence

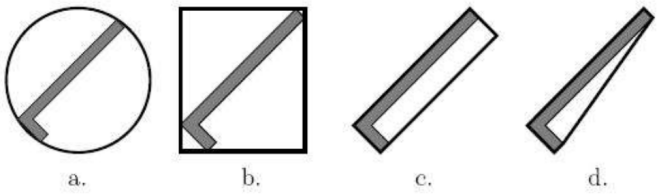
### 2.5 Point representation

In order to determine the C-space / free space of configurations, a point representation can be useful. For example, if we have a circular robot with radius $r$ and some obstacles in the workspace, the robot can be reduced to a point if the obstacle boundaries are "blown up" by the radius of the robot. In other words, to get a smaller representation of the robot without changing the properties of the workspace, the obstacles have to get bigger. $\Rightarrow$ Minkowski sum algorithm



### 2.6 Collision Detection

Create a hull (circle, rectangle, convex hull, ...) around an object and perform a collision check. Circle: 1 distance calculation. Rectangle (axis aligned): 4 distance calculations.
Circle approach: Start with one big circle, perform check: if collision, split circle into smaller circles and repeat until minimum radius is reached, else no collision.




## 3 Bug Algorithms

Bug algorithms are simple strategies that use local knowledge (e.g. from sensor input) instead of global knowledge.

**Assumptions:**

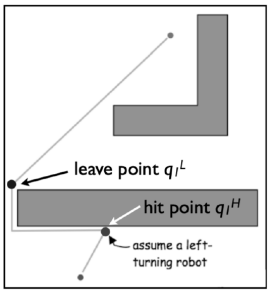
- Robot is a point in a plane.
- Contact sensor to detect obstacle boundaries.

**Basic idea:** Move on a straight line towards goal and follow the boundaries around obstacles.

### 3.1 Bug 0

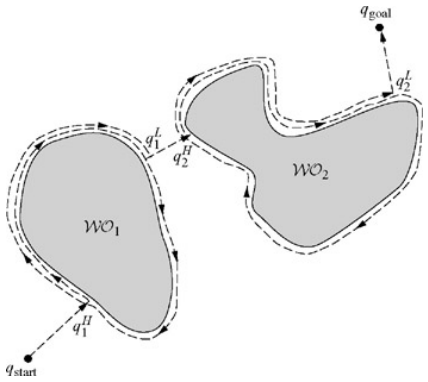
1. Head towards the goal.
2. When hit point set, follow the wall until you can move towards goal again (leave point).
3. Continue from 1.

Fails for certain obstacle shapes! (e.g. spiral)




### 3.2 Bug 1

1. Move towards goal on the *m-line* connecting $q_{start}$ and $q_{goal}$.
2. If an obstacle is detected at a hit-point $q_i^H$, move left (or right) along its boundary until you return to $q_i^H$ again.
3. While circumnavigating the obstacle, calculate and save the distance to the goal at any coordinate.
4. Determine the closest point to the goal on the boundary and use it as a leave point $q_i^L$.
5. From $q_i^L$, go straight to the goal again on the m-line $q_i^L$ and $q_{goal}$.
6. If the line that connects $q_i^L$ and the goal intersects the current obstacle, then there is no path to the goal.

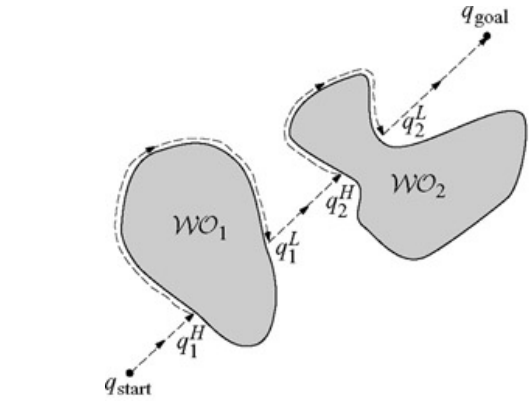



*Worst case:* $L_{Bug1} \leq d(q_{start}, q) + 1.5 \sum_{i=1}^{n} p_i$ with $p_i$ being the length of the contour of the $i$-th obstacle.
*Best case:* $d(q_{start}, q)$.
*Problems:* Fails when started inside of closed obstacles and possibly changed approach vector to the goal (problematic when grabbing objects).

### 3.3 Bug 2

The m-line of Bug 2 does not change and it connects $q_{start}$ and $q_{goal}$.

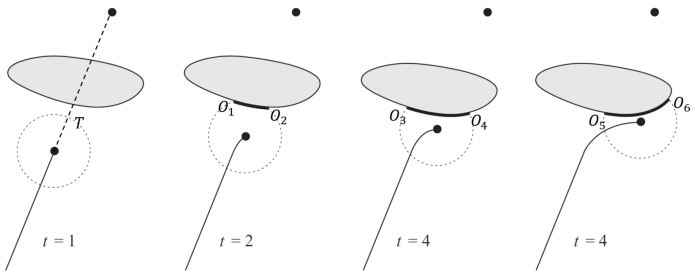
1. Move towards goal on the *m-line* connecting $q_{start}$ and $q_{goal}$.
2. If an obstacle is detected at a hit-point $q_i^H$, move left (or right) along its boundary.
3. If a point on the m-line is found and it is closer to the goal than $q_i^H$, use it as a leave point $q_i^L$.
4. From $q_i^L$, go straight to the goal again on the m-line.
5. If the robot re-encounters the original departure point $q_i^H$ from the m-line, then there is no path to the goal.




Fails when started inside of closed obstacles. Problematic shapes: spirals. Worst case: $L_{Bug2} \leq d(q_{start}, q) + 0.5 \sum_{i=1}^{n} n_i p_i$ with $n_i$ being the number of times that the m-line hits the obstacle.

### 3.4 Tangential Bug

The robot is equipped with a (finite-range) radial distance sensor. If an obstacle is encountered within the range of the sensor, the robot changes its direction tangentially to the border of the obstacle.




Essentially, the bug algorithms have two behaviors: Drive toward a point and follow an obstacle.

**Algorithm 3** Tangent Bug Algorithm

**Input:** A point robot with a range sensor

**Output:** A path to the $q_{goal}$ or a conclusion no such path exists

1: **while** True **do**
2: **repeat**
3: Continuously move toward the point $n \in \{T, O_i\}$ which minimizes $d(x, n) + d(n, q_{goal})$
4: **until**
- the goal is encountered **or**
- The direction that minimizes $d(x, n) + d(n, q_{goal})$ begins to increase $d(x, q_{goal})$, i.e., the robot detects a "local minimum" of $d(\cdot, q_{goal})$.

5: Chose a boundary following direction which continues in the same direction as the most recent motion-to-goal direction.
6: **repeat**
7: Continuously update $d_{reach}$, $d_{followed}$, and $\{O_i\}$.
8: Continuously moves toward $n \in \{O_i\}$ that is in the chosen boundary direction.
9: **until**
- The goal is reached.
- The robot completes a cycle around the obstacle in which case the goal cannot be achieved.
- $d_{reach} < d_{followed}$

10: **end while**

$d_{followed}$ is the shortest distance between the boundary which had been sensed and the goal. $d_{reach}$ is the distance between the goal and the closest point on the followed obstacle that is within line of sight of the robot.

## 4 Planners

In order to describe more complicated path planners, we need to be able to specify the position of the robot and the (C-)space it occupies. Gradients work by changing the repulsive potential as a robot gets closer to an obstacle. The start point is given a medium gradient potential, the end point is given a low gradient potential, and the obstacles are given high gradient potentials. All the robot needs to do is "roll down the hill."

### 4.1 Brushfire Planner

The Brushfire Algorithm is a discrete version of the aforementioned gradient algorithm and involves the use of a grid to determine the potential of cells. It starts the fire at all boundaries of obstacles and overall boundary and iterates through the grid. "2" for cells neighboring an obstacle and so on. When two weights meet, same distance to obstacles is achieved (important for Voronoi).

### 4.2 Wave-Front Planner

Variation of the Brushfire algorithm. Using a grid it assigns a "1" to each cell that has an obstacle (or part of an obstacle). The start point is labeled "2" and the "wave" propagates from that point. Each adjacent cell, if empty, is given an incrementally higher number until all the cells have a number. If the goal has a number in its cell, the goal is reachable in that many moves minus one. The algorithm can also start from the goal towards the start.
*Benefit:* Avoids the local minima problem by planning one step at a time on a grid.

## 5 Classical path planning

Using offline pre-processing and exact environment knowledge in order to do the path planning beforehand and save online performance, we distinguish:

- Roadmaps
- Cell decomposition
- Potential field

### 5.1 Roadmaps (with exact knowledge)

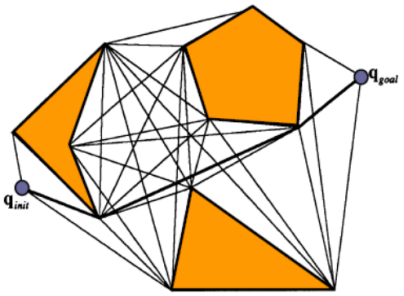
*Idea:* Represent the connectivity of the free space in a network of 1D curves.

Using this network, paths can be calculated using graph-search algorithms, for example.
*Advantage:* Roadmap has to be constructed once if the topology of the workspace (i.e. obstacle positions) does not change.
*Disadvantage:* Roadmaps can be not very efficient in dynamic environments.

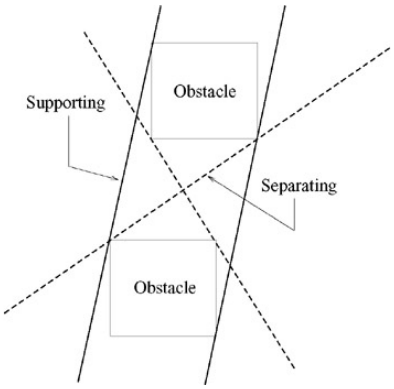
#### 5.1.1 Visibility Graph

The visibility graph applies to 2D spaces with polygonal obstacles.
Nodes: $q_{init}$ (start) and $q_{goal}$ (goal) and obstacle vertices.
Edges: Two nodes are connected with an edge if the connecting

line is an obstacle edge or does not intersect with an obstacle. Time $O(n^3)$, space $O(n^2)$.

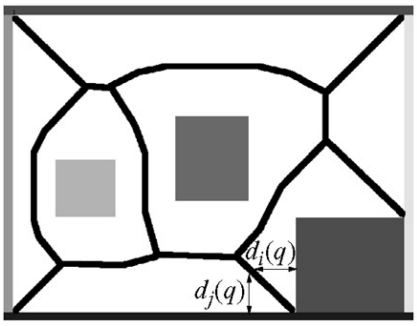



The visibility graph can contain useless edges. This can be resolved by constructing the reduced visibility graph, where only edges are kept that are separating or supporting lines between two obstacles (or lines on the border of obstacles).




### 5.1.2 Voronoi Diagram

The goal of the Voronoi diagram is to find paths that maximize the distance to the obstacles. Therefore, the roadmap lines can be curved.
Time $O(n \log n)$, space $O(n)$.




To construct the Voronoi diagram, one can use the Brushfire algorithm, which calculates a distance map on a grid:

- Grid initialization: free space = 0, obstacle space = 1
- For each point in the grid, it assigns the distance to the closest obstacle point.
- Every point, where the distance to two (or more) different obstacles is the same, lies on the Voronoi diagram.

→ "Wavefront from obstacles, Voronoi diagram where two wavefronts meet."

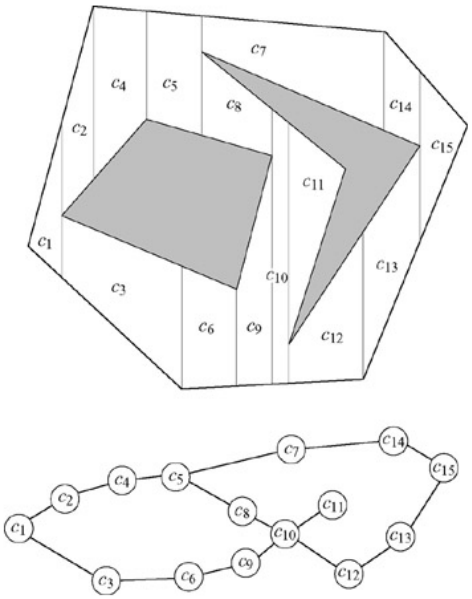
## 5.2 (Vertical) Cell Decomposition

*Idea:* Decompose the free space into simple cells and represent the connectivity of the free space $F$ by the adjacency graph of these cells (mostly in 2D).

It is called an exact cell decomposition if the union of all cells is exactly $F$, meaning there is no overlap between cells. The shape of the cells can be triangles, trapezoids, ...

### 5.2.1 Trapezoidal Decomposition

Free space is decomposed into trapezoids and triangles. Then, the adjacency graph of the cells is calculated.
Time $O(n \log n)$, space $O(n)$.



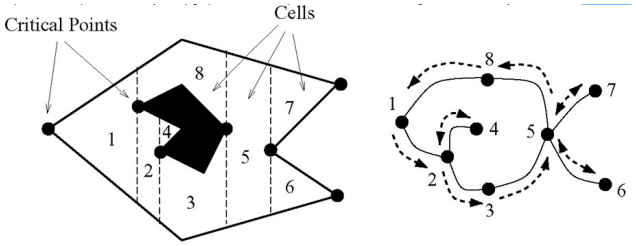




The planner searches the adjacency graph to determine the path of nodes. Then the planner constructs the point to point path, one trapezoid at a time, by connecting the midpoints of the vertical extensions to the centroids of each trapezoid. To connect the start and goal points, simply draw a straight line to the vertical extensions' midpoints of the appropriate trapezoids.

### 5.2.2 Boustrophedon Decomposition

Points on the obstacles, from which a separating line can be drawn in the upper and lower direction are called critical points. Then, an exhaustive walk through the critical points is performed in order to obtain a connectivity graph.




Exhaustive walk 1–2–4–2–3–5–6–5–7–5–8–1

## 5.3 Cell Decomposition in higher dimensions

### 5.3.1 The Halting Problem

**Completeness:** A complete algorithm finds a path if one exists and reports no otherwise.

Is it possible to determine whether a program will stop or will it run forever?
Suppose an algorithm $H$ that can decide whether $X$ will halt. By proof by contradiction, it can be shown that such an algorithm $H$ cannot exist. Furthermore, it can be shown that not all planning algorithms are complete.
*But:* Combinatorial algorithms (cell decomposition, visibility graphs & Voronoi diagrams) are completed since they cover every single point in $C_{free}$. $\rightarrow$ Motivates the use of combinatorial algorithms of rhigher dimensions.

### 5.3.2 Extension of vertical Cell Decomposition

Extension of vertical CD to $n$ dimensions:

1. Pick a dimension $x_i$
2. Sweep the configuration space using a (n-1)-dimensional plane orthogonal to $x_i$. Stop when critical connectivity events happen. When such a change happens, a new (n-1)-dimensional slice is produced.
3. Repeat for the (n-1)-dimensional slices
4. Sweeping yields convex n-cells, (n-1)-cells, ... (with a n-cell being a n-dim. polytope sandwiched between (n-1)-dim. slices).

### 5.3.3 k-d tree

Tree like data structure useful for finding points with certain properties. Construction:

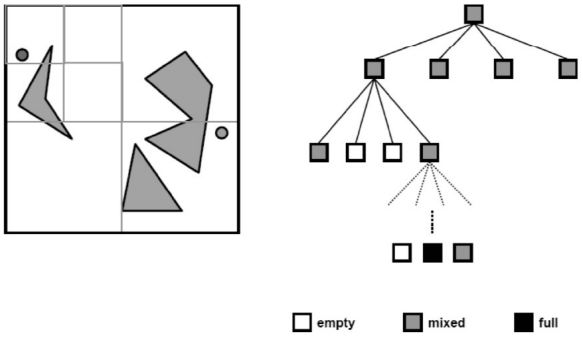
1. Pick dimension $i$, pick a point with coordinates $x$.
2. Split the points based on $x_i$ (greater or less than).
3. Repeat the above two steps recursively.

*Note:* depth $= \log n$ if balanced; construction takes $O(kn \log n)$ time.
Disadvantage: If the environment changes the k-d tree needs to be fully recomputed.

## 5.4 Approximate Cell Decomposition

*Idea:* The free space $F$ is represented by a collection of non-overlapping cells whose union is contained in $F$.
Cells usually have simple, regular shapes. It facilitates the hierarchical space decomposition. Examples are Quadtrees (2D) and Octrees (3D):

- Splitting stops if node is empty or full (e.g. up to $95\%$).
- Advantage: If the environment changes only some nodes need to be recomputed (not the whole tree).



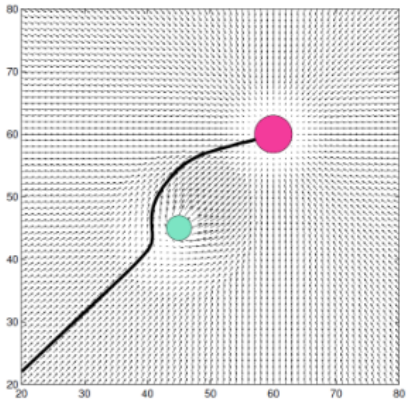

## 5.5 Potential Field Method

*Idea*: Define a potential function over the free space that has a global minimum at the goal and follow the steepest descent of the potential function (gradient descent).

Obstacles are surrounded by a repulsive field and the goal location by an attractive field. Ideal potential field:

- Global minimum at the goal.
- No local minima.
- Grows to infinity near obstacles.
- Is smooth.

*Advantage:* Easy to compute.
*Disadvantage:* Possible local minima (where robot gets stuck) and no consideration of dynamic constraints in their initial form (forces can be too high for the robot).




***Attractive component:*** A linear potential function results in a constant velocity (gradient) and thus an overshoot at the goal. That is why a quadratic function is better suited since the minimum is reached with a velocity of 0. Generally, it is combined with a linear function at a distance $d^*$ so that the gradient (which corresponds to the velocity) is not too large for the robot.

$$U_{att}(q) = \begin{cases} \frac{1}{2}\zeta d^2(q, q_{goal}), & d(q, q_{goal}) \leq d^*_{goal} \\ d^*_{goal}\zeta d(q, q_{goal}) - \frac{1}{2}\zeta (d^*_{goal})^2, & d(q, q_{goal}) > d^*_{goal} \end{cases}$$

$$\nabla U_{att}(q) = \begin{cases} \zeta(q - q_{goal}), & d(q, q_{goal}) \leq d^*_{goal} \\ \frac{d^*_{goal}\zeta(q - q_{goal})}{d(q - q_{goal})}, & d(q, q_{goal}) > d^*_{goal} \end{cases}$$

***Repulsive component:*** $Q^*$: Radius around obstacle where the

repulsive force acts. $D(q)$: Distance to obstacle.

$$U_{rep}(q) = \begin{cases} \frac{1}{2}\eta\left(\frac{1}{D(q)} - \frac{1}{Q^*}\right)^2, & D(q) \leq Q^* \\ 0, & D(q) > Q^* \end{cases}$$

$$\nabla U_{rep}(q) = \begin{cases} \eta\left(\frac{1}{Q^*} - \frac{1}{D(q)}\right)\frac{1}{D^2(q)}\nabla D(q), & D(q) \leq Q^* \\ 0, & D(q) > Q^* \end{cases}$$

## 6 Roadmapping with Random Sampling

*Problem with classical approaches:* Running time increases exponentially with the dimension of the C-Space. Modelling the free space becomes an arduous task.

Instead of looking at the whole space, sampling based methods are usually more efficient because they only require point-wise evaluations. They are probabilistically complete (+), meaning the probability that they will produce a solution approaches 1 as more time is spent. By reducing the map to samples, it will be easier to find a solution path since the computation effort is reduced and they can be applied to high-dimensional C-spaces (+). However, they are not as robust as methods with full knowledge of the space and cannot determine if there is no solution to the path planning problem.

Sampling uniformly in 2D and 3D has to be correctly parametrized! Cases:

- Unit line, square, cube: Pick a random number $r \in [0, 1]$ for each dimension.
- Sample in interval $[a, b]$: $(b - a)r + a,\ r \in [0, 1]$.
- Angle: Pick angle $\theta$ uniformly at random from $[0, 2\pi]$.
- Rotations: Depends on the representation, in axis-angle notation the axis and the angle are sampled separately.

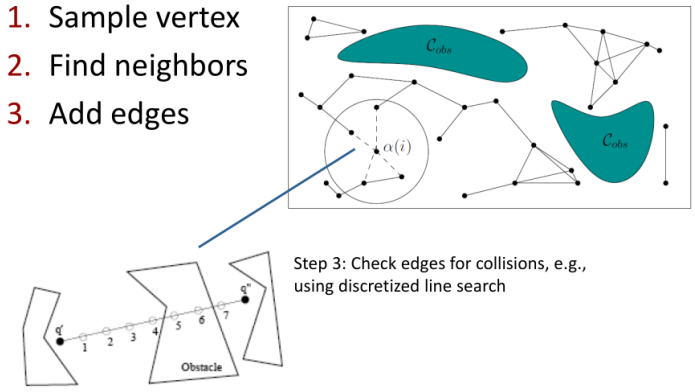
### 6.1 Multi-Query

A multiple query approach tries to capture the connectivity of the free space as good as possible with the goal to answer multiple, different queries for paths very fast.
Assumption: Static obstacles.

#### 6.1.1 PRM - Probabilistic Roadmaps

**Basic steps for constructing PRMs:**

1. Sample vertices (uniformly) and keep vertices that do not collide with obstacles (= milestones).
2. Find neighbour vertices/milestones with
   k-nearest neighbour or
   neighbours within a specified radius.
3. Connect neighbouring vertices with edges (lines) (and check for collisions on connecting line using e.g. discretized line search).
4. Add vertices and edges until roadmap is dense enough.




**Query processing:**

1. Connect $\boldsymbol{q}_{start}$ and $\boldsymbol{q}_{goal}$ with the graph (check for collisions on connecting line).
2. Search a path from $\boldsymbol{q}_{start}$ to $\boldsymbol{q}_{goal}$ in the graph.

**Drawback 1:** PRMs don't perform well when there are narrow passages ($\rightarrow$ you probably won't end up with a fully connected graph).
**Solution 1:** *Expansion*

- Further sampling ($\rightarrow$ increase number of nodes).
- Resampling with a random walk (when an obstacle is hit, continue in a random direction until a maximum length $L$ is reached).
- Bridge Sampling which focuses on generating samples in narrow passages.
- OBPRM.

**Drawback 2:** Many abrupt velocity changes due to the typical zick-zack paths.
**Solution 2:** Smooth the resulting path.

**Advantages of PRMs:**

- Probabilistically complete.
- Apply easily to high-dimensional C-space.
- Fast queries (with enough preprocessing).

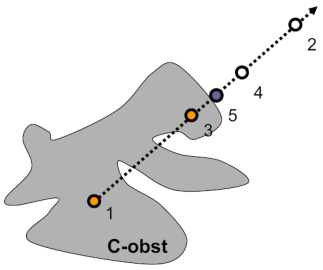
**Homotopic paths:** Two paths with the same endpoints are homotopic if one can be continuously deformed into the other. A homotopic class of paths contain all paths that are homotopic to one another. Homotopy ensures that after the path correction, the space topology gets visited from the same side as originally planned. (If during the merging process the path goes through an object, it is not considered homotopic.)

#### 6.1.2 OBPRM - Obstacle-Based PRM

Obstacle-based PRMs are constructed by sampling only close to obstacles.

Basic idea (for workspace obstacle $S$):

1. Find a point in $S$'s C-obstacle (robot placement colliding with $S$).
2. Select a random direction in C-space.
3. Find a free point in that direction.
4. Find boundary point between them using binary search (collision checks).



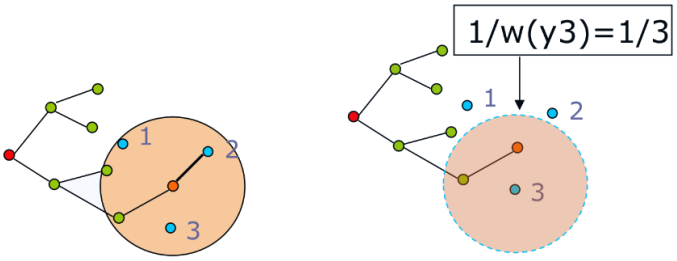

## 6.2 Single-Query PRM

Single query planners try to solve a single query as fast as possible, without trying to cover the whole free space.

**Idea:** Grow two trees from *Start* and *Goal* configurations. Randomly sample nodes around existing nodes. Connect a node in the tree rooted at *Start* to a node in the tree rooted at *Goal*. → Expansion + Connection.

**Expansion + Connection:**

1. Expand trees from *Start* and *Goal*.
2. Pick a node $x$ with probability $1/w(x)$ ($w(x)$: number of nodes within certain radius, including node $x$)
3. Randomly sample $k$ points $(y_1, ..., y_k)$ around $x$.
4. Add sample $y_i$ with weight $w(y_i)$ and probability $1/w(y_i)$ to the tree if
   - $\frac{1}{w(y_i)} > \frac{1}{w(x)} \Leftrightarrow w(y_i) < w(x)$.
   - $y_i$ is collision free.
   - $y_i$ can see $x$.
5. If a pair of nodes from start tree and goal tree are close and can see each other, then connect them and terminate.




**Termination condition:** The program iterates between Expansion and Connection, until

- two trees are connected, or
- max. number of expansion & connection steps is reached.

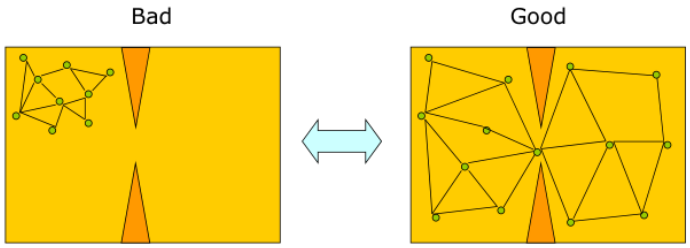
**Drawback:** If no connection possible, the algorithm expands and does not terminate if no maximal number of steps is specified.

## 6.3 Coverage, Connectivity, $\epsilon, \alpha, \beta$ - Expansiveness

For narrow passages the number of the milestones (vertices) used is essential. How do you how many vertices are actually needed? Typical issues of PRMs are coverage and connectivity.

**Coverage:** The coverage is good if the milestones are distributed in such a way that (almost) any point in the free C-space can be connected to one milestone via a straight line.

**Connectivity:** The connectivity is good if every milestone is reachable from any other milestone. Especially with narrow passages, the connectivity can be hard to capture.




**Expansiveness:** The coverage and connectivity are characterized by the ($\epsilon$, $\alpha$, $\beta$)-expansiveness of the space. The free space $F$ is ($\epsilon$, $\alpha$, $\beta$)-expansive if $F$ is $\epsilon$-good and for each subset $S$ of $F$, its $\beta$-lookout is at least $\alpha$ fraction of $S$. If the C-space is expansive, then a roadmap can constructed efficiently with good connectivity and coverage.

**Burschka-Style:**
Definitions:

- reach$(x) \subseteq F$: Visibility set of point $x$, i.e. all configurations in free space $F$ that can be connected to $x$ by a straight-line path in $F$.
- reach$(G)$: The space that can be seen from any point in a set $G$.
- $\mu(G)$ : Volume/area of some set $G$.
- $\beta$-lookout$(S)$: Subset of point in $S$ that can see at least $\beta$ fraction of $F \setminus S$.

Approach:

1. Find point $x \in F$ with smallest $\mu(\text{reach}(x))$. Then:
$$S = \text{reach}(x)$$
2. Every free configuration sees at least $\epsilon$ fraction of the free space:
$$\epsilon = \frac{\mu(S)}{\mu(F)} \in (0, 1]$$
The corresponding configuration is called $\epsilon$-good.
3. Choose some subset $A \subseteq S$ that can reach a lot and has an easy to calculate area.
4. $\alpha = \frac{\mu(A)}{\mu(S)} \in (0, 1]$
5. $\beta$-lookout (not really according to the definition, but best way to do it in the exam):
$$\beta = \frac{\mu(\text{reach}(A) \setminus S)}{\mu(F \setminus S)} \in (0, 1]$$

The goal is to keep $\alpha$ and $\beta$ balanced. Larger parameters = lower cost of constructing a roadmap with good connectivity and coverage.

Using these values, the number $n$ of samples that is needed for a good connectivity can be calculated:

$$n = \frac{8 \ln(\frac{8}{\epsilon\alpha\gamma})}{\epsilon\alpha} + \frac{3}{\beta},$$

where $1-\gamma$ (success rate) measures the probability that the uniformly sampled milestones have the correct connectivity ($\gamma \in (0,1]$, failure rate).

***Hsu's paper & Rickert's lecture:*** A free space $F$ is ($\alpha$, $\beta$, $\epsilon$)-expansive, if it satisfies these conditions:

1. A free space $F$ is $\epsilon$-good if for every $x \in F$, $\mu(\text{reach}(x)) \geq \epsilon \cdot \mu(F)$.
2. The $\beta$-lookout of a set $S \subset F$ is the set of points in $S$ that see a $\beta$-fraction of $F \setminus S$:

$$\beta\text{−lookout}(S) = \{q \in S \mid \mu(\text{reach}(q) \setminus S) \geq \beta \cdot \mu(F \setminus S)\}.$$

3. ($\epsilon$, $\alpha$, $\beta$)-expansive: $F$ is ($\epsilon$, $\alpha$, $\beta$)-expansive if it is $\epsilon$-good and each of its subsets $S$ has a $\beta$-lookout with a volume of at least $\alpha \cdot \mu(S)$:

$$\mu(\beta\text{-lookout}(S)) \geq \alpha \cdot \mu(S).$$

With $\epsilon, \alpha, \beta \in (0,1]$, reach$(\cdot)$ as the set of visible points of a point and $\mu(\cdot)$ denoting the volume of a set of points.

**Theorems:**

- Probability of achieving good connectivity increases ***exponentially*** with the number of milestones (in an expansive space). If $(\epsilon, \alpha, \beta)$ decreases, then there is a need to increase the number of milestones (to maintain good connectivity).
- Probabilistic completeness: In an expansive space, the probability that a PRM planner fails to find a path when one exists goes to $0$ exponentially in the number of milestones (~running time).

$$\gamma = \frac{8}{\epsilon\alpha} \exp\left(-\frac{\epsilon\alpha}{8}\left(n - \frac{3}{\beta}\right)\right) \propto \exp(-kn), k \in \mathbb{R}$$

**Limitations in practice:** It does not tell you when to stop growing the roadmap. A planner stops when either a path is found or max steps are reached.

**Methods to improve connectivity:**

- Increase number of samples $n$
- Random walk
- OBPRM
- Repairing approximate paths

## 6.4 RRT - Rapidly Exploring Random Trees

A roadmap is likely to have a lot of useless information stored if we want to run a single query.

RRTs without obstacles simply grow trees from a point. Basically, they try to connect new points to the closest part of the existing ones with a linear search. The tree evolves iteratively over time. With obstacles, they try to extend the tree as much as possible, in case of collision with one, they stop at the collision point and generate a new node. In order to control the expansion of the tree, a growth limit $\Delta q$ for the maximum length of the edges can be implemented, where the target node gives the direction, but the actual implemented node is constrained by $\Delta q$. Another way to steer the growth is by increasing the probability of generating new nodes near of $q_goal$, so that the tree expands in that direction.

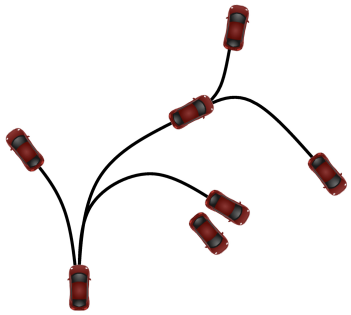
**Procedure:**

1. Initialize tree with $q_{start}$ as first node.
2. Sample a random point $q_s$ (every $n$-th iteration, choose $q_{goal}$).
3. Find the closest neighbor $q_n$ in tree.
4. Add points on the connecting line between $q_s$ and $q_n$ to the tree.
   - With step width $d$.
   - Checkpoints for collision with obstacles.
   - Stop when a collision is detected.
5. Repeat steps 2 - 5 until $q_{goal}$ is reached.

*Note:* For faster execution, a bi-directional tree can be generated that grows from $q_{start}$ and $q_{goal}$.

### 6.4.1 Kinodynamic RRT

For solving problems for systems with differential constraints (like non-holonomic motion constraints), kinodynamic RRT can be used. Standard PRM and RRT cannot be applied here. Kinodynamic RRT grows the tree respecting the differential constraints.



### 6.4.2 RRT*

Since RRT (and also PRM) are not optimal, we have to "re-wire" the graph structure to gain optimality.
Procedure: For each new sample $q_n$, check its $\log n$ neighborhood and if there are better paths from $q_{start}$ to $q_n$, pick that path.
This is an asymptotically optimal sampling based algorithm, because as the number of samples goes to infinity, an optimal path from $q_{start}$ to $q_{goal}$ is obtained.

# 7 Probabilistic Robotics

## 7.1 Kalman Filter

The basic idea of a Kalman filter is that you interpret your robots state as a normal (gaussian) distribution with mean and variance. Besides the estimate for the state $x$ you also have some estimate for the uncertainty in your value for $x$ that is represented through a normal distribution and its variance / covariance matrix $P$. You also model your sensors with white noise (gaussian, zero-mean).

***Explanantion from the internet:*** A Kalman filter is an optimal estimator i.e. infers parameters of interest from indirect, inaccurate and uncertain observations. It is recursive so that new measurements can be processed as they arrive . If the noise is Gaussian the Kalman filter is optimal, i.e. it minimizes the mean square error of the estimated parameters.

***Requirements for optimality:***

- Linear system and measurement model
- Gaussian and zero-mean measurement and process noise

Illustrative: The more certain the initial estimate is, the higher and narrower the peak. The greater the noise (amount of error), the faster the peak spreads.

Kalman Gain:

- $K_k = 0$: $R \to \infty$ or $P_K^- = 0$, i.e. trust in measurement is very low or trust in model is very high.
- $K_k = 1$: $R = 0$ or $P_K^- \to \infty$, i.e. trust in measurement is very high or trust in model is very low.

**Prediction step**

| | |
|---|---|
| Mean | $\hat{x}_{k+1}^- = A\hat{x}_k + Bu_k$ |
| Covariance $\sigma_3^2 = \sigma_1^2 + \sigma_2^2$ | $P_{k+1}^- = AP_kA^\top + Q$ |

**Update step**

| | |
|---|---|
| Kalman Gain | $K_k = P_k^- H^\top (HP_k^- H^\top + R)^{-1}$ |
| Mean | $\hat{x}_k = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-)$ |
| Covariance $\frac{1}{\sigma_3^2} = \frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}$ | $P_k = (I - K_kH)P_k^-$ |

**with**

| | |
|---|---|
| $\hat{x} \in \mathbb{R}^N$ | state estimate mean |
| $\hat{x}^- \in \mathbb{R}^N$ | predicted state estimate mean |
| $A \in \mathbb{R}^{N\times N}$ | state-transition model |
| $u \in \mathbb{R}^M$ | control input |
| $B \in \mathbb{R}^{N\times M}$ | control-input model |
| $P \in \mathbb{R}^{N\times N}$ | state covariance matrix |
| $Q \in \mathbb{R}^{N\times N}$ | process noise covariance matrix |
| $K \in \mathbb{R}^{N\times Z}$ | Kalman gain |
| $R \in \mathbb{R}^{Z\times Z}$ | measurement noise covariance matrix |
| $z \in \mathbb{R}^Z$ | measurement |
| $H \in \mathbb{R}^{Z\times N}$ | observation model / measurement matrix |

#### 7.1.1 Observability

$$O = \begin{bmatrix} H \\ HA^1 \\ \vdots \\ HA^{n-1} \end{bmatrix}$$

The system is observable if rank$(O) = n$ (i.e. $n$ linearly independent rows). $n$ is the dimension of the state vector $x$.

#### 7.1.2 Difficulties

- Presence of the unknown and unmeasurable noise vectors $v(k)$ (observation noise) and $w(k)$ (process noise / model error).
- The state in general can not be directly observed from the outputs, $H$ may not be invertible.

### 7.2 Extended Kalman Filter

The (first order) EKF allows to use non-linear system models for Kalman Filtering and is a sub-optimal extension of the original KF algorithm. The EKF adapts techniques from calculus, namely multivariate Taylor Series expansions, to linearize the functions $f$ and $h$ around a working point (use the means).

**Prediction step**

| | |
|---|---|
| Mean | $\hat{x}_{k+1}^- = f(\hat{x}_k, u_k, 0)$ |
| Covariance | $P_{k+1}^- = AP_kA^\top + WQW^\top$ |

**Update step**

| | |
|---|---|
| Kalman Gain | $K_k = P_k^- H^\top (HP_k^- H^\top + VRV^\top)^{-1}$ |
| Mean | $\hat{x}_k = \hat{x}_k^- + K_k(z_k - h(\hat{x}_k^-, 0))$ |
| Covariance | $P_k = (I - K_kH)P_k^-$ |

**with**

| | |
|---|---|
| $x_{k+1} = f(x_k, u_k, w_k) \in \mathbb{R}^N$ | state transfer function |
| $w_k \in \mathbb{R}^N$ | process noise |
| $A = \frac{\partial f}{\partial x_k} \in \mathbb{R}^{N\times N}$ | Jacobian of $f$ with respect to $x_k$ |
| $W = \frac{\partial f}{\partial w_k} \in \mathbb{R}^{N\times N}$ | Jacobian of $f$ with respect to $w_k$ |
| $z_k = h(x_k, v_k) \in \mathbb{R}^Z$ | measurement function |
| $v_k \in \mathbb{R}^Z$ | measurement noise |
| $H = \frac{\partial h}{\partial x_k} \in \mathbb{R}^{Z\times N}$ | Jacobian of $h$ with respect to $x_k$ |
| $V = \frac{\partial h}{\partial v_k} \in \mathbb{R}^{Z\times Z}$ | Jacobian of $h$ with respect to $v_k$ |

Missing here: EKF for SLAM

### 7.3 Unscented Kalman Filter

Instead of using first order approximations of the Taylor series (EKF), the unscented filter extends to the second (or higher) order approximations. EKF may cause significant error for highly nonlinear systems because local linearity assumptions break down when the higher order terms become significant.

The idea of the UKF is to produce several sampling points (Sigma Points) from the current state and noise distributions. Then, propagating these points through the nonlinear map to get a more accurate estimation of the mean and covariance of the mapping results.

The main assumption here is the unscented transformation, which allows to calculate statistics about a random variable that has undergone a nonlinear transformation by saying that it is easier to approximate a probability distribution than an arbitrary nonlinear function. The estimates of the mean and covariance are accurate to the second order of the Taylor series expansion of the the nonlinear map. In this way, it avoids the need to calculate the Jacobian, hence incurs only the similar computation load as the EKF.

**Initialization**

State mean $\hat{x}_0 = E(x_0)$

Mean vector $\hat{x}_0^a = \begin{bmatrix} \hat{x}_0^\top & 0 & 0 \end{bmatrix}^\top$

Contains mean values of state ($x$), process noise ($v$) and measurement noise ($n$). $x_t^a = [x_t^\top, v_t^\top, n_t^\top]$

Covariance matrix $P_0^a = \begin{bmatrix} P_0 & 0 & 0 \\ 0 & Q & 0 \\ 0 & 0 & R \end{bmatrix}$

Covariance matrices of state $P_0$, process noise $Q$ and measurement noise $R$.

**Prediction step**

*(a) Computing sigma points*

Sigma points $\chi_{k-1}^a = \begin{bmatrix} \hat{x}_{k-1}^a & \hat{x}_{k-1}^a \pm \sqrt{(n_x + \lambda) P_{k-1}^a} \end{bmatrix}$

*(b) Time update*

Sigma points $(\chi_k^x)^- = f(\chi_{k-1}^x, \chi_{k-1}^v)$

State mean $\hat{x}_k^- = \sum_{i=0}^{2n_x} W_i^{(m)} (\chi_{i,k}^x)^-$

Covariance $P_k^- = \sum_{i=0}^{2n_x} W_i^{(c)} \left[ (\chi_{i,k}^x)^- - \hat{x}_k^- \right] \left[ (\chi_{i,k}^x)^- - \hat{x}_k^- \right]^\top$

**Update step**

Predicted meas. $Y_k^- = h((\chi_k^x)^-, \chi_{k-1}^n)$

Meas. mean $\hat{y}_k^- = \sum_{i=0}^{2n_x} W_i^{(m)} Y_{i,k}^-$

Meas. cov. $P_{y_k y_k} = \sum_{i=0}^{2n_x} W_i^{(c)} \left[ Y_{i,k}^- - \hat{y}_k^- \right] \left[ Y_{i,k}^- - \hat{y}_k^- \right]^\top$

Cross cov. $P_{x_k y_k} = \sum_{i=0}^{2n_x} W_i^{(c)} \left[ \chi_{i,k}^{x-} - \hat{x}_k^- \right] \left[ Y_{i,k}^- - \hat{y}_k^- \right]^\top$

Kalman Gain $K_k = P_{x_k y_k} P_{y_k y_k}^{-1}$

State mean $\hat{x}_k = \hat{x}_k^- + K_k (y_k - \hat{y}_k^-)$

Covariance $P_k = P_k^- - K_k P_{y_k y_k} K_k^\top$

**Notation**

$x^a = [x^\top, v^\top, n^\top], \quad \chi^a = [(\chi^x)^\top, (\chi^v)^\top, (\chi^n)^\top]^\top$

$n_a = n_x + n_v + n_n$ : Dimension of $x$, $v$ and $n$.

$\lambda$ : Composite scaling parameter.

$W$ : Weights

**Weights**

$$W_0^{(m)} = \frac{\lambda}{n_x + \lambda}$$
$$W_0^{(c)} = \frac{\lambda}{n_x + \lambda} + (1 - \alpha^2 + \beta)$$
$$W_i^{(m)} = W_i^{(c)} = \frac{\lambda}{2(n_x + \lambda)}, i = 1, \dots, 2n_x$$

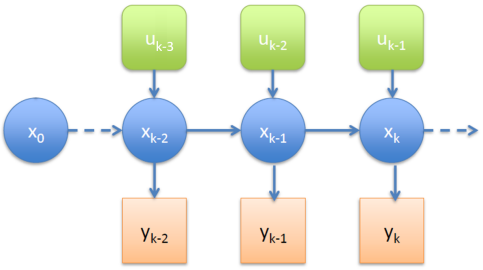
## 7.4 Bayes Filter

**Assumptions:**

- Markov property: $P(x_k | x_{k-1}, \dots, x_0) = P(x_k | x_{k-1})$
- Observation: $P(y_k | x_k, \dots, x_0) = P(y_k | x_k)$

**Given:**

- Measurements: $y_{1:k} = y_1, \dots, y_k$
- Control inputs: $u_{0:k-1} = u_0, \dots, u_{k-1}$
- Dynamic model (e.g. motion model): $P(x_{k+1} | x_k, u_k)$
- Prior probability: $P(x_0)$
- Sensor model: $P(y_k | x_k)$




**Compute:**

Most likely state $x$ at time $k$ given commands $u_{0:k-1}$ and measurements $y_{1:k-1}$(posterior distribution)

$$P(x_k | y_{1:k}, u_{0:k-1}) = \frac{P(y_k | x_k, y_{1:k-1}, u_{0:k-1}) P(x_k | y_{1:k-1}, u_{0:k-1})}{P(y_k | y_{1:k-1}, u_{0:k-1})}$$
$$= \eta_k \underbrace{P(y_k | x_k)}_{\text{observation}} \int_{x_{k-1}} \underbrace{P(x_k | u_{k-1}, x_{k-1})}_{\text{state prediction}} \underbrace{P(x_{k-1} | y_{1:k-1}, u_{0:k-2})}_{\text{recursive instance}} dx_{k-1}$$

Without measurements, only prediction:

$$P(x_k) = \int_{x_{k-1}} P(x_k | x_{k-1}, u_{k-1}) P(x_{k-1}) dx_{k-1}$$

## 7.5 Particle Filter

The state representations of the Bayes and Kalman filter are restricted to probability density or Gaussian functions. They also cannot model multiple hypotheses in their initial form, for example the uncertainty when a robot could be in one of multiple locations. (Multi-hypothesis KF could do that, but it's only a combination of multiple KFs).

The particle filter uses a population of randomly initialized particles to track high-likelihood regions of the state space. The basic procedure is:

1. If no initial distribution is given, start with a uniform sample distribution.
2. Apply motion model to particles by sampling from the corresponding distribution.

3. After observing the measurement, weight each particle by its likelihood for the observation.
4. Resampling: Generate a new set of samples by weighted random selection from the current set of particles.
5. Repeat from 2.

At any time, the distribution is represented by the weighted set of particles.

**Particle Deprivation:** There are not particles in the vicinity of the correct state.

- Occurs as the result of the variance in random sampling. An unlucky series of random numbers can wipe out all particles near the true state. This has non-zero probability to happen at each time $\rightarrow$ will happen eventually.
- Popular solution: Add a small number of randomly generated particles when resampling.

**Problem with Resampling:** Resampling induces loss of diversity. The variance of the particles decreases, the variance of the particle set as an estimator of the true belief increases.

- Solution 1: Resample only when effective sample size is low.
- Solution 2: Low-variance-sampling.