Principles of Robot Autonomy I

Models: Collisions and C-space, Kinematic/dynamics motion models
Planning and Control: A* path planning

Stanford University

IPRL

Principles of Robot Autonomy I

Announcements:

- Lab section 1 this week (self-study, not in-lab)
- HW 1 due next Thurs, Oct 9
- Slides uploaded to canvas before lecture

Lecture 3:

- Collision checking and C-space
- Robot kinematic/dynamic motion models
- Planning-control stack intro
- A* planning

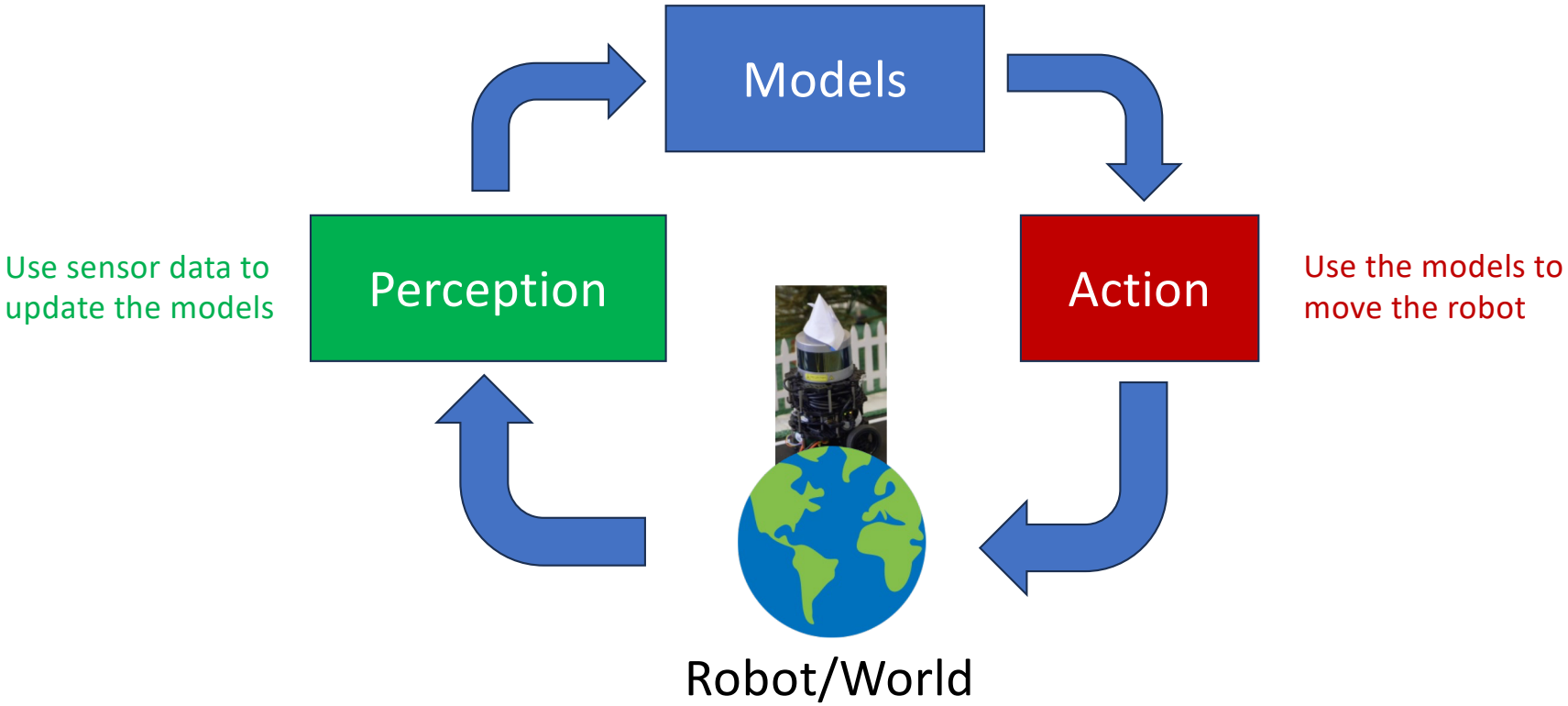
Stanford University

MSL

IPRL

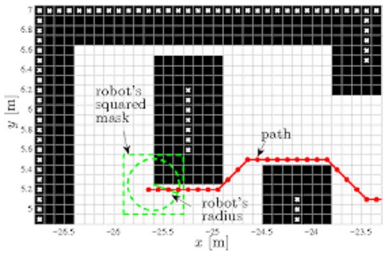
ASL Autonomous Systems Lab

Full Stack Autonomy: the Perception-Action Loop

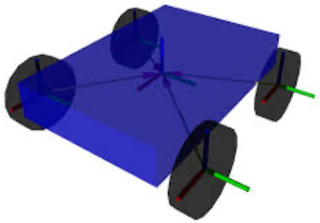


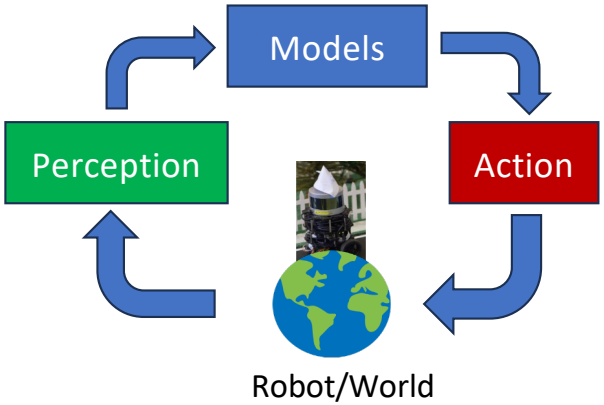
Models: Environment and Robot Representations

Maps:



Robot models
or simulators:



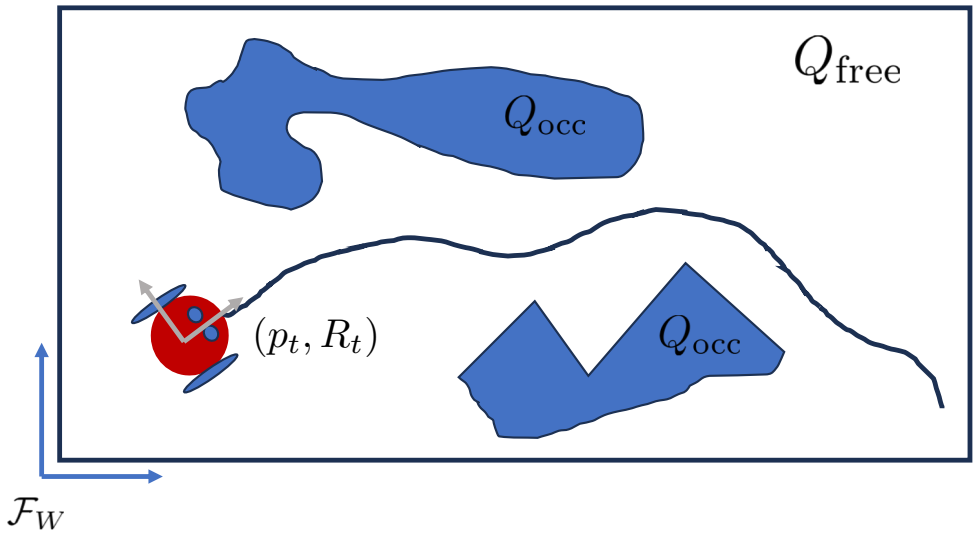


Robot-Map Collisions

Main point: Use robot pose to represent robot body in world frame to check for **collisions** and to plan **collision free motion.**

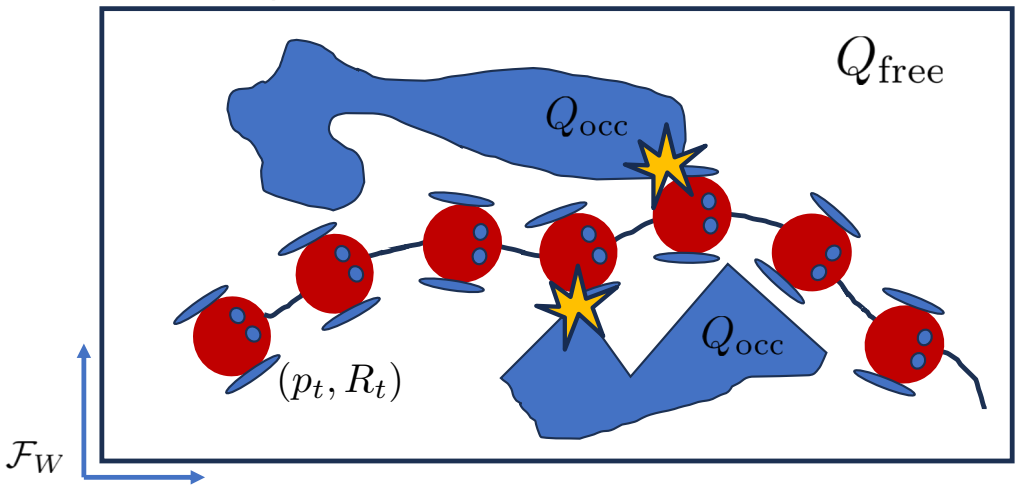
Robot body in world frame:

$$B(p_t, R_t) = \{R_t b + p_t \mid b \in B\}$$



Collisions and C-Space

Task Space



More rigorously, set of robot poses:

$$C_{\text{free}} = \{(p, \theta) \mid B(p, \theta) \cap Q_{\text{occ}} = \emptyset\}$$

$$C_{\text{occ}} = \{(p, \theta) \mid \exists b \in B(p, \theta) \text{ s.t. } b \in Q_{\text{occ}}\}$$

Can simplify to: $C_{\text{occ}} = \{(p, \theta) \mid p \in Q_{\text{occ}} \oplus -B(0, \theta)\}$

Collision check: $\boxed{(p, \theta) \in C_{\text{occ}}}$

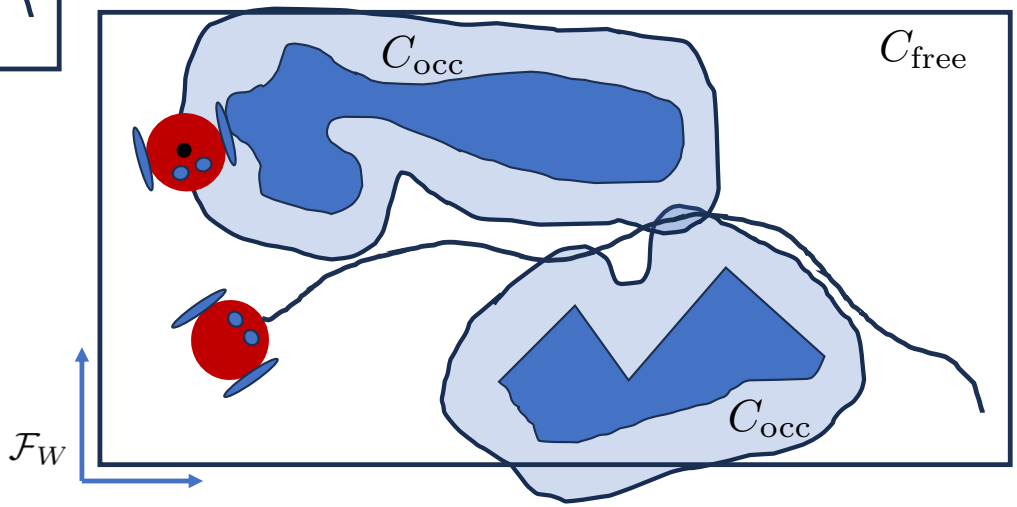
Robot body in world frame:

$$B(p_t, R_t) = \{R_t b + p_t \mid b \in B\}$$

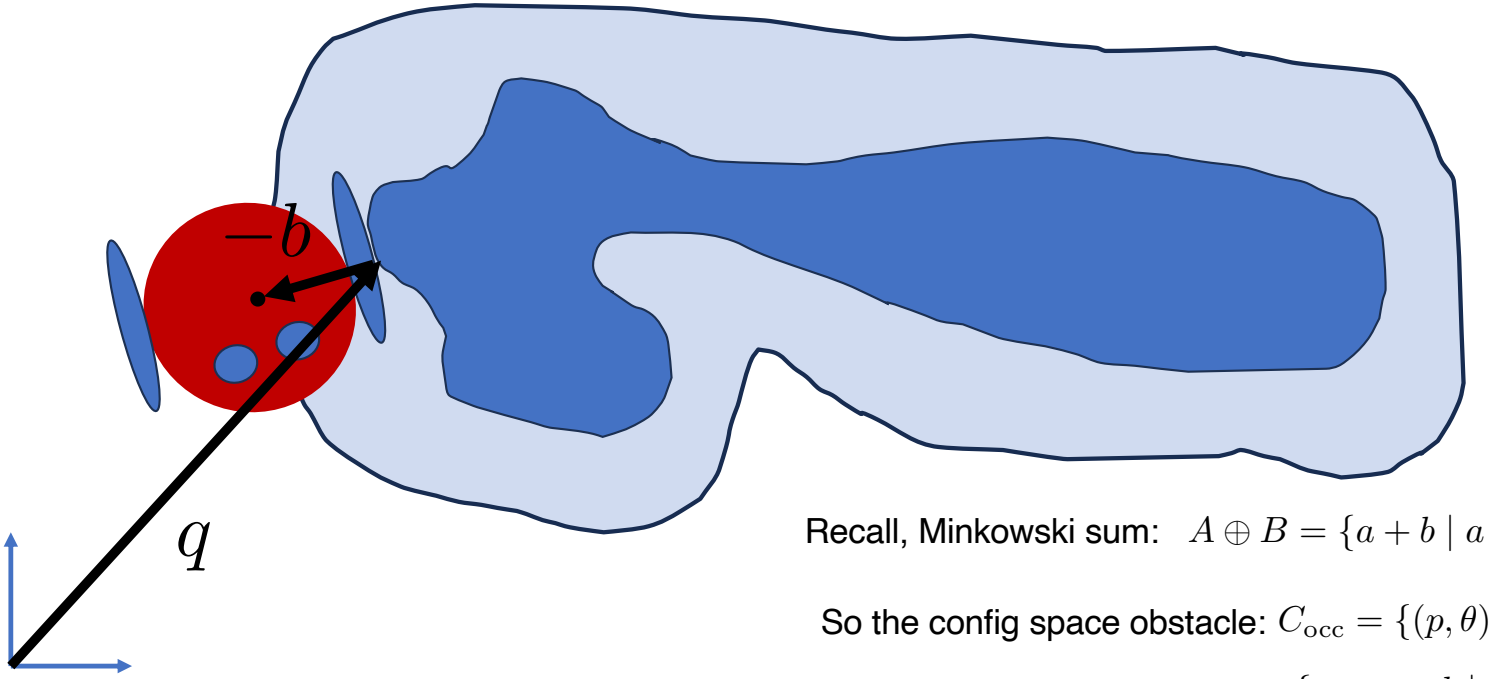
Collision check: $B(p_t, R_t) \cap Q_{occ} = \emptyset$

Can be computationally challenging!

Configuration Space (C-Space)



Challenges and Practicalities

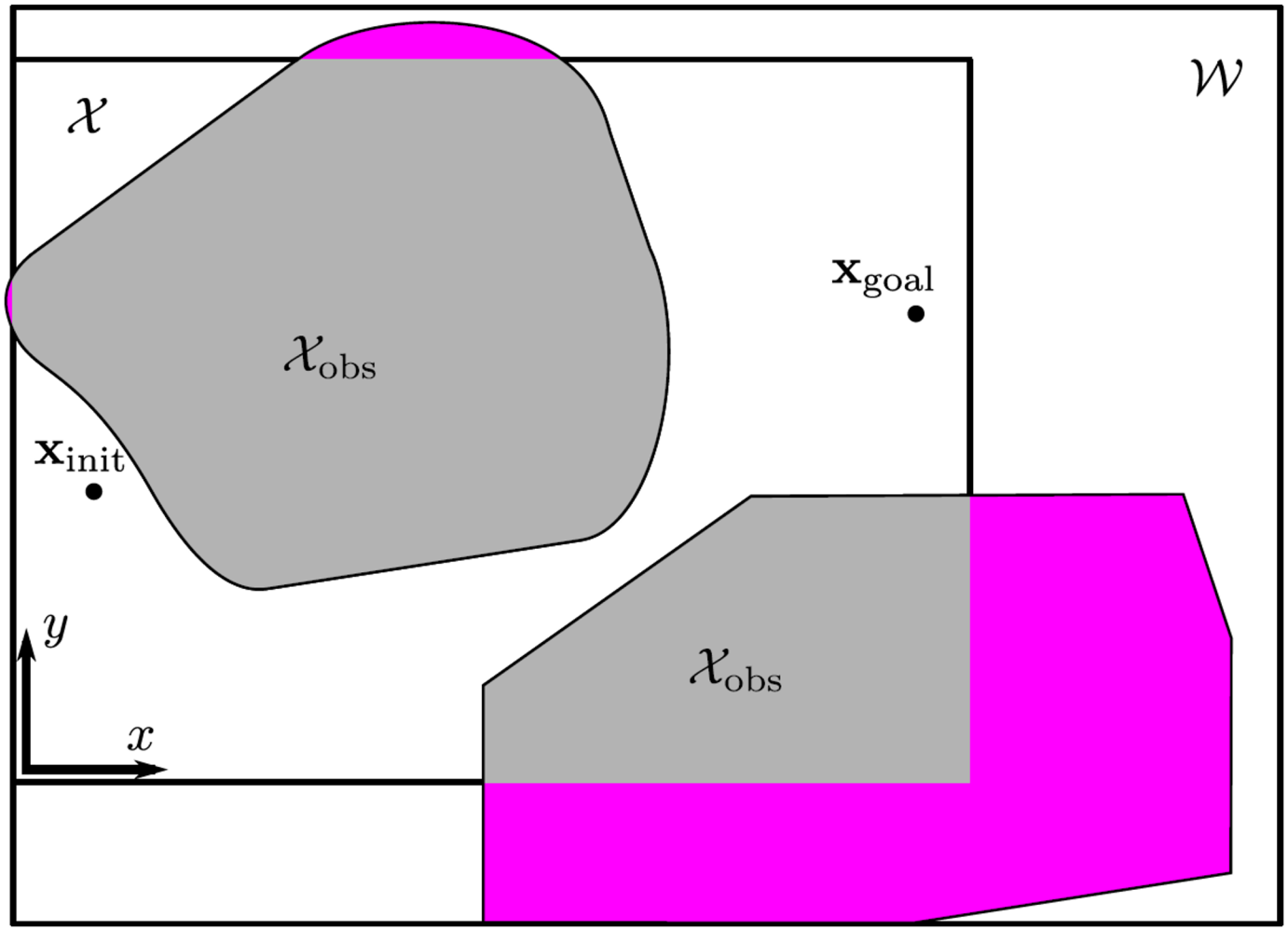


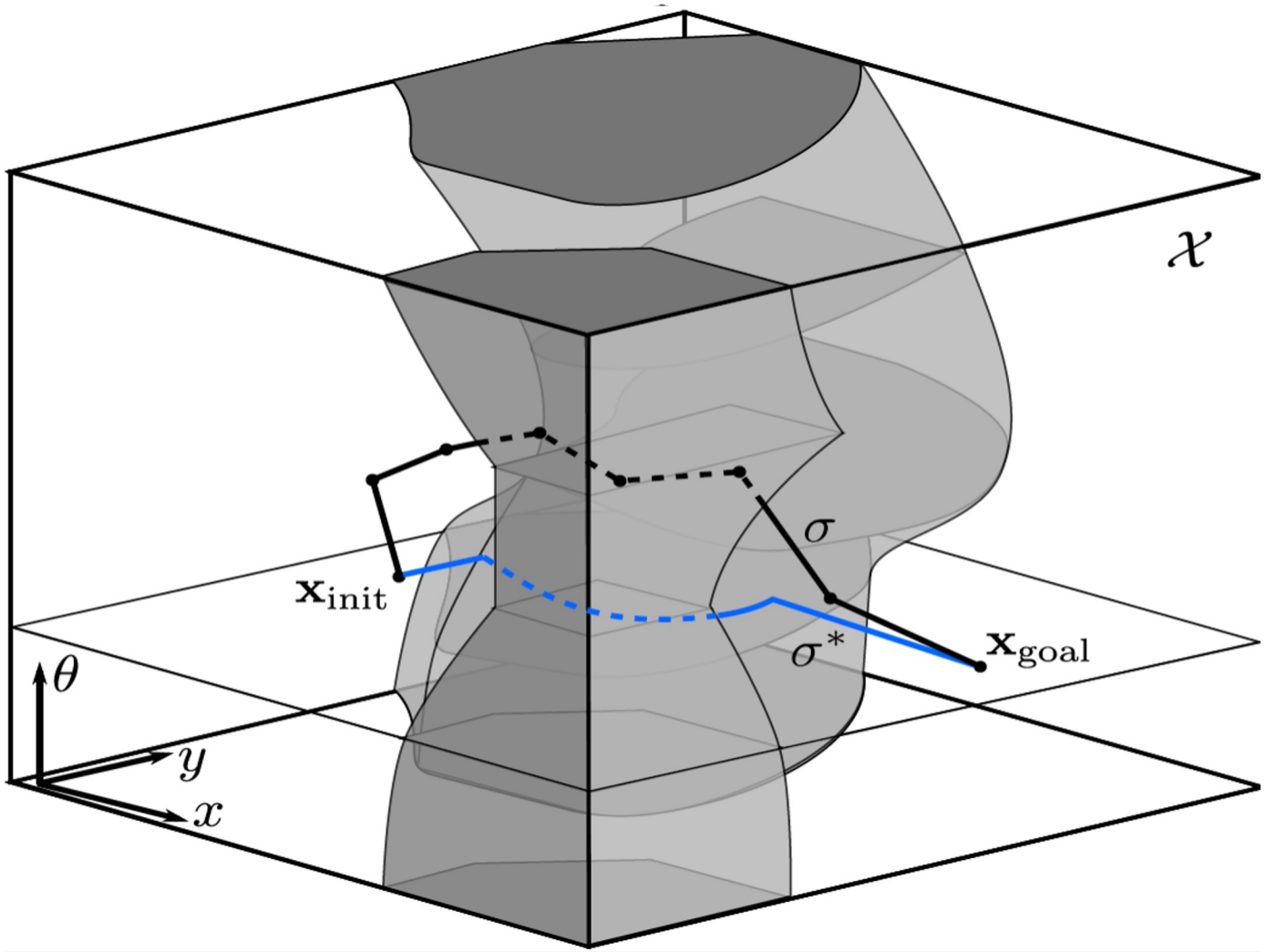
Recall, Minkowski sum: $A \oplus B = \{a + b \mid a \in A, b \in B\}$

So the config space obstacle: $C_{\text{occ}} = \{(p, \theta) \mid p \in Q_{\text{occ}} \oplus -B(0, \theta)\}$

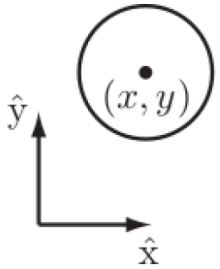
Just means: $\{p = q - b \mid q \in Q_{\text{occ}}, b \in B(0, \theta)\}$

Useful conceptual tool, but In practice this is all very difficult to compute! We usually approx. B as circle or sphere, or use specialized collisions checking algorithms.

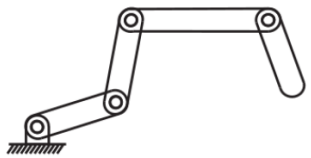

$\mathcal{X}$
$\mathcal{W}$
$\mathcal{X}_{\text{obs}}$
$\mathbf{x}_{\text{goal}}$
$\mathbf{x}_{\text{init}}$
y
x
$\mathcal{X}_{\text{obs}}$


$\mathcal{X}$
σ
$\mathbf{x}_{\text{init}}$
σ^*
$\mathbf{x}_{\text{goal}}$
θ
y
x

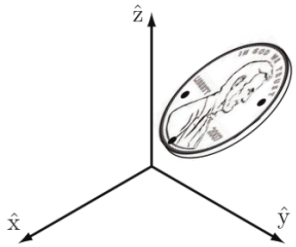
Configuration space examples



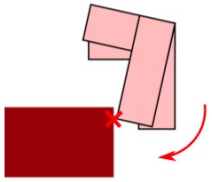
$d = 2$

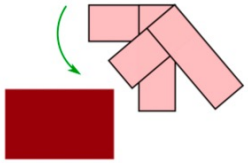


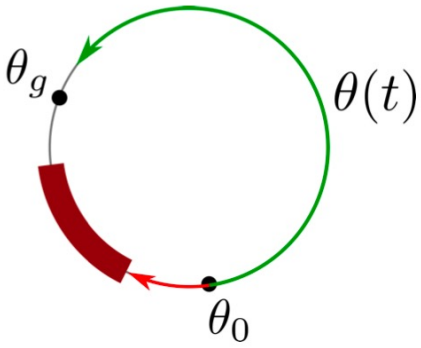
$d = 4$



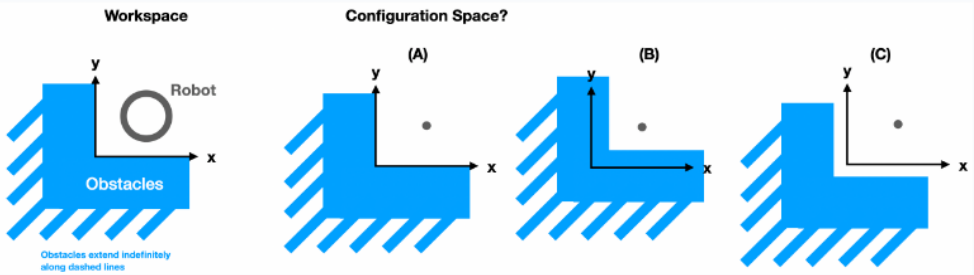
$d = 6$







For the robot workspace on the left with blue obstacles, what is the correct configuration space?

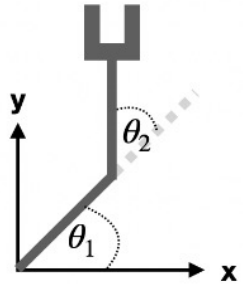


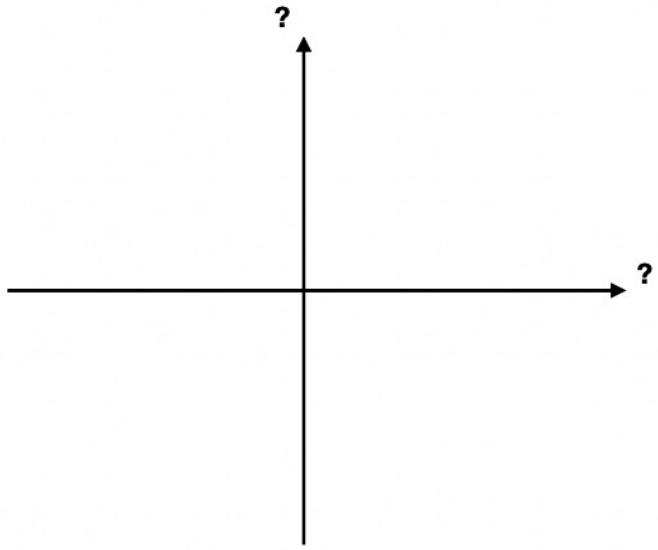
(A)

(B)

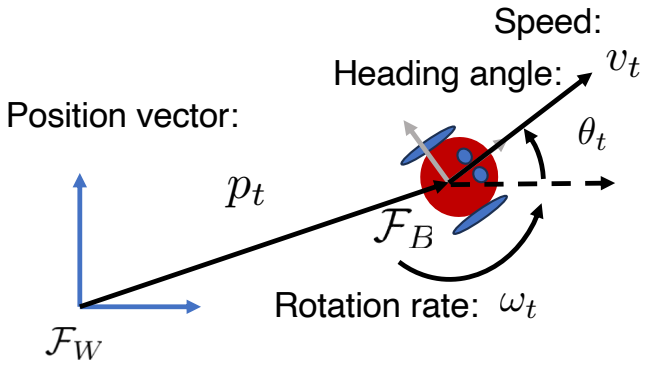
(C)

For the current state of the two-link robot robot arm on the left, click on the corresponding point in the configuration space on the right





Robot Kinematics and Dynamics



Robot pose: (p_t, θ_t)

Control inputs: (v_t, ω_t)

Differential Drive Robot (continuous time):

$$\begin{aligned} \dot{x}_t^p &= v_t \cos(\theta_t) \\ \dot{y}_t^p &= v_t \sin(\theta_t) \\ \dot{\theta}_t &= \omega_t \end{aligned}$$

Recall, first order Euler time discretization:

$$\dot{x} = f(x, u)$$
$$\frac{x_{t+1} - x_t}{\delta t} \approx f(x_t, u_t)$$
$$x_{t+1} = x_t + f(x_t, u_t)\delta t$$

Differential Drive Robot (kinematic/dynamic):

$$\begin{aligned} x_{t+1}^p &= x_t^p + v_t \cos(\theta_t)\delta t \\ y_{t+1}^p &= y_t^p + v_t \sin(\theta_t)\delta t \\ \theta_{t+1} &= \theta_t + \omega_t \delta t \\ v_{t+1} &= v_t + a_t \delta t \\ \omega_{t+1} &= \omega_t + \alpha_t \delta t \end{aligned}$$

Velocities

Accelerations (or forces/torques)

Nonholonomic: constraint on velocity vector
Nonlinear

Models: Robot Kinematics and Dynamics

Concept of state:

Kinematic model:

$$x^p_{t+1} = x^p_t + v_t \cos(\theta_t)\delta t$$
$$y^p_{t+1} = y^p_t + v_t \sin(\theta_t)\delta t$$
$$\theta_{t+1} = \theta_t + \omega_t \delta t$$

Kinematic state (a.k.a pose):

$$x_t = \begin{bmatrix} x^P_t \\ y^p_t \\ \theta_t \end{bmatrix}$$

Dynamic model:

$$x^p_{t+1} = x^p_t + v_t \cos(\theta_t)\delta t$$
$$y^p_{t+1} = y^p_t + v_t \sin(\theta_t)\delta t$$
$$\theta_{t+1} = \theta_t + \omega_t \delta t$$
$$v_{t+1} = v_t + a_t \delta t$$
$$\omega_{t+1} = \omega_t + \alpha_t \delta t$$

Dynamic state (pose+velocities):

$$x_t = \begin{bmatrix} x^P_t \\ y^p_t \\ \theta_t \\ v_t \\ \omega_t \end{bmatrix}$$

State: All information required to predict trajectory forward given an input signal

State space models:

General nonlinear ss model:

$$x_{t+1} = f(x_t, u_t)$$

state (n x 1) — dynamics — Input (m x 1)

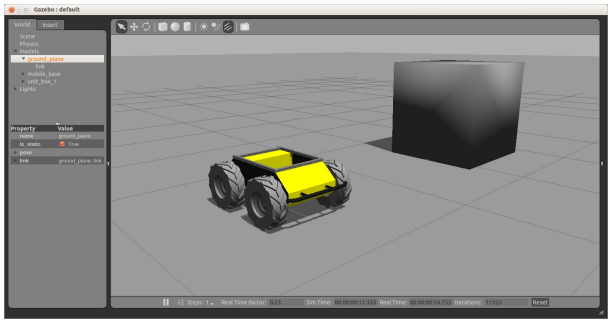
Linear ss model:

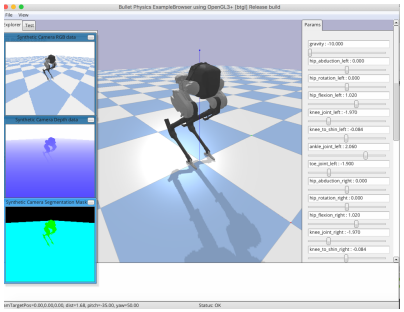
$$x_{t+1} = Ax_t + Bu_t$$

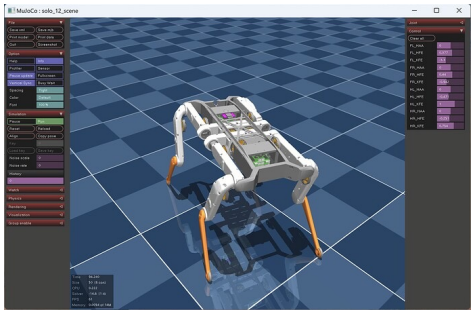
dynamics matrix (n x n) — control matrix (n x m)

Robot Simulators

Purpose: To automate all of this (geometry, collision checking, coordinate transforms, kinematics/dynamics). To scale up to systems too complex to model by hand.







GAZEBO

BULLET PHYSICS LIBRARY

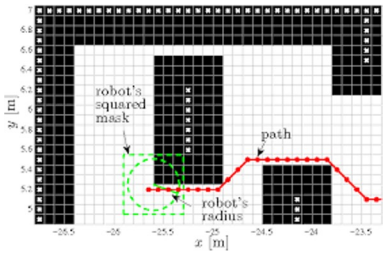
MuJoCo

Advanced physics simulation

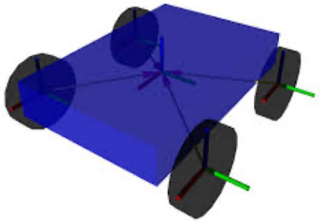
Many, many more: IsaacLab, Drake, Webots, DoJo, TiaChi …

Models: Environment and Robot Representations

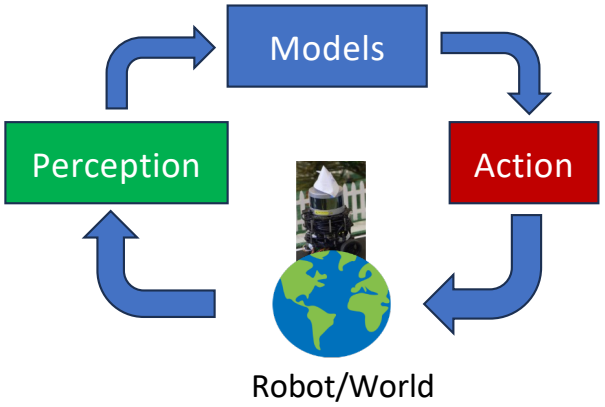
Maps:



Robot models
or simulators:

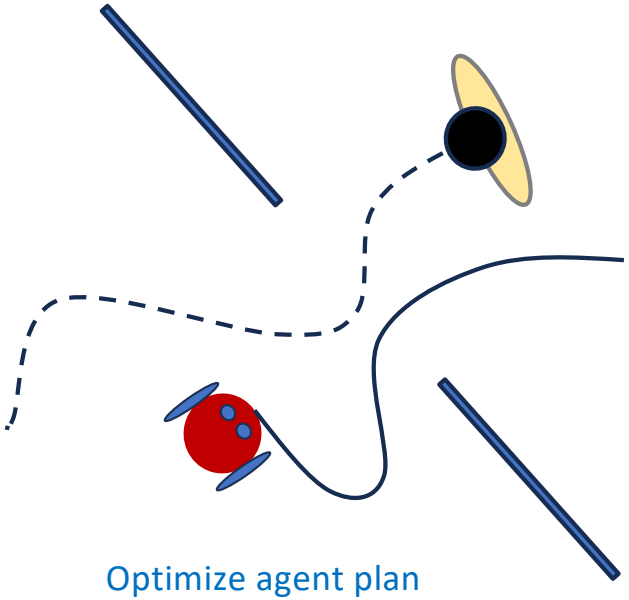


Agent models or
simulators:



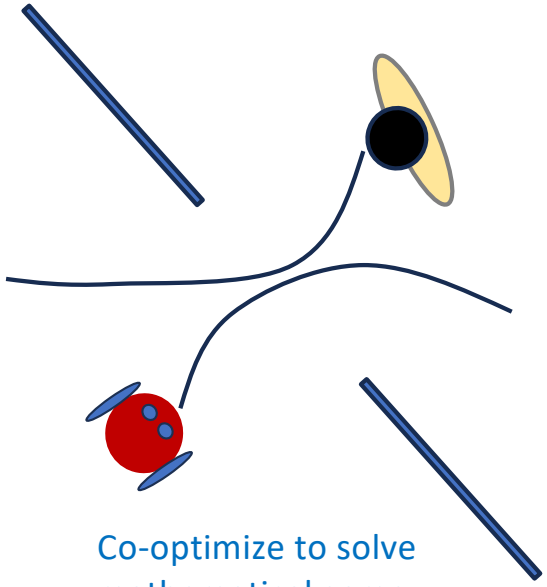
Agent Models

I. Predict then plan



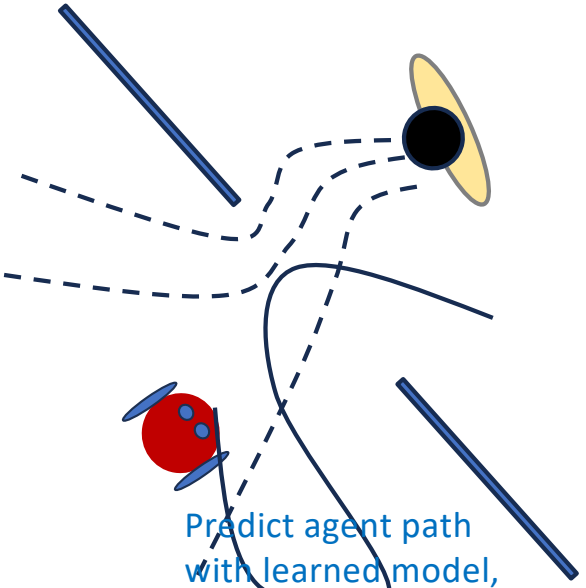
Optimize agent plan first, then robot

II. Game theoretic



Co-optimize to solve mathematical game

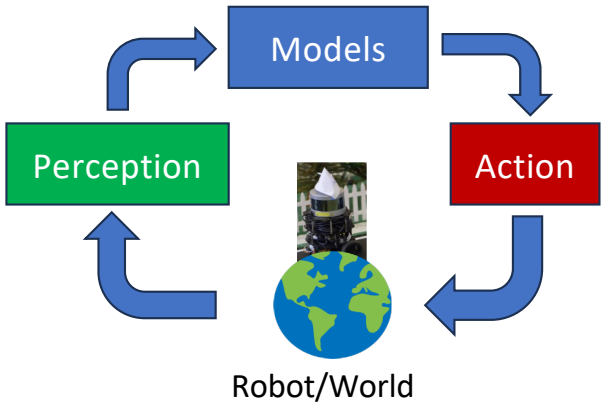
III. Learning based



Predict agent path with learned model, that plan robot

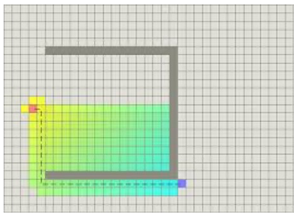
Action: Planning and Control Stack

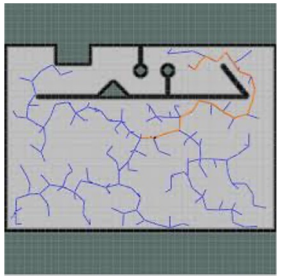
Use the models to move the robot



Path planning:

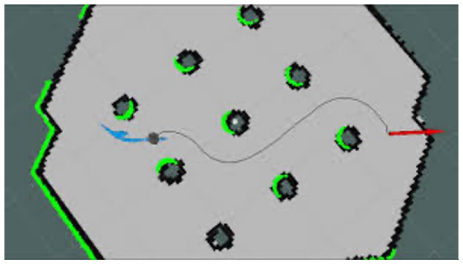
- A* planning on a graph
- RRT, RRT*





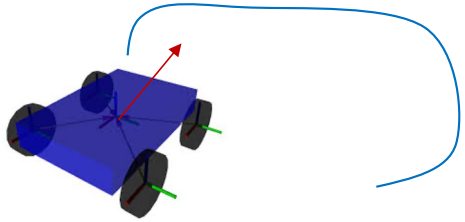
Trajectory optimization:

- Optimal control
- Numerical solvers



Trajectory following:

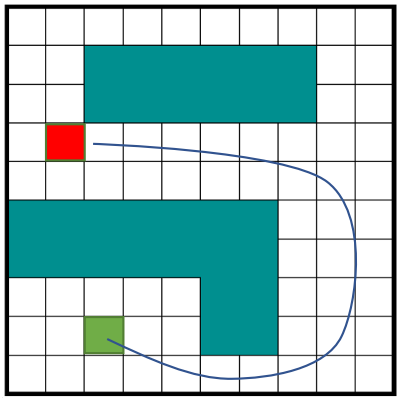
- Kinematic/dynamic models
- PID control
- LQR, gain scheduled LQR



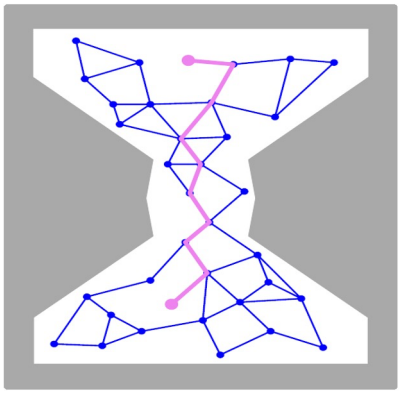
Path Planning

Map representation

Graph search planning, A*

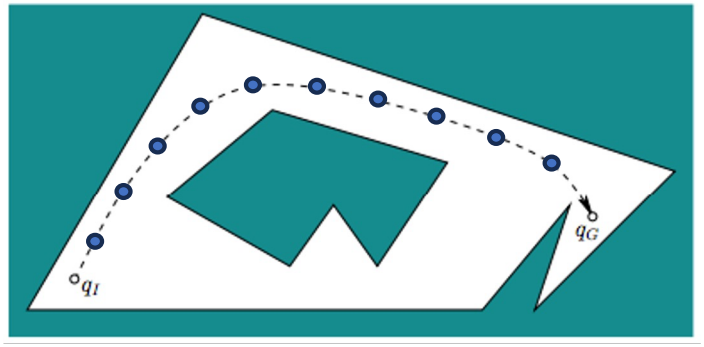


Planning on random graph, RRT*, PRM



Goal: Find a path so the robot doesn't hit things!

Trajectory Optimization



Trajectory optimization problem:

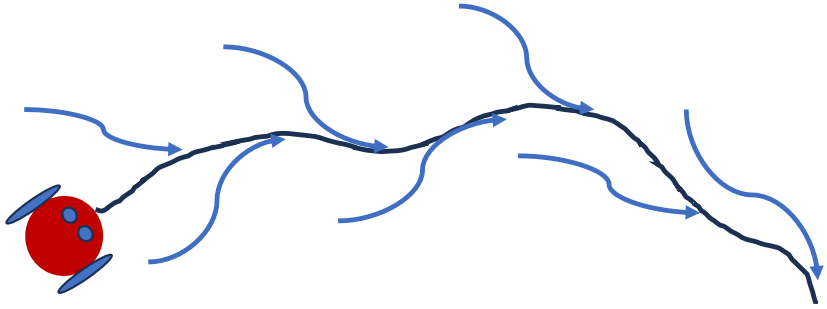
$$\min_{u_{0:T}} \sum_{t=0}^{T} c_t(x_t, u_t)$$ Traj. cost

$$\text{s.t.} \quad x_{t+1} = f(x_t, u_t)$$ Dynamic constraints

$$x_t \in \mathbb{X}_{\text{Free}}$$ Collision constraints

Goal: Find a dynamically feasible trajectory (close to the path) so the robot doesn't hit things!

Trajectory Following



Motion model:

$$x_{t+1} = f(x_t, u_t)$$

Feedback control law:

$$u_t = h(x_t, x_t^*)$$

planned traj waypoints

Goal: Use closed loop feedback control to track desired trajectory, rejecting disturbances and being robust to model errors.

Path Planning vs Trajectory Planning

A* path plan:

(i) Global search

(ii) Collision free

(iii) Not dynamically feasible

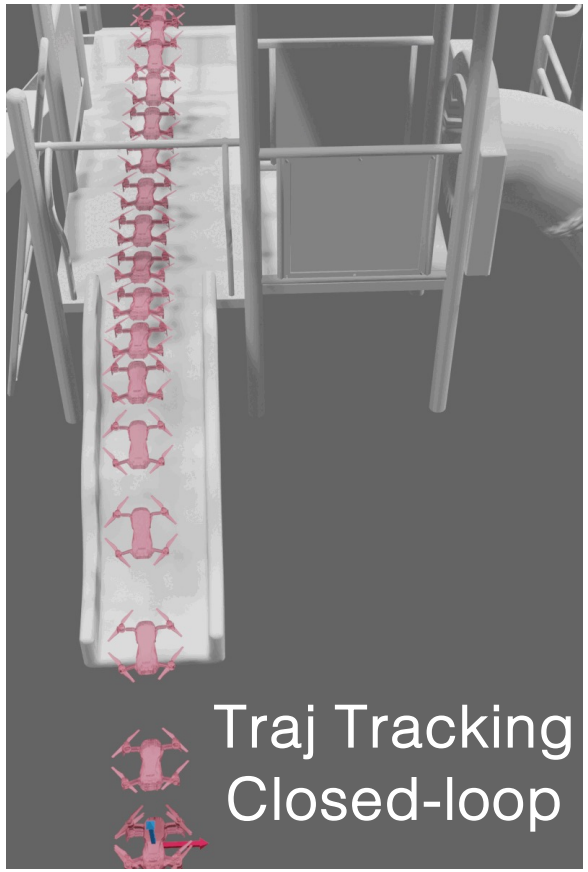
Traj Opt

(i) Local search

(ii) Collision free

(iii) Dynamically feasible

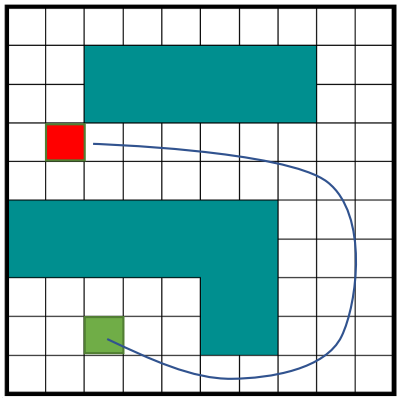
Trajectory optimization vs Trajectory following



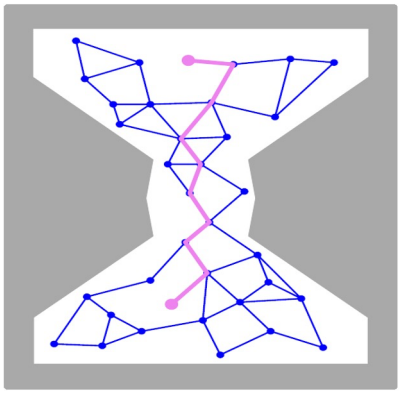
Path Planning

Map representation

Graph search planning



Planning on random graph



Goal: Find a path so the robot doesn't hit things!

History

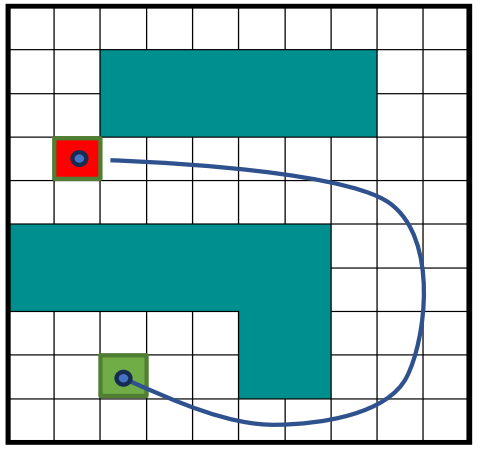
- *Potential fields* [Khatib ‘85, Rimon & Koditschek ‘92]: create forces on the robot that pull it toward the goal and push it away from obstacles
- *Grid-based planning* [Stentz ‘94]: discretizes problem into grid and runs a graph-search algorithm (Dijkstra, A*, …)
- *Combinatorial planning* [LaValle ‘06]: constructs structures in the configuration (C-) space that completely capture all information needed for planning
- *Sampling-based planning* [Kavraki et al ‘96, LaValle & Kuffner ‘06, Karaman and Frazzoli, ‘11]: uses collision detection algorithms to probe and incrementally search the C-space for a solution, rather than completely characterizing all of the C_{free} structure. Asymptotic optimality with RRT*/PRM*.
- Current research: inclusion of differential and logical constraints, planning under uncertainty, parallel implementation, deep learning warm starting

Graph search planning

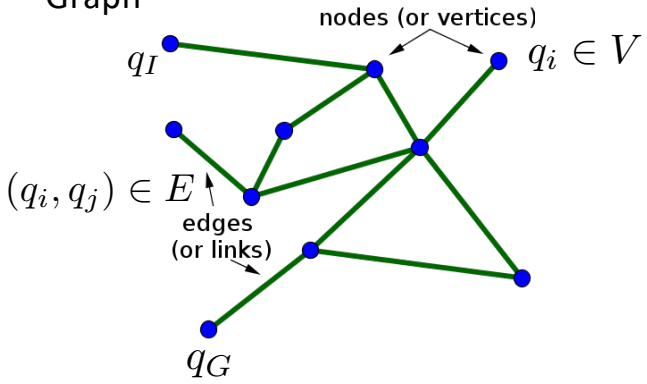
- Discretize robot's **configuration space** into a grid
  - Each grid cell is either free or occupied
  - Robot moves between adjacent free cells
  - **Goal:** find shortest sequence of free cells from start to goal

- Mathematically, this corresponds to pathfinding in a discrete graph $G = (V, E)$
  - Each vertex $q_i \in V$ represents a free cell
  - Edges $(q_i, q_j) \in E$ connect adjacent grid cells

Map

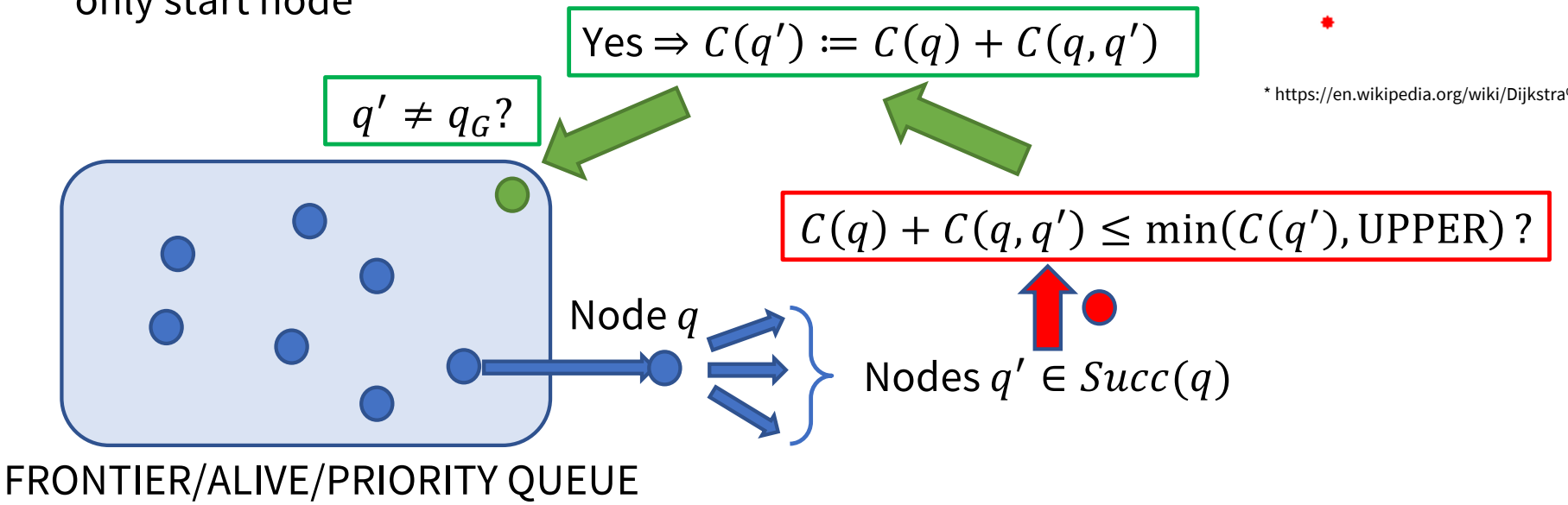


Graph



Graph search algorithms

- Having determined decomposition, how to find "best" path?
- **Label-Correcting Algorithms**: $C(q)$: *cost-to-come* from q_I to q
- Set to ∞ for all nodes, except start node to 0, frontier queue has only start node



* https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Label correcting algorithm

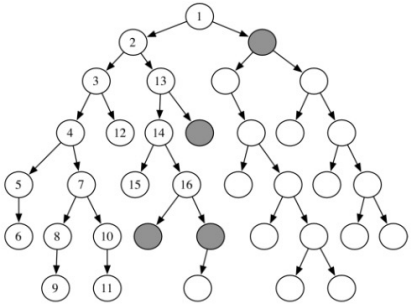
Initialization: set the labels of all nodes to ∞, except for the label of the origin node, which is set to 0, frontier queue has only start node.

Step 1. Remove a node q from frontier queue Q and for each child q' of q, execute step 2

Step 2. If $C(q) + C(q, q') \leq \min(C(q'), \text{UPPER})$, set $C(q') := C(q) + C(q, q')$ and set q to be the parent of q'. In addition, if $q' \neq q_G$, place q' in the frontier queue if it is not already there, while if $q' = q_G$, set UPPER to the new value $C(q) + C(q, q_G)$

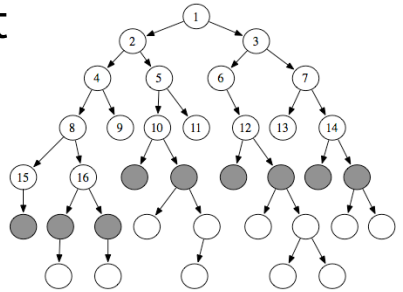
Step 3. If the frontier queue is empty, terminate, else go to step 1

How to prioritize the queue?



Depth-First-Search (DFS): Maintain Q as a **stack** – Last in/first out

- Lower memory requirement (only need to store part of graph)



Breadth-First-Search (BFS, Bellman-Ford): Maintain Q as a **list** – First in/first out

- Update cost for all edges up to current depth before proceeding to greater depth
- Can deal with negative edge (transition) costs

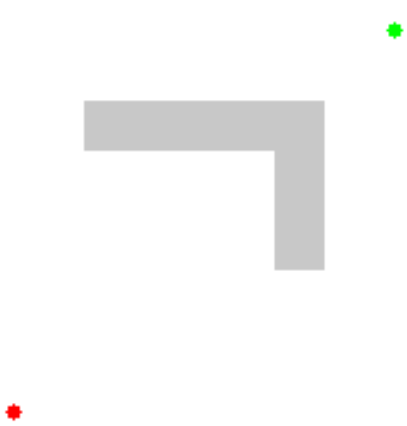
Best-First (BF, Dijkstra): Greedily select next q: $q = \operatorname{argmin}_{q \in Q} C(q)$

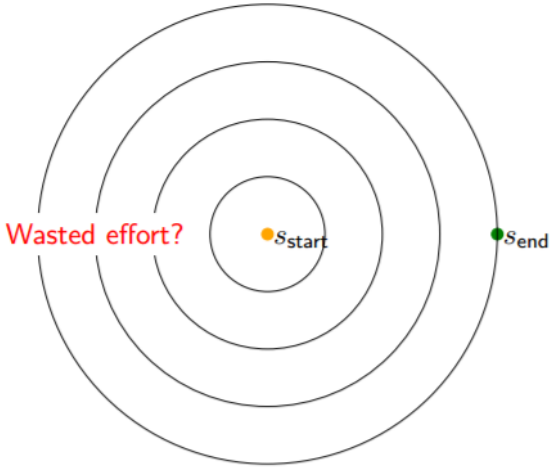
- Node will enter the frontier queue at most *once*
- Requires costs to be non-negative

Correctness and improvements

Theorem

If a feasible path exists from q_I to q_G, then algorithm terminates in finite time with $C(q_G)$ equal to the optimal cost of traversal, $C^*(q_G)$.





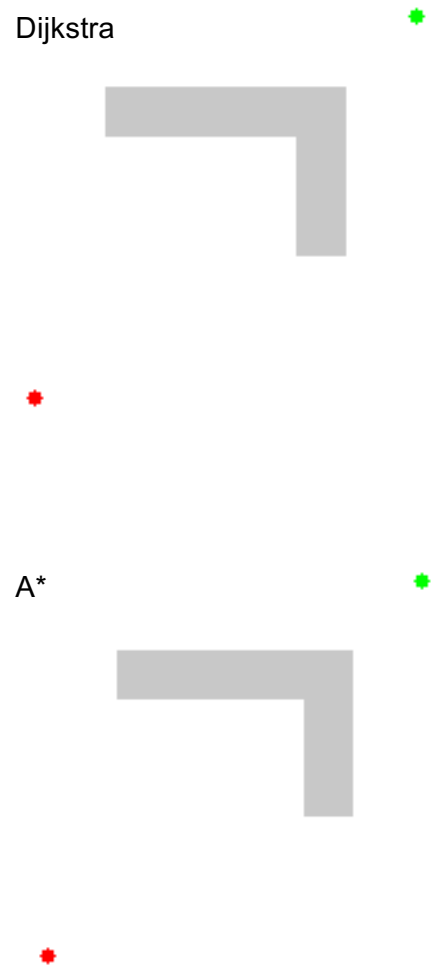
A*: Improving Dijkstra

- Dijkstra orders by optimal "*cost-to-come*"
- Faster results if order by "*cost-to-come*"+ *(approximate) "cost-to-go"*
- Take the node with lowest C(q) + h(q) from frontier set, where $h(q)$ is *a* heuristic for optimal cost-to-go (specifically, a positive *underestimate*)
- When testing to add neighbor into frontier set, use stronger

$$C(q) + C(q, q') + h(q') \leq \text{UPPER}$$

Instead of $C(q) + C(q, q') \leq \text{UPPER}$

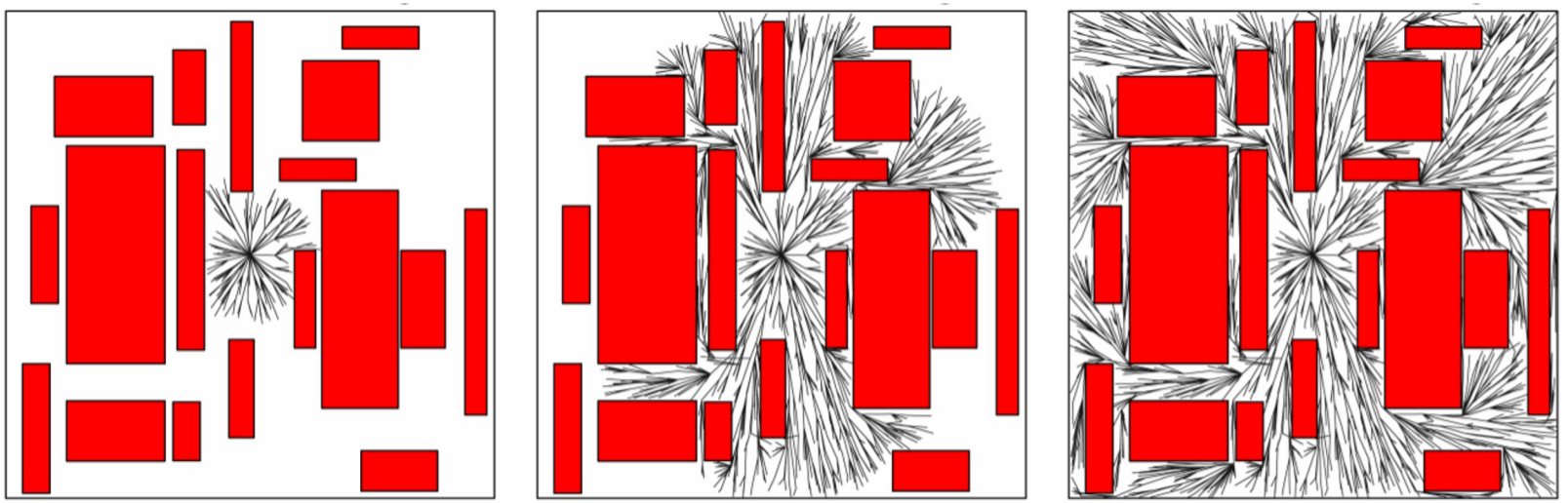
- In this way, fewer nodes will be placed in the frontier queue
- This modification still guarantees that the algorithm will terminate with a shortest path



Graph search approaches: summary

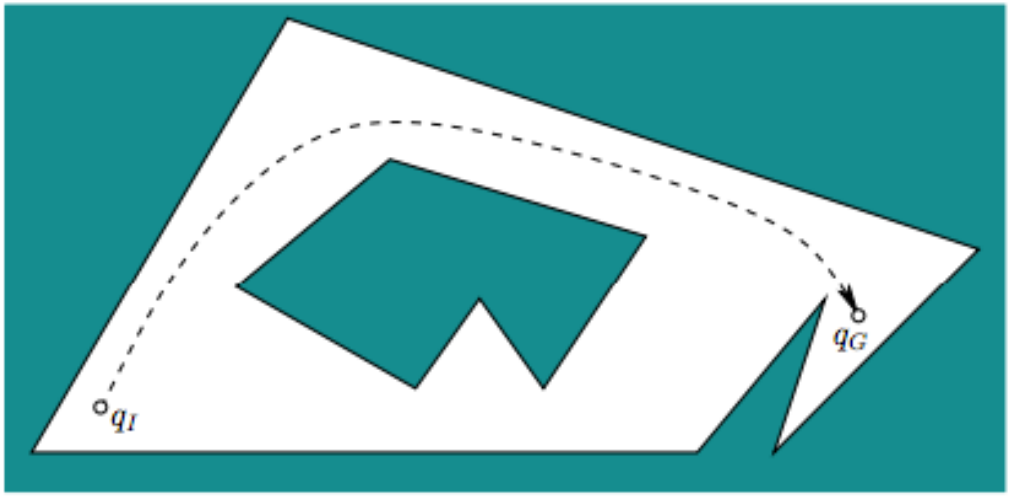
- Pros:
  - Simple and easy to use
  - Fast (for some problems)
- Cons:
  - Resolution dependent
    - Not guaranteed to find solution if grid resolution is not small enough
  - Limited to simple robots
    - C-space grid size is exponential in the number of DOFs

Next time: sampling-based planning, RRT, RRT*, PRM



Combinatorial planning

Key idea: compute a roadmap, which is a graph in which each vertex is a configuration in C_{free} and each edge is a path through C_{free} that connects a pair of vertices



Free-space roadmaps

Given a complete representation of the free space, we compute a roadmap that captures its connectivity

A roadmap should preserve:

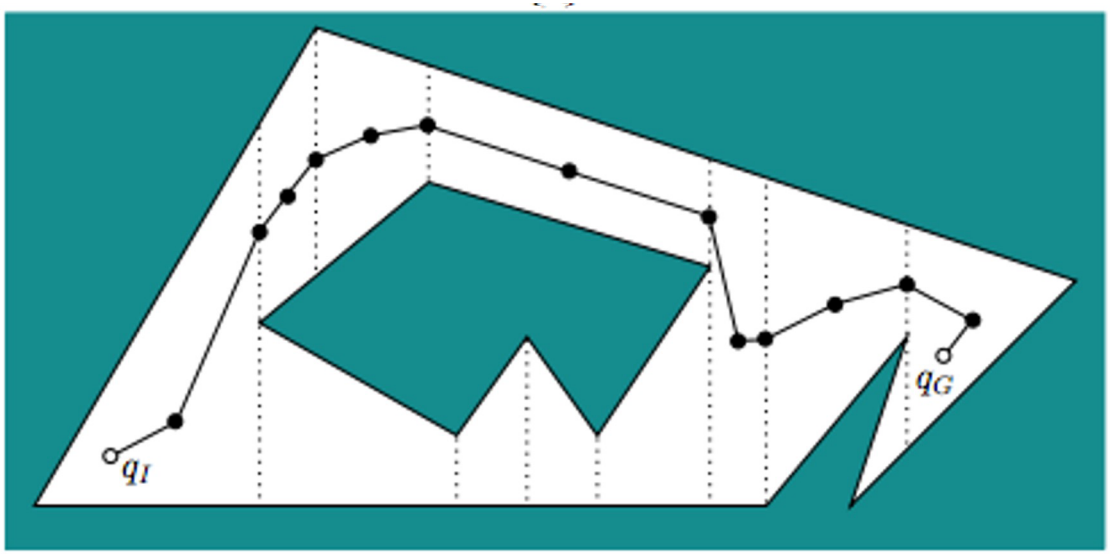
1. Accessibility: it is always possible to connect some q to the roadmap (e.g., $q_I \rightarrow s_1, q_G \rightarrow s_2$)
2. Connectivity: if there exists a path from q_I to q_G, there exists a path on the roadmap from s_1 to s_2

Main point: a roadmap provides a discrete representation of the continuous motion planning problem *without losing* any of the original connectivity information needed to solve it

Cell decomposition

Typical approach: cell decomposition. General requirements:

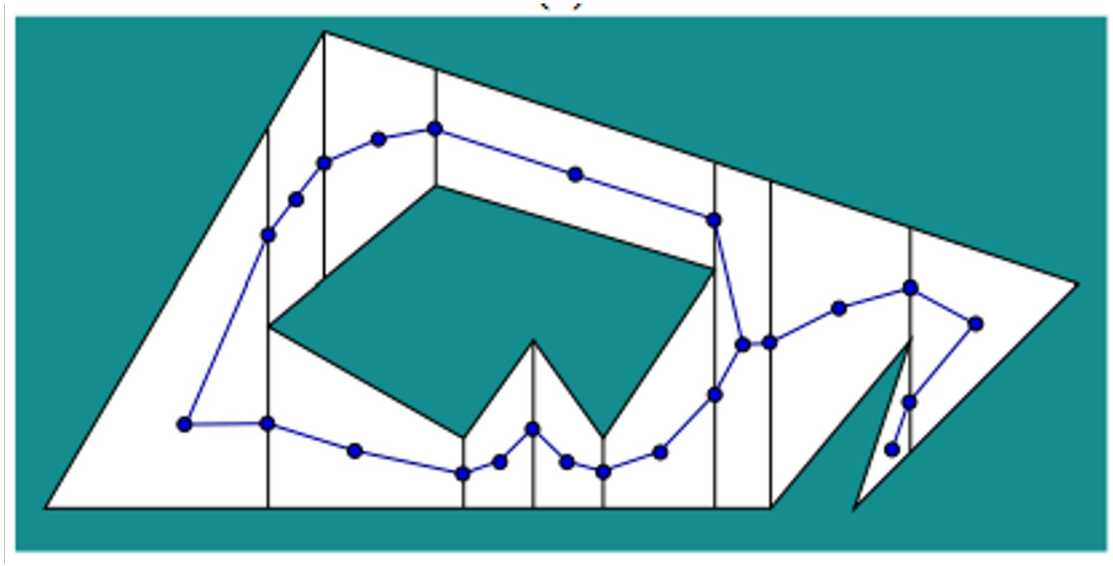
- Decomposition should be easy to compute
- Each cell should be easy to traverse (ideally convex)
- Adjacencies between cells should be straightforward to determine



Computing a trapezoidal cell decomposition

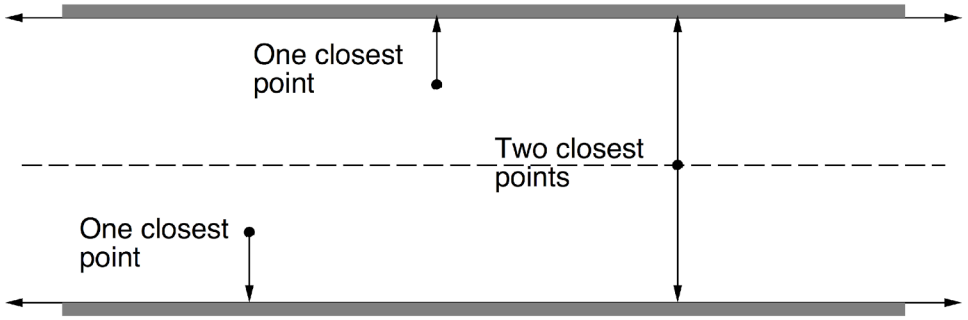
For every vertex (corner) of the forbidden space:

- Extend a vertical ray until it hits the first edge from top and bottom
  - Compute intersection points with all edges, and take the closest ones
  - More efficient approaches exists

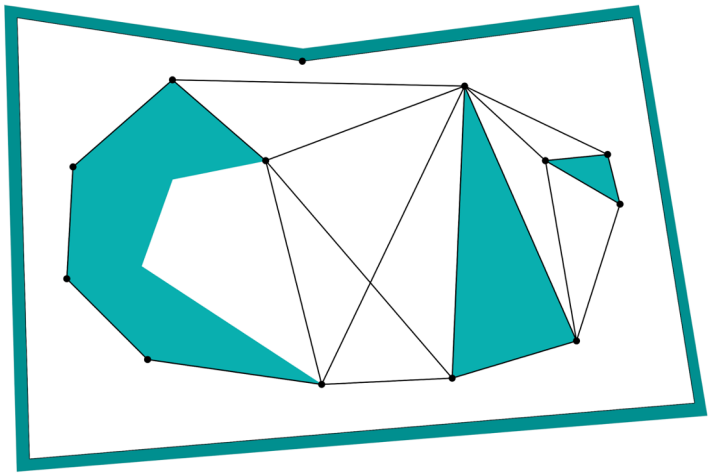


Other roadmaps

Maximum clearance (medial axis)

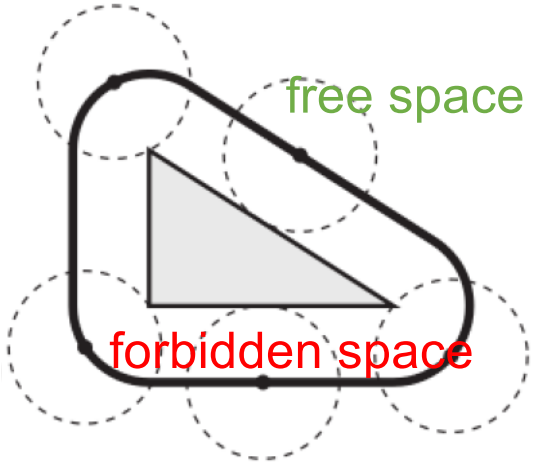


Minimum distance (visibility graph)



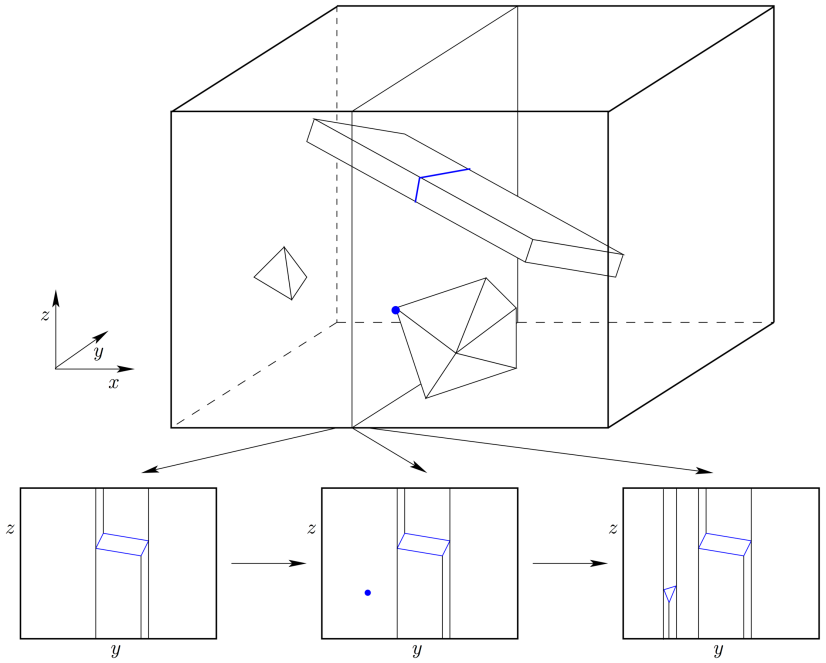
Caveat: free-space computation

- The free space is not known in advance
- We need to compute this space given the ingredients
  - Robot representation, i.e., its shape (polygon, polyhedron, …)
  - Representation of obstacles
- To achieve this, we do the following:
  - Contract the robot into a point
  - In return, inflate (or stretch) obstacles by the shape of the robots



Higher dimensions

- Extensions to higher dimensions is challenging ⇒ algebraic decomposition methods



Combinatorial planning: summary

- These approaches are complete and even optimal in some cases
  - Do not discretize or approximate the problem
- Have theoretical guarantees on the running time
  - I.e., computational complexity is known
- Usually limited to small number of DOFs
  - Computationally intractable for many problems
- Problem specific: each algorithm applies to a specific type of robot/problem
- Difficult to implement; requires special software to reason about geometric data structures (CGAL)

Additional resources on combinatorial planning

- Visualization of C-space for polygonal robot: https://www.youtube.com/watch?v=SBFwgR4K1Gk
- Algorithmic details for Minkowski sums and trapezoidal decomposition: de Berg et al., “Computational geometry: algorithms and applications”, 2008
- Implementation in C++: Computational Geometry Algorithms Library

CGAL