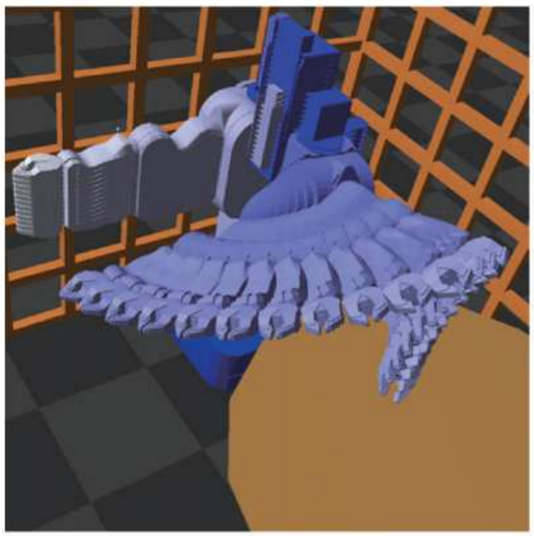
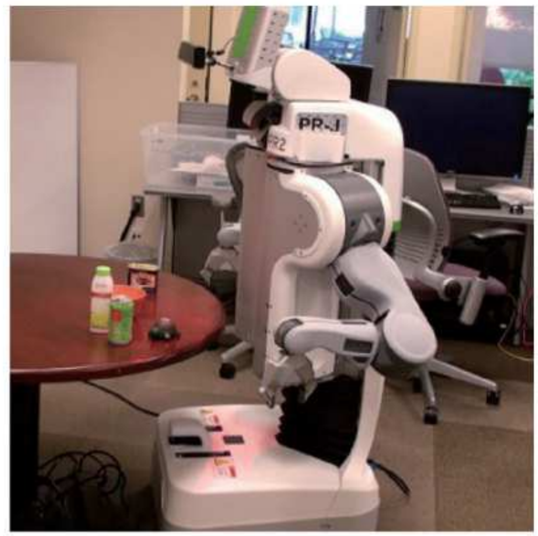
CS182: Artificial Intelligence

Lecture 12: Robot Motion Planning I





Brian Plancher
Harvard University
Fall 2018

Slides adapted from
Scott Kuindersma

Announcements

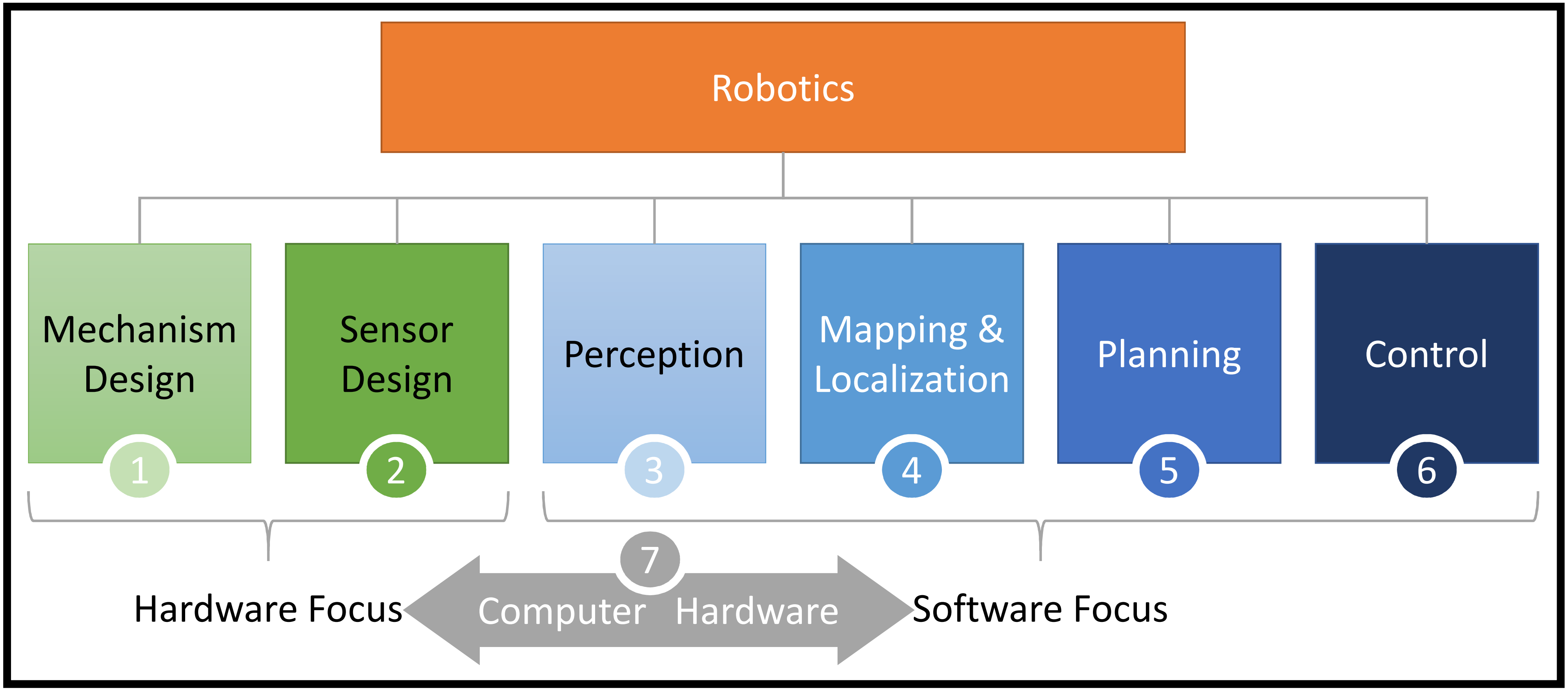
- **Please submit your homework in the correct places** – and you check that your grade moves to Canvas when I announce grades are out. At this point in the semester you will start losing points / getting 0s if you don't do this correctly so please be careful…
- **Midterm 1 is a week from Monday and covers L1-L11, P1-P3, S1-S6**
  - Next week's section will become **midterm review** – time TBD most likely later in the week / over the weekend and longer
- The Robotics material from today and Monday will be on Midterm 2 (next Wednesday's guest lecture will have a problem on P4) so come!

Announcements

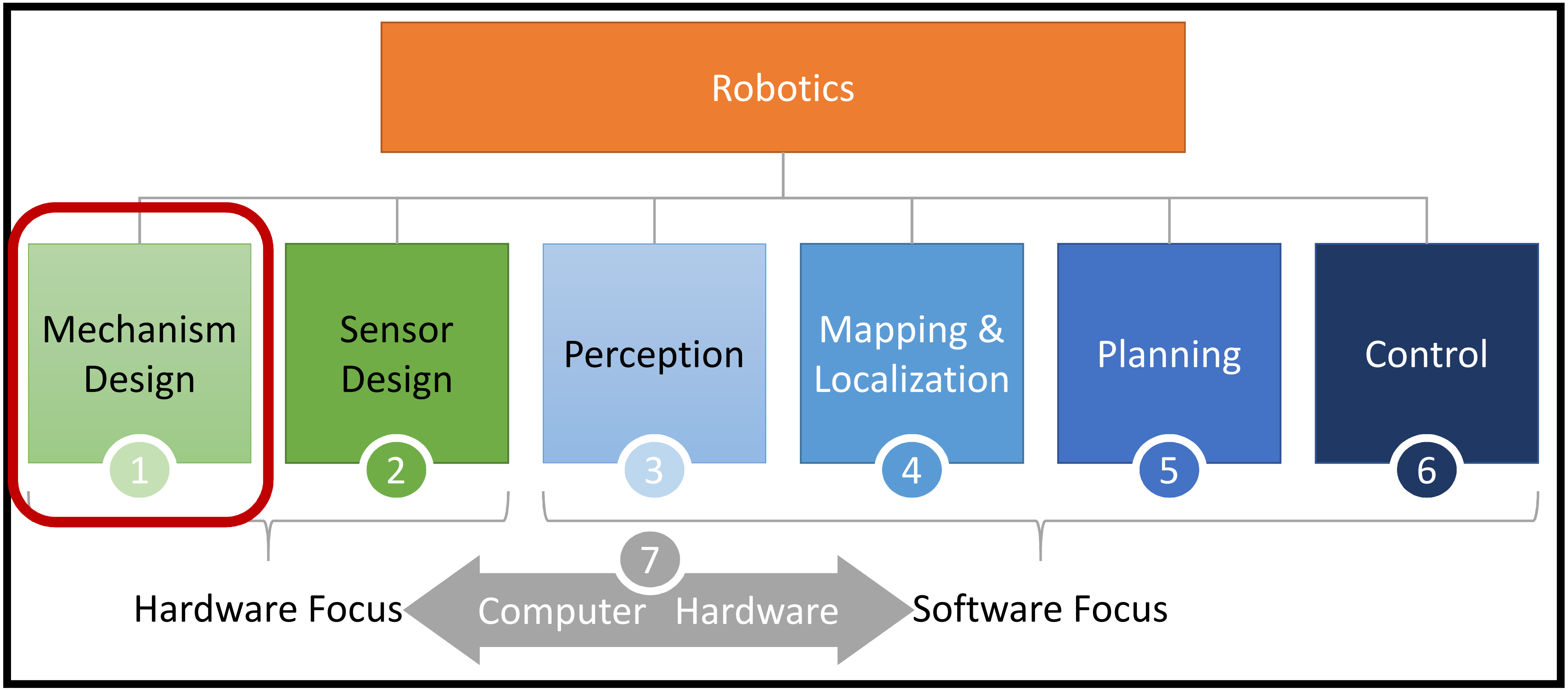
- **Please submit your homework in the correct places** – and you check that your grade moves to Canvas when I announce grades are out. At this point in the semester you will start losing points / getting 0s if you don't do
- **Midterm 1 is** **P1-P3, S1-S6**
  - Next wee ime TBD most likely later in the week / over the weekend and longer
- The Robotics material from today and Monday will be on Midterm 2 (next Wednesday's guest lecture will have a problem on P4) so come!

Let me know if you have feedback from class today and I can try to incorporate that for Monday!

Robotics is a **BIG** space



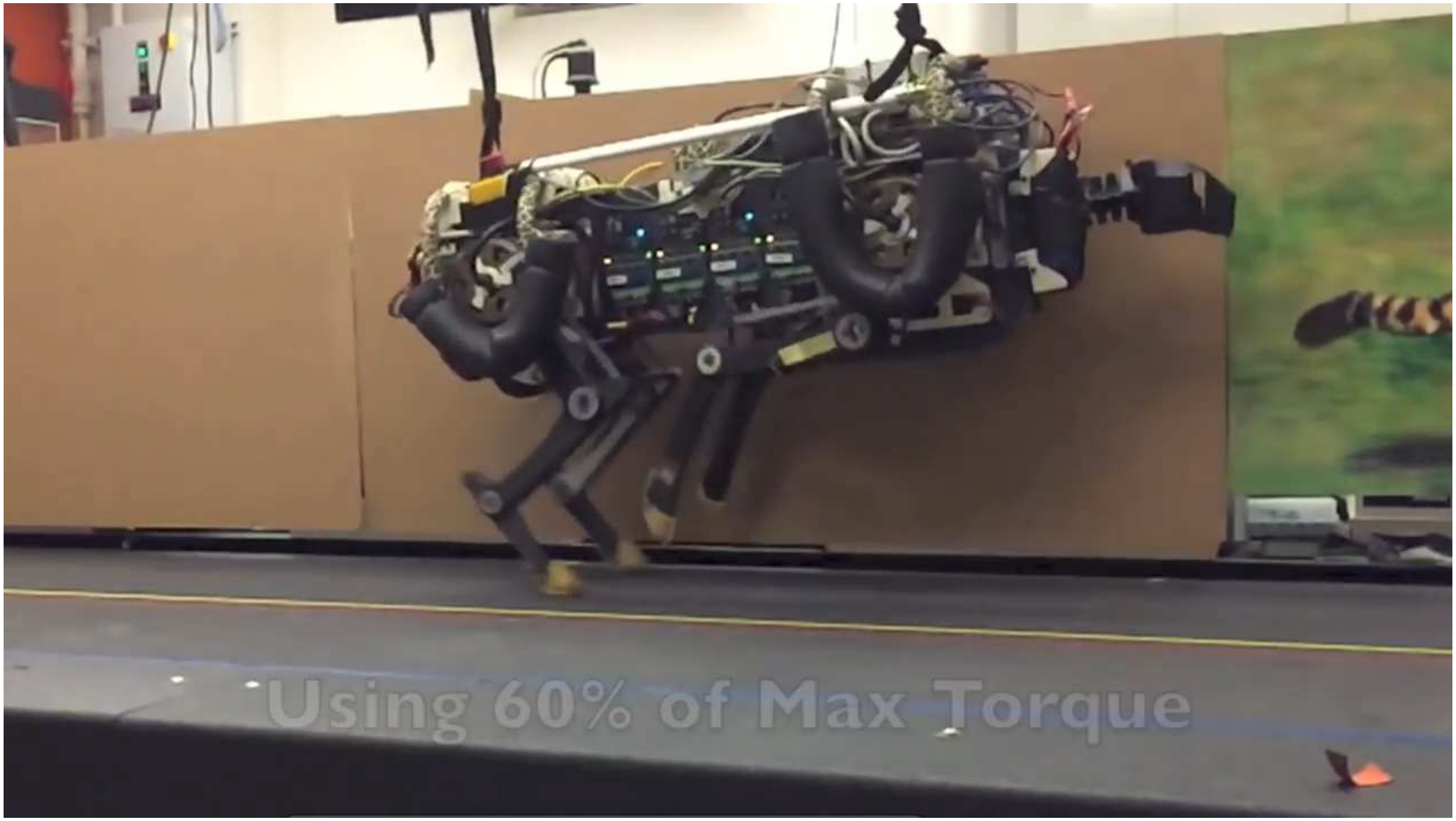
Robotics is a **BIG** space



1

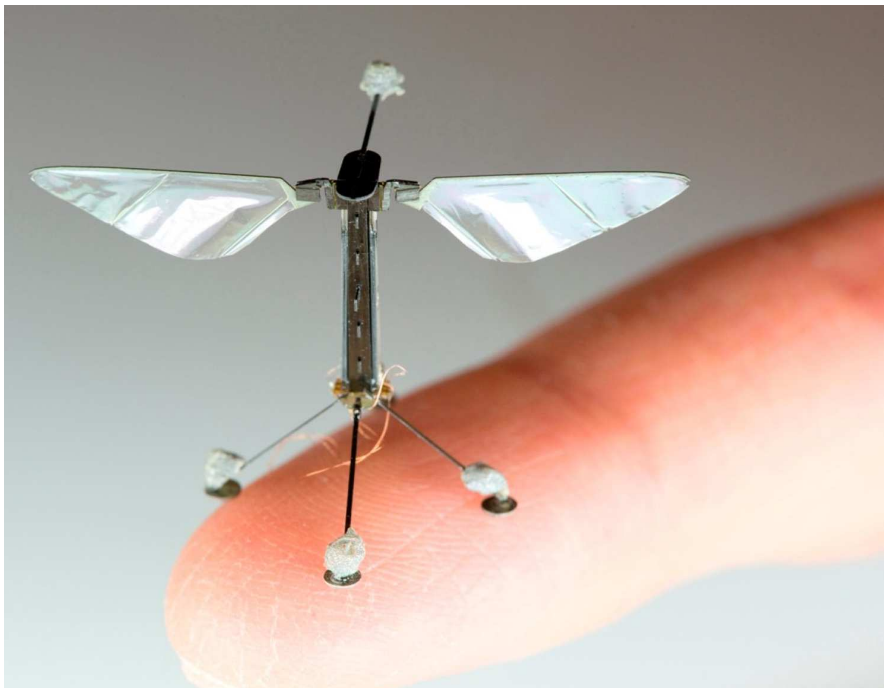
Mechanism designers create new robots and actuators

MIT 2.74



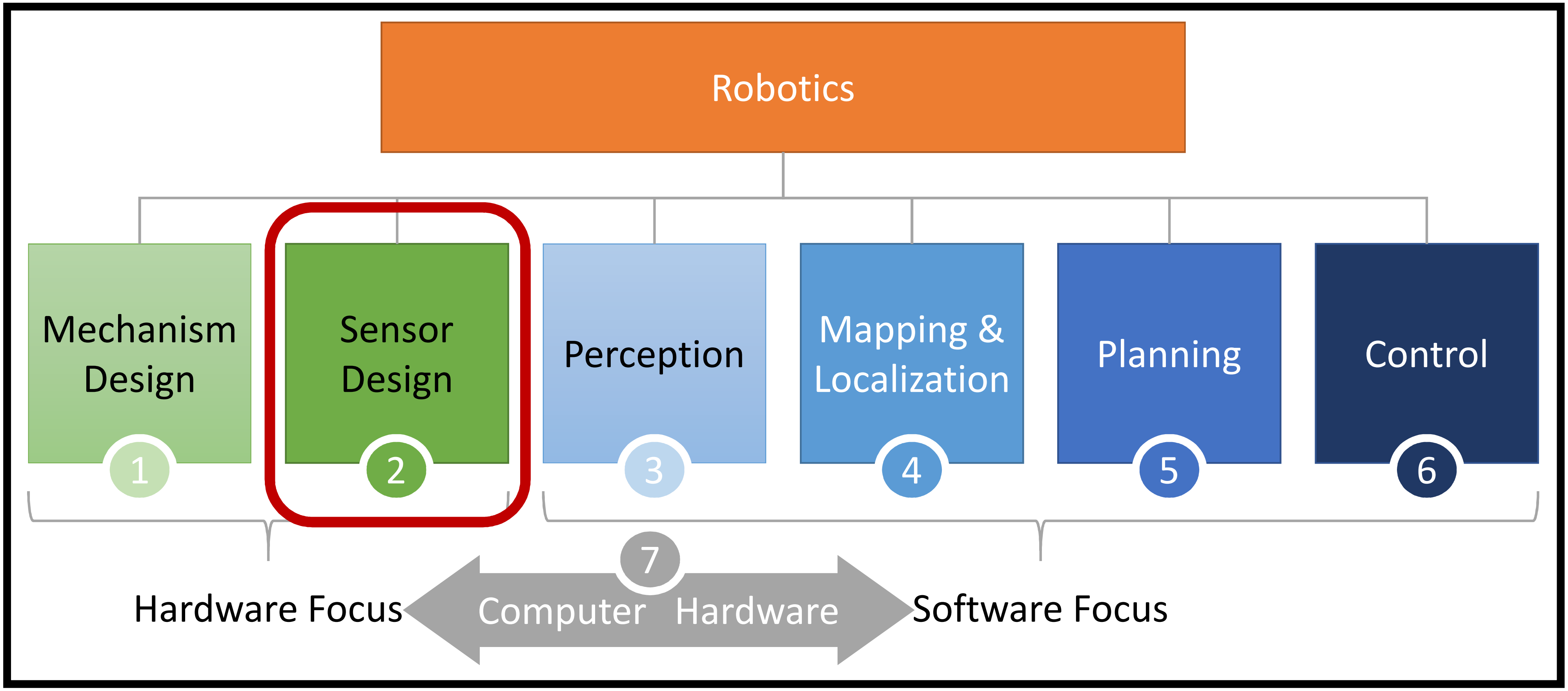
1

Mechanism designers create new robots and actuators



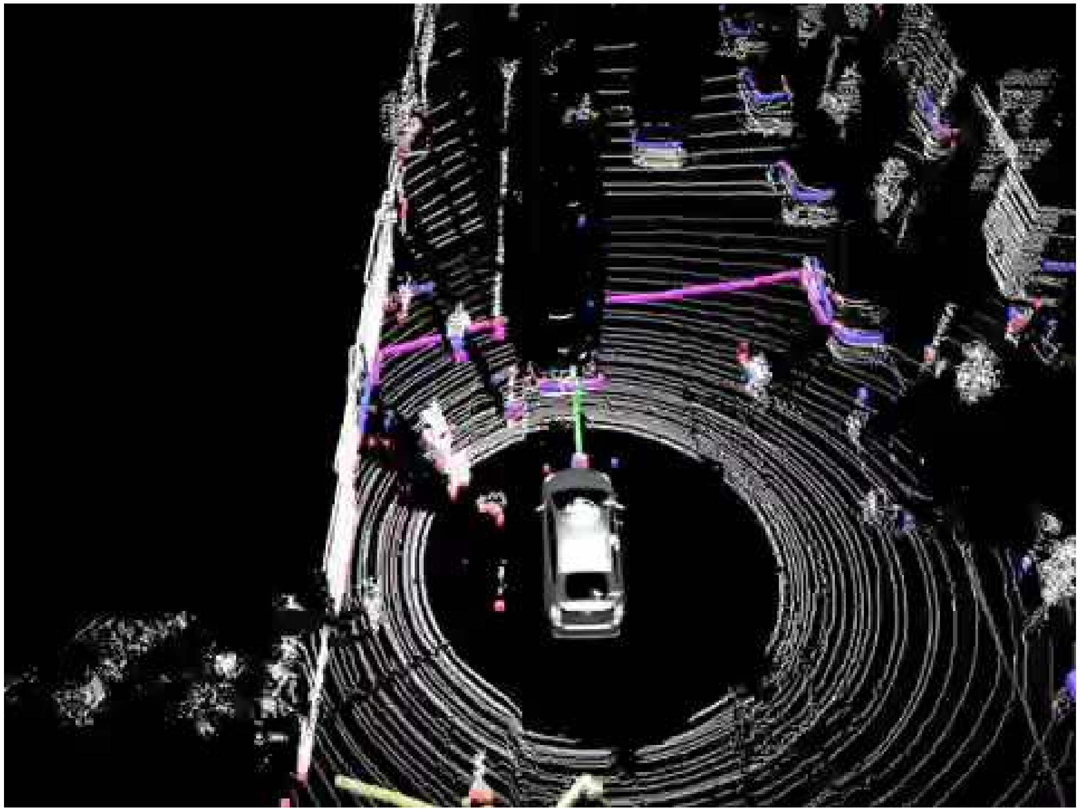
WYSS INSTITUTE

Robotics is a **BIG** space

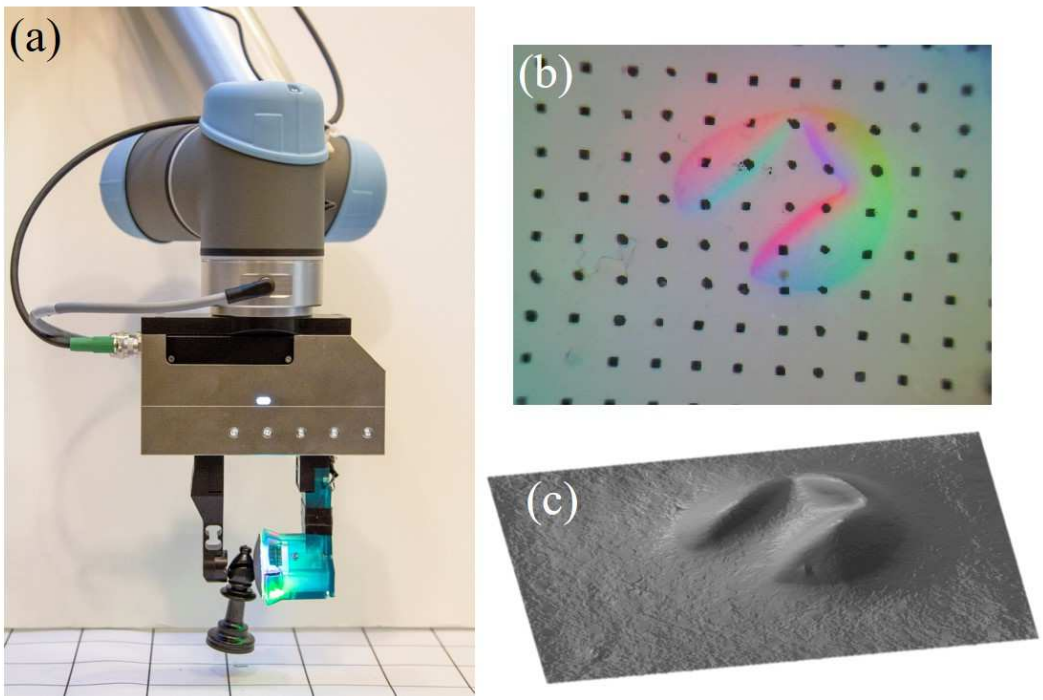


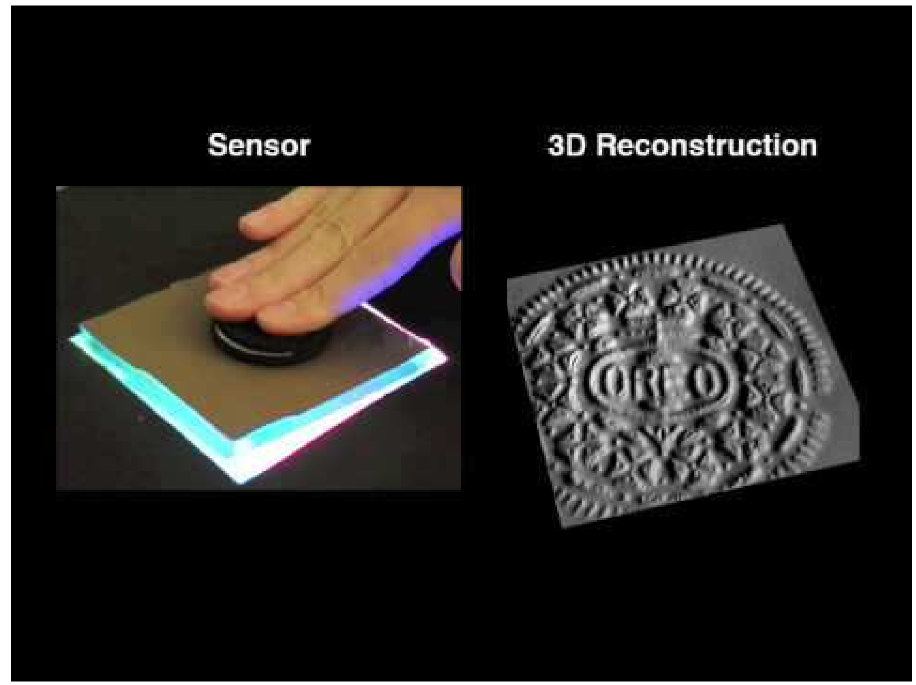
2

Sensor designers try to find new ways to collect data about the world around the robot



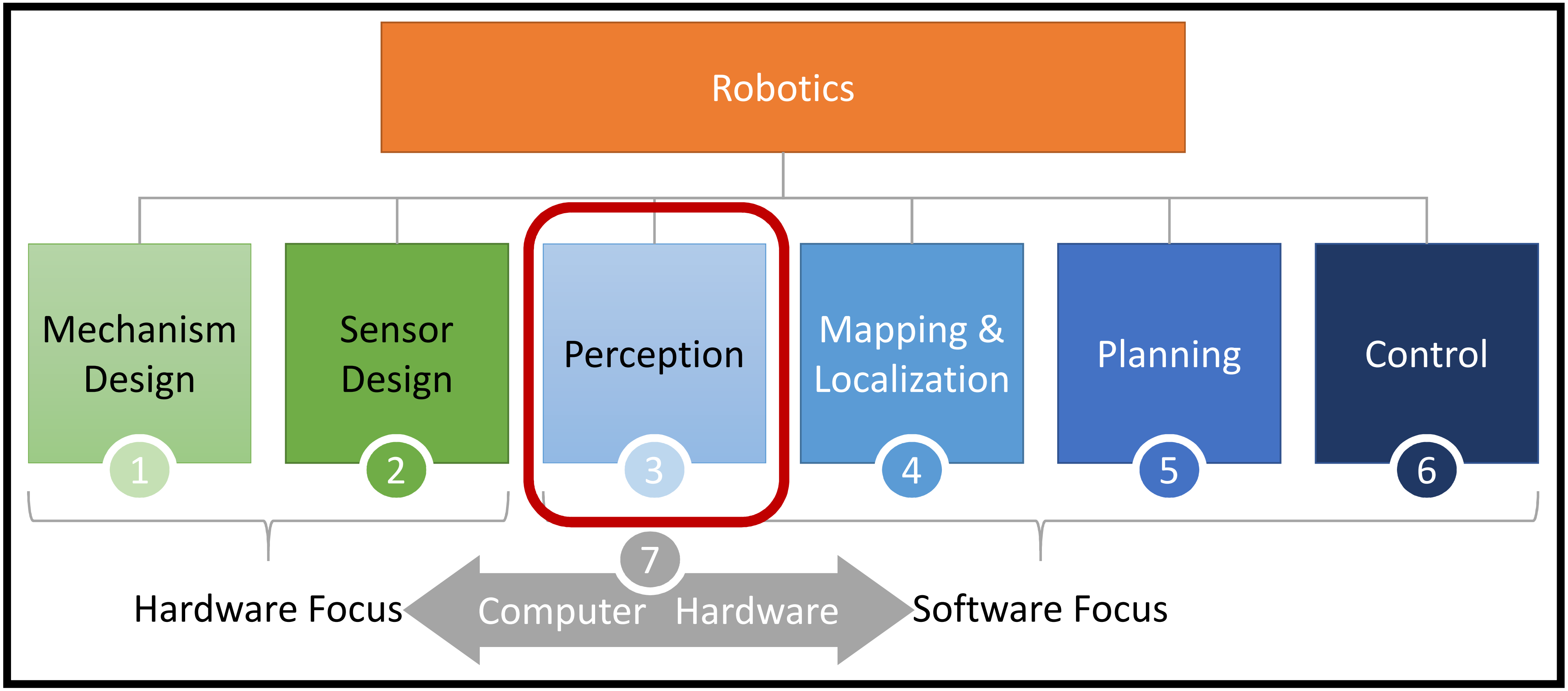
2 Sensor designers try to find new ways to collect data about the world around the robot





http://www.gelsight.com/

Robotics is a **BIG** space



3

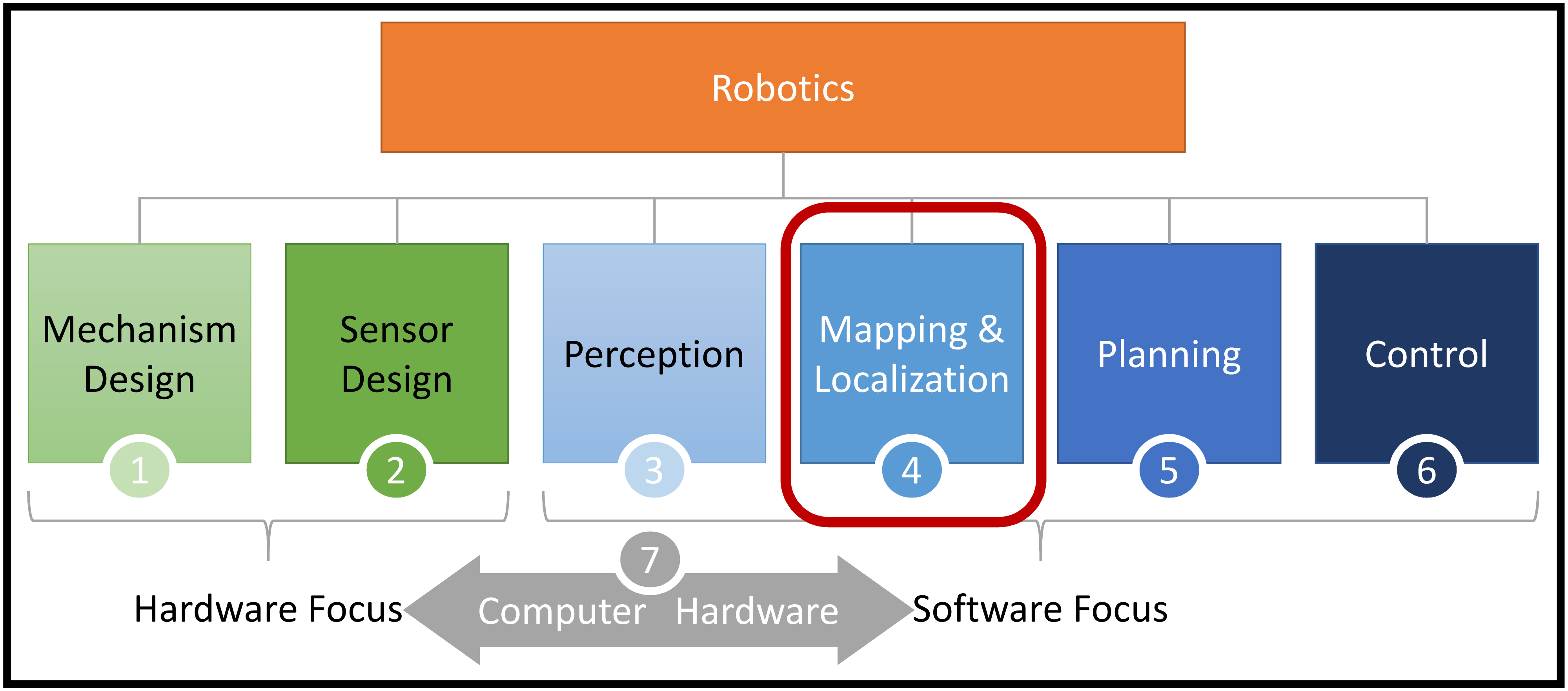
Perception is the processing of sensor data to understand the world around the robot

Fig. 7: *PowderSkier* (top left) mean shifted (top right) with and clustered (bottom left) with $(h_s, h_r, M) = (12, 8, 20)$ and post processed (bottom right).

CS 283

Perception is the processing of sensor data to understand the world around the robot

Robotics is a **BIG** space



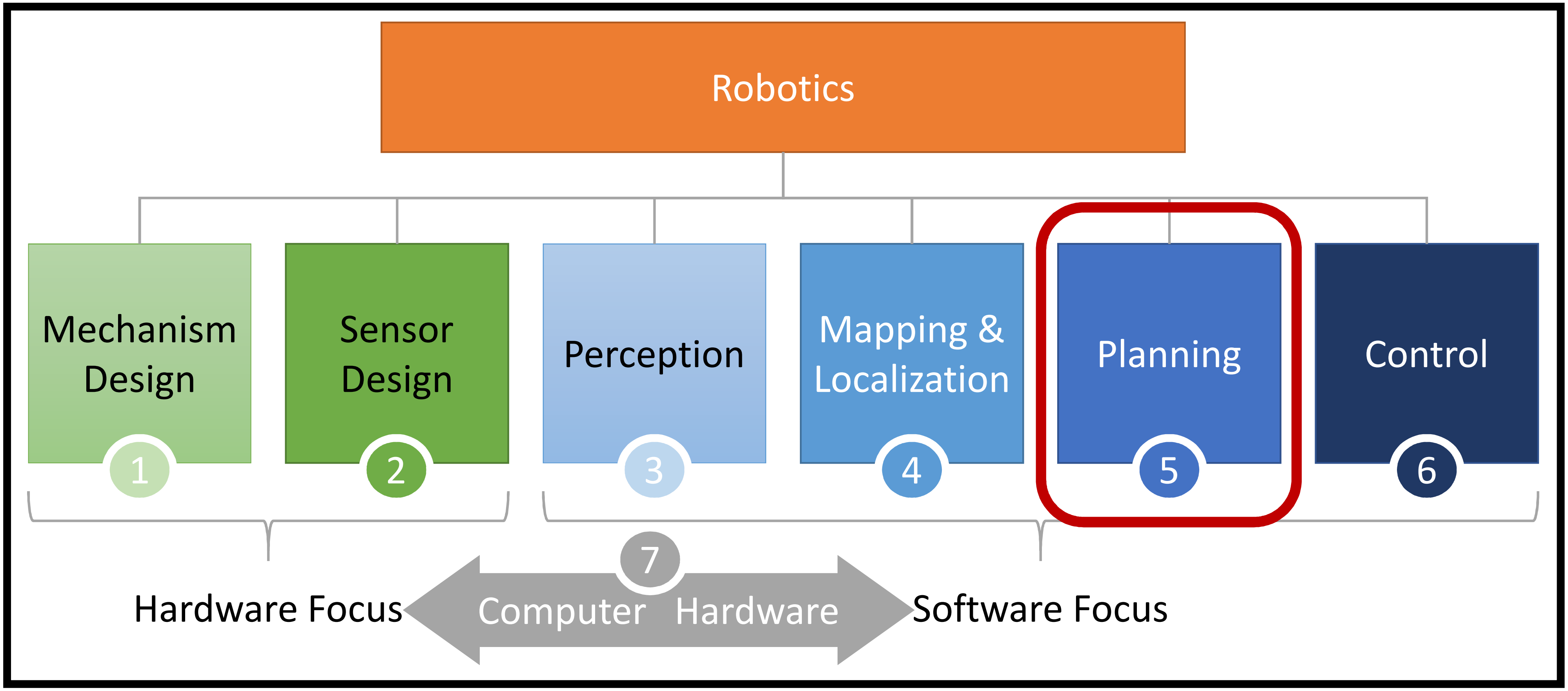
4

Mapping & Localization is the process of using sensor data to understand where a robot is in the world

4 Mapping & Localization is the process of using sensor data to understand where a robot is in the world

We will talk about particle filtering (a technique used to do this) later in the course! (HMMs)

Robotics is a **BIG** space



5

Planning is the process of computing an action plan for a robot based on the previously computed information

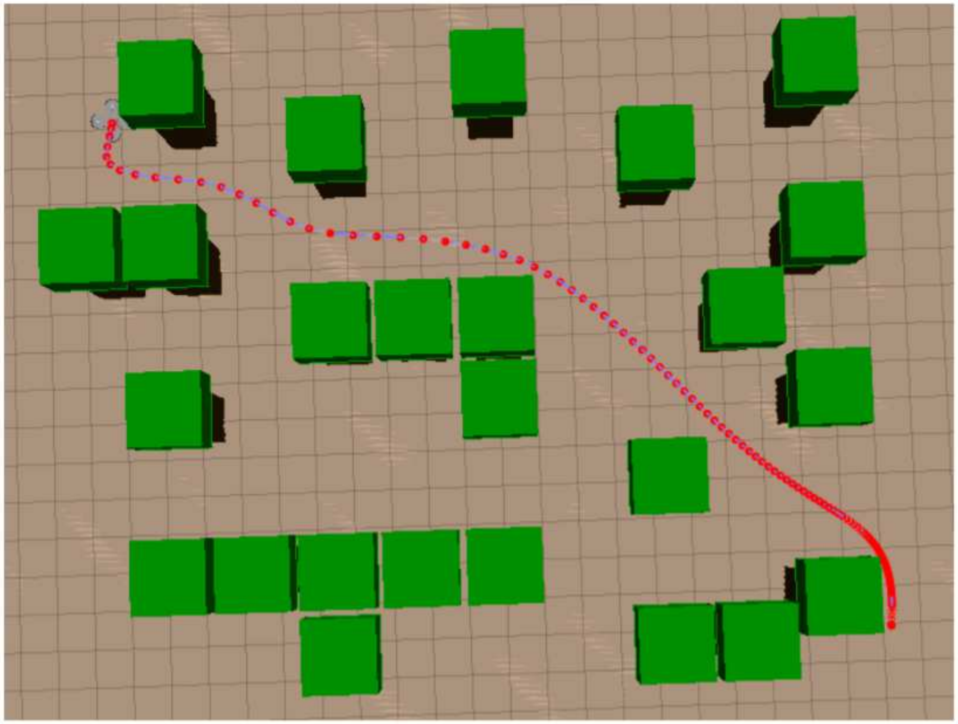
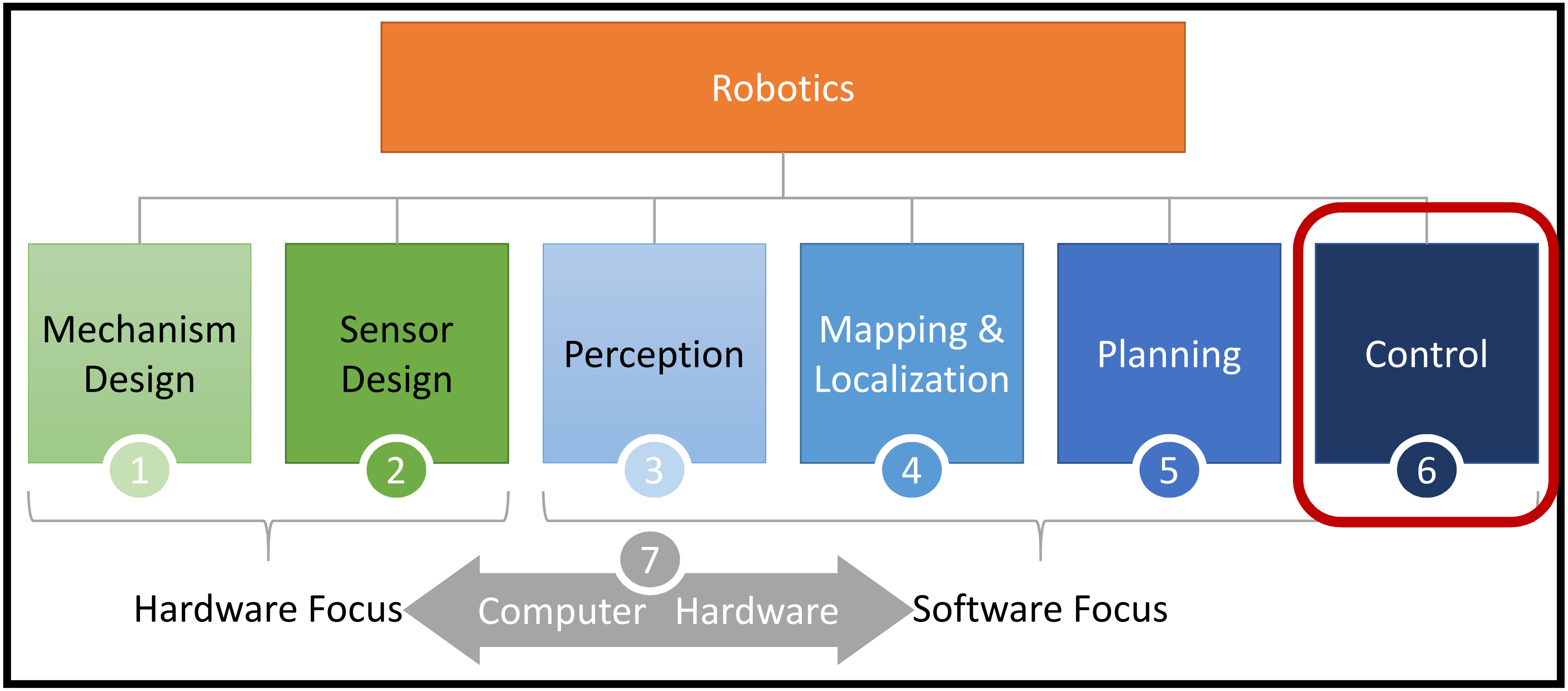
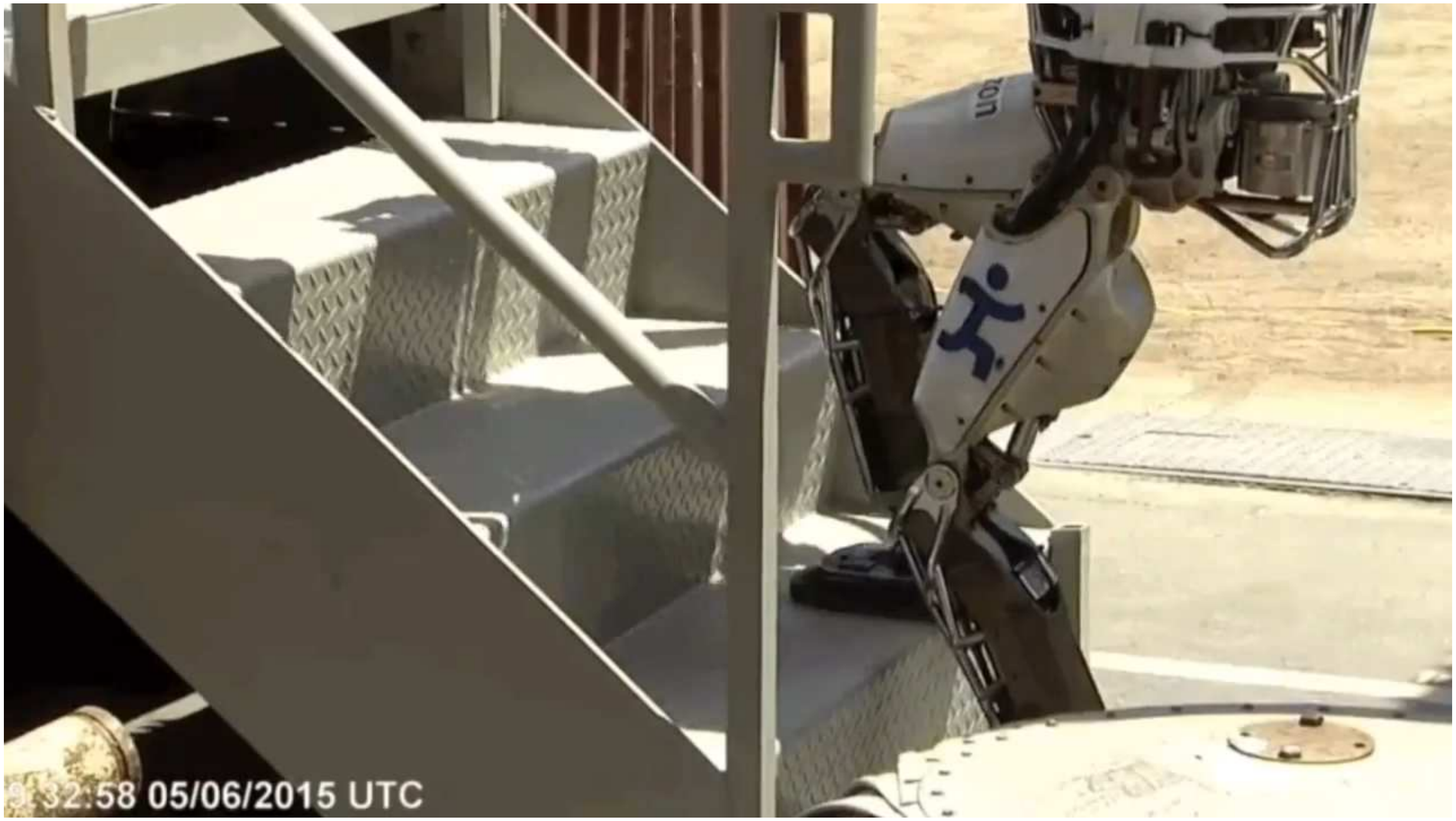


Fig. 3. Collision-free quadrotor trajectory computed by constrained UDP.

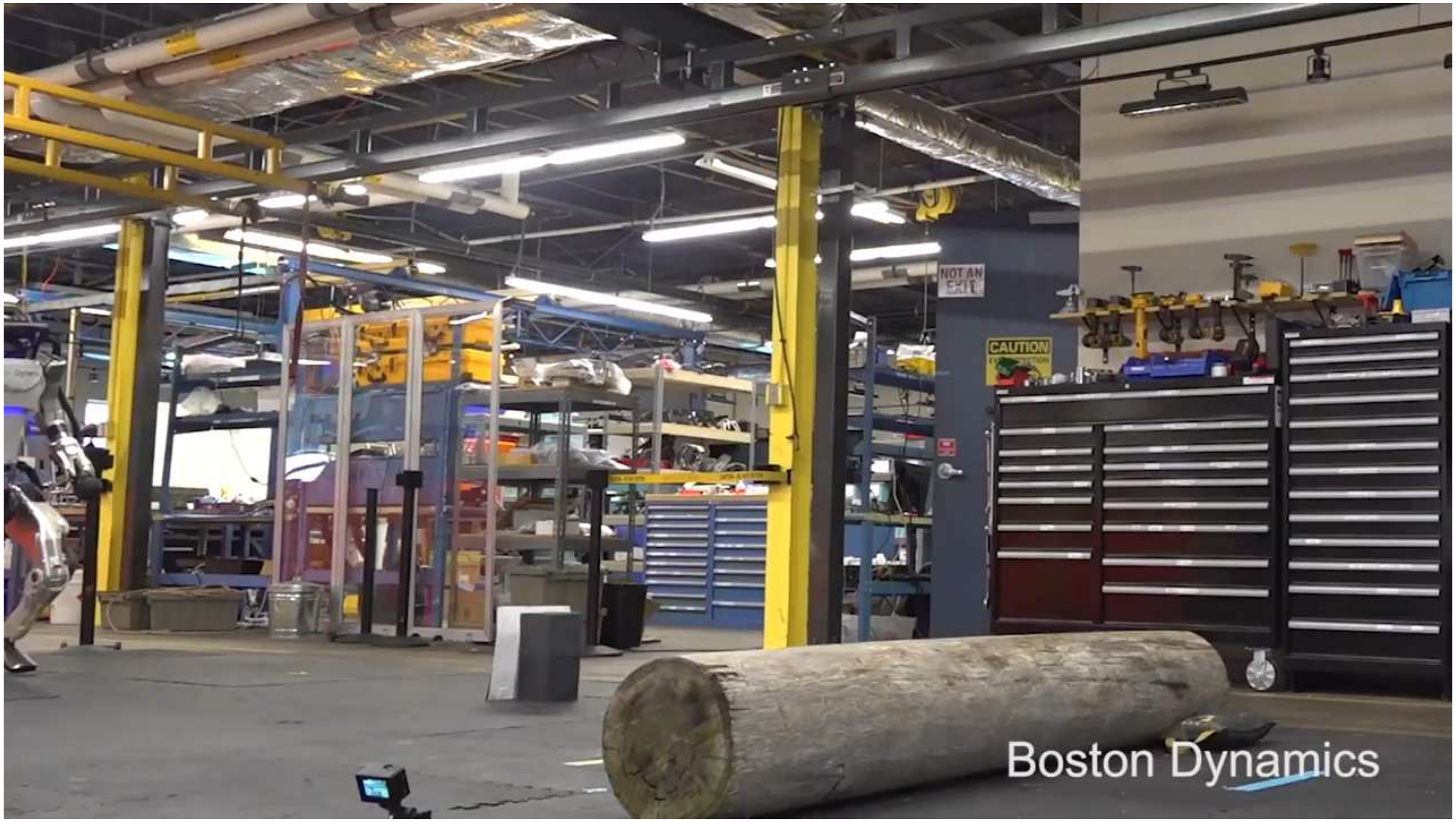
Robotics is a **BIG** space



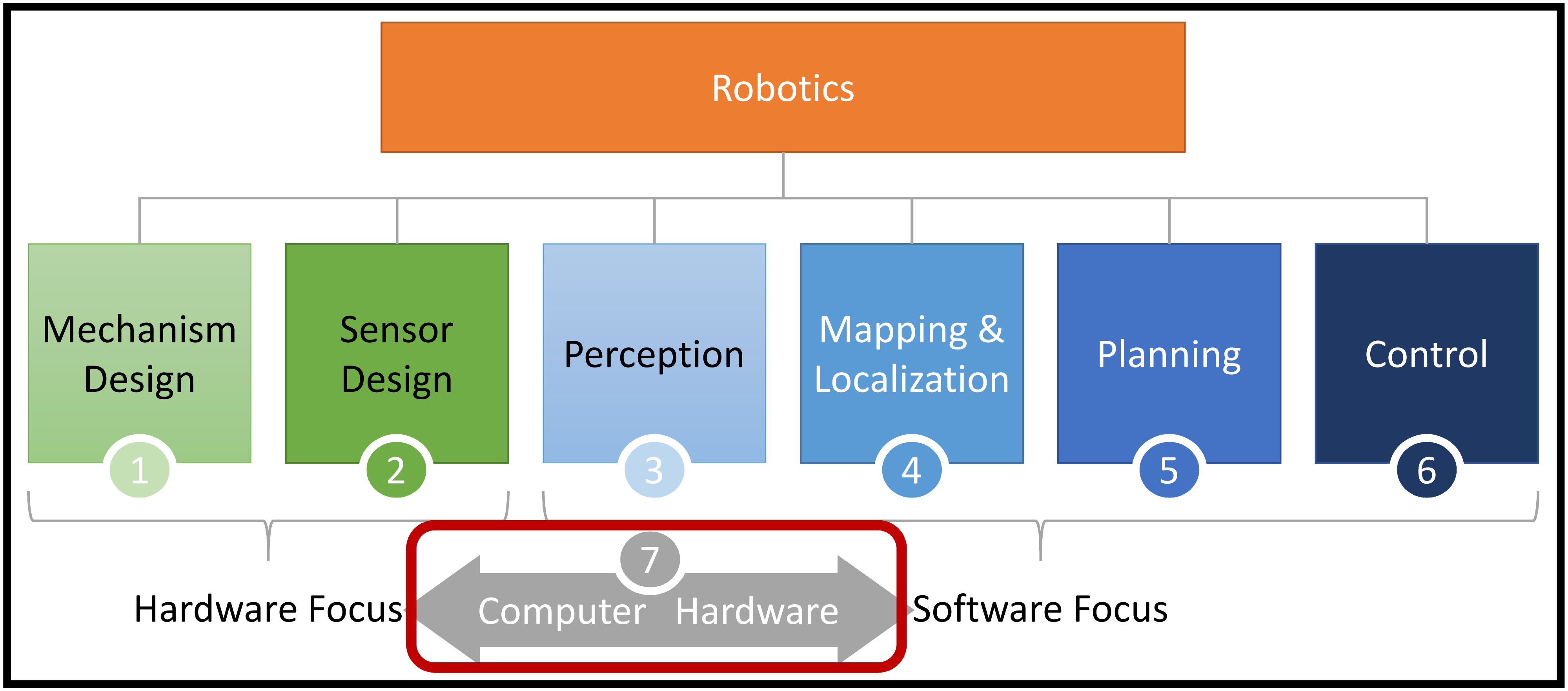
6 Control is the process of executing a plan in the real world



6 Control is the process of executing a plan in the real world



Robotics is a **BIG** space



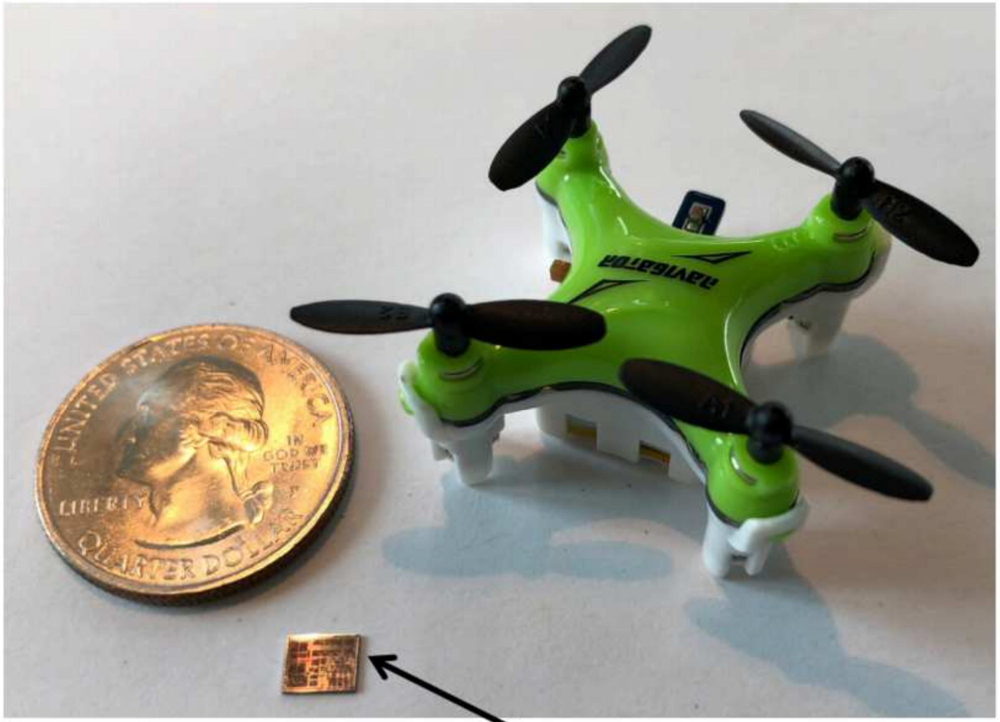
7

Computer Hardware Designers are coming up with new custom chips to deliver real time low power performance

http://navion.mit.edu

- A. Suleiman, Z. Zhang, L. Carlone, S. Karaman, V. Sze, "**Navion: A Fully Integrated Energy-Efficient Visual-Inertial Odometry Accelerator for Autonomous Navigation of Nano Drones,**" *IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, June 2018.

- Z. Zhang*, A. Suleiman*, L. Carlone, V. Sze, S. Karaman, "**Visual-Inertial Odometry on Chip: An Algorithm-and-Hardware Co-design Approach,**" *Robotics: Science and Systems (RSS)*, July 2017.



Navion

7

Computer Hardware Designers are coming up with new custom chips to deliver real time low power performance

Platform	Xeon (E5-2667)	ARM (Cortex A15)	Navion (Peak w/ Max Config)	Navion (Real-time w/ Optimized Config)
Trajectory Error (%)	0.22%		0.28%	0.27%
Camera rate (fps)	63	19	**71**	**20**
Keyframe rate (fps)	12	2	**19**	**5**
Average Power (W)	27.9	2.4	**0.024**	**0.002**
Energy (mJ/KF)	3,638	1,573	**2.3**	**0.7**

CS 14x

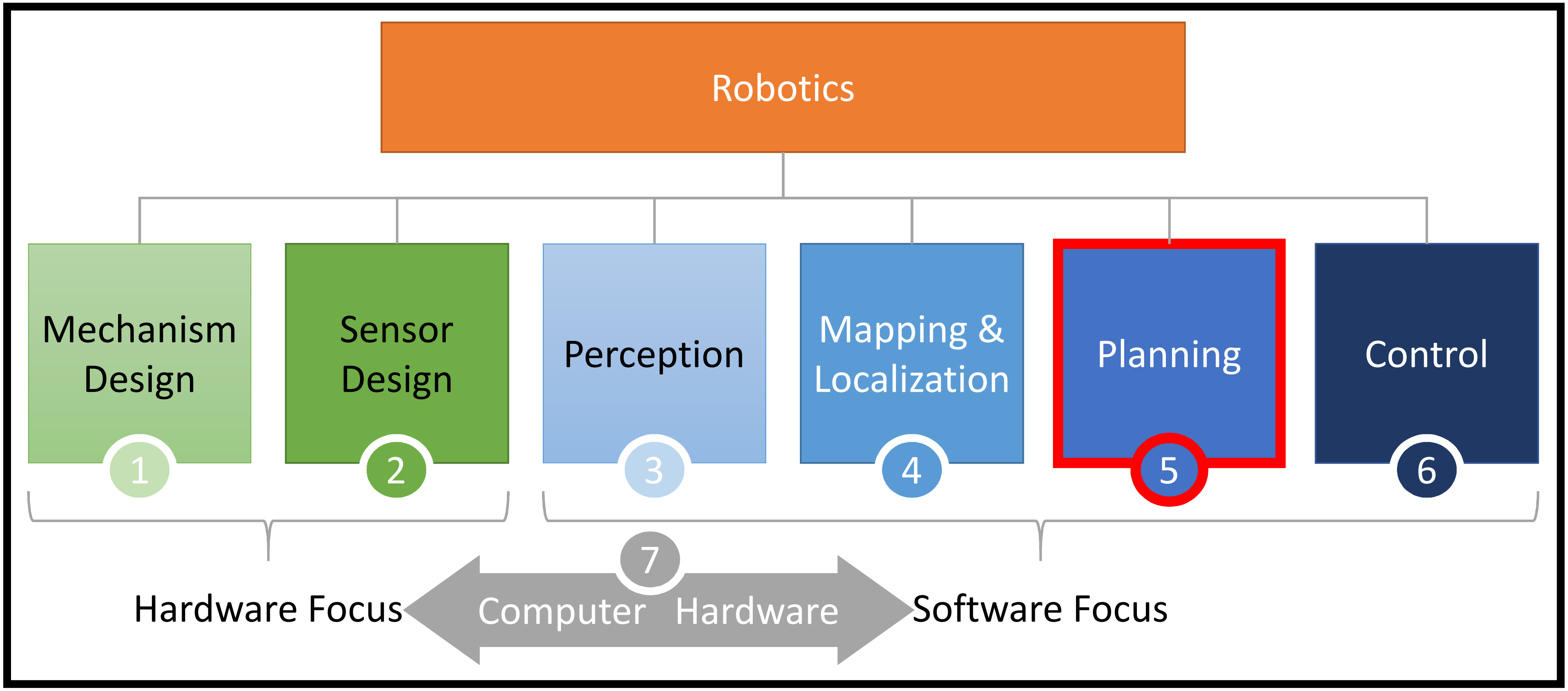
CS 24x

Navion Energy:

684x or 2,247x less than embedded ARM CPU

1,582x or 5,197x less than server Xeon CPU

Our Focus for today: Robot Motion Planning



Our Focus for today: Robot Motion Planning

- How do we plan motions in high-dimensional continuous spaces?
- Why planning (and not policies)?
  - Plans are often cheaper to compute than policies
  - Robot operation often a series of self-contained tasks
    that can be formulated as independent planning problems

Our Focus for today: Robot Motion Planning

- How do we plan motions in high-dimensional continuous spaces?
- Why planning (and not policies)?
  - Plans are often cheaper to compute than policies
  - Robot operation often a series of self-contained tasks that can be formulated as independent planning problems
- Example: *pick up the ball and put it in the bin*

Our Focus for today: Robot Motion Planning

- How do we plan motions in high-dimensional continuous spaces?
- Why planning (and not policies)?
  - Plans are often cheaper to compute than policies
  - Robot operation often a series of self-contained tasks that can be formulated as independent planning problems
- Example: *pick up the ball and put it in the bin*
  - If we can come up with a good representation of states, actions, transition model, etc. then we can simply search that space for a plan!

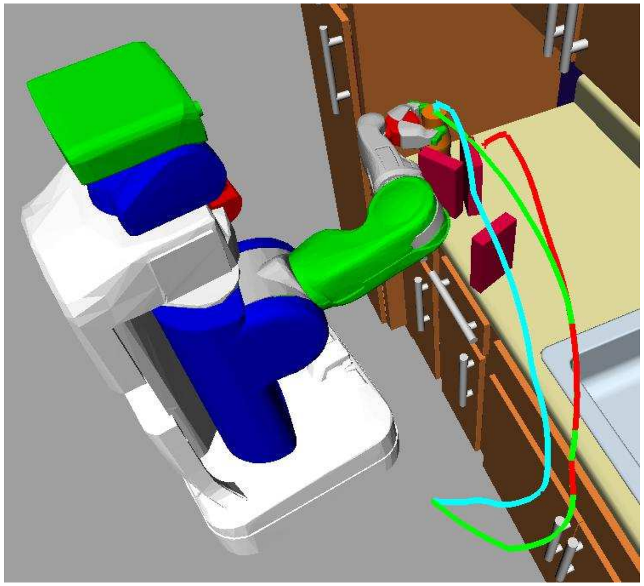
Our Focus for today: Robot Motion Planning

- How do we plan motions in high-dimensional continuous spaces?
- Why planning (and not policies)?
  - Plans are often cheaper to compute than policies
  - Robot operation often a series of self-contained tasks that can be formulated as independent planning problems
- Example: *pick up the ball and put it in the bin*
  - If we can come up with a good representation of states, actions, transition model, etc. then we can simply search that space for a plan!

But what kind of space should we search in?

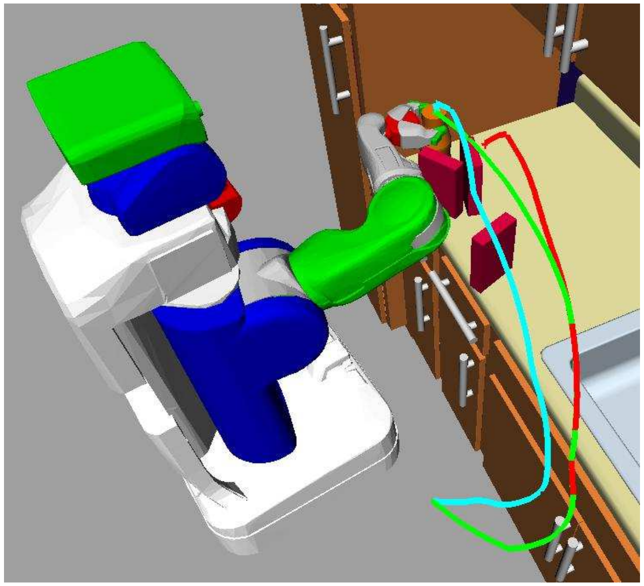
Spaces and Transformations

- **Task space**: the 3D workspace of the robot
  - E.g., the **pose** (x,y,z,roll,pitch,yaw) of the robot's hand or an object



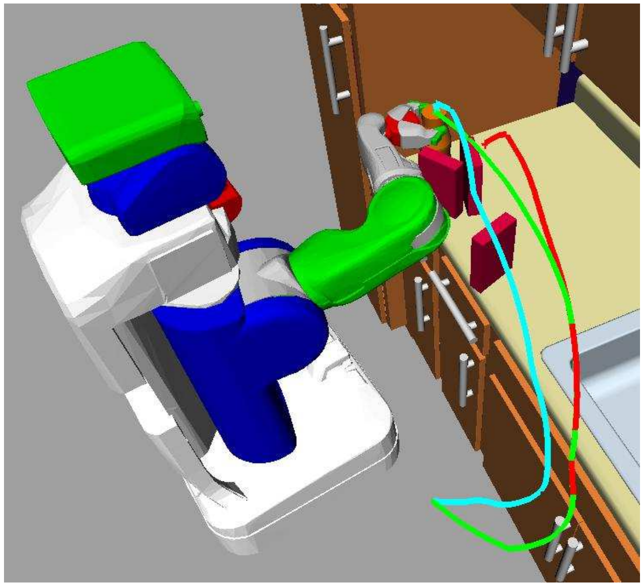
Spaces and Transformations

- **Task space**: the 3D workspace of the robot
  - E.g., the **pose** (x,y,z,roll,pitch,yaw) of the robot's hand or an object
- **Configuration space**: the *n*-dimensional space of joint angles + robot world position
  - Vector $q \in \mathbb{R}^n$



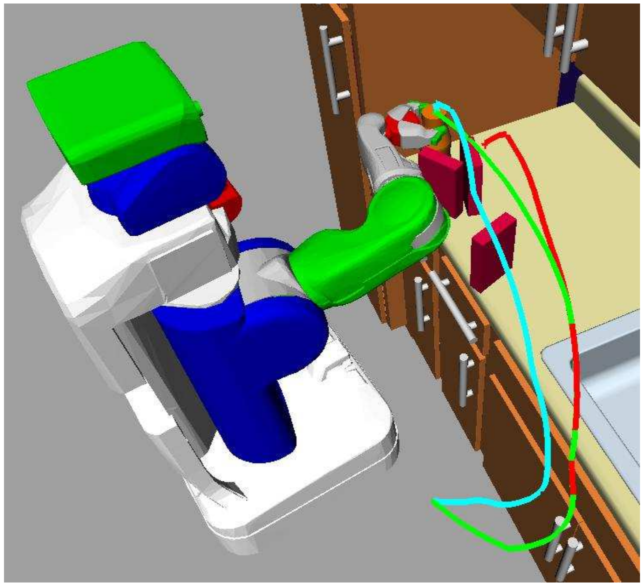
Spaces and Transformations

- **Task space**: the 3D workspace of the robot
  - E.g., the **pose** (x,y,z,roll,pitch,yaw) of the robot's hand or an object
- **Configuration space**: the *n*-dimensional space of joint angles + robot world position
  - Vector $q \in \mathbb{R}^n$
- **Forward kinematics**: maps *q* to outputs in task space (e.g. hand position)
- **Inverse kinematics**: maps task space poses to configuration space



Spaces and Transformations

- **Task space**: the 3D workspace of the robot
  - E.g., the **pose** (x,y,z,roll,pitch,yaw) of the robot's hand or an object
- **Configuration space**: the *n*-dimensional space of joint angles + robot world position
  - Vector $q \in \mathbb{R}^n$
- **Forward kinematics**: maps *q* to outputs in task space (e.g. hand position)
- **Inverse kinematics**: maps task space poses to configuration space



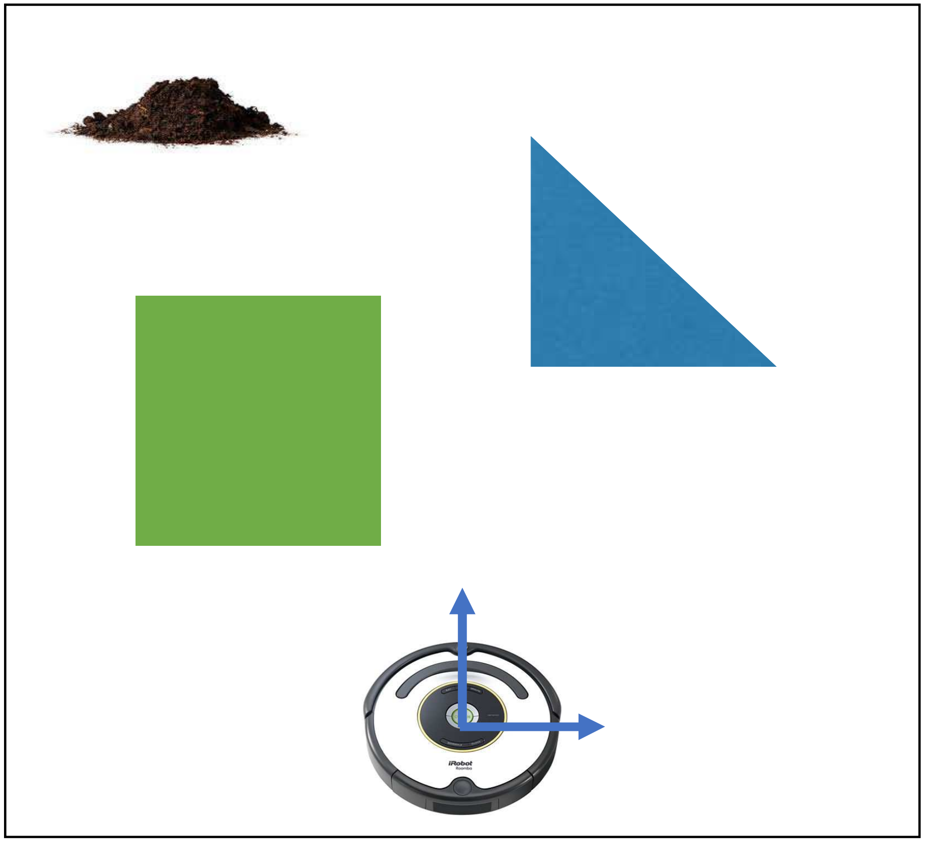
Q: Are forward and inverse kinematics unique?

Spaces and Transformations

- **Task space**: the 3D workspace of the robot
  - E.g., the **pose** (x,y,z,roll,pitch,yaw) of the robot's hand or an object
- **Configuration space**: the *n*-dimensional space of joint angles + robot world position
  - Vector $q \in \mathbb{R}^n$
- **Forward kinematics**: maps *q* to outputs in task space (e.g. hand position)
- **Inverse kinematics**: maps task space poses to configuration space

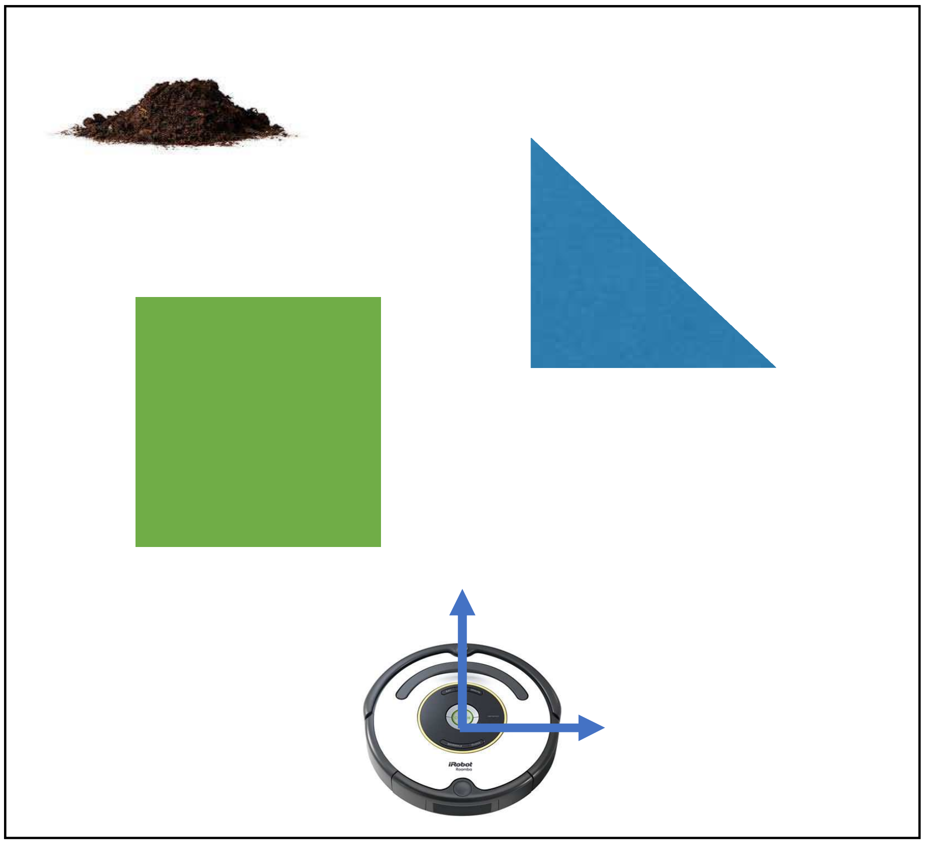
- Insight: mapping task space obstacles and goals into configuration space turns this into a problem of **planning a path for a single point**

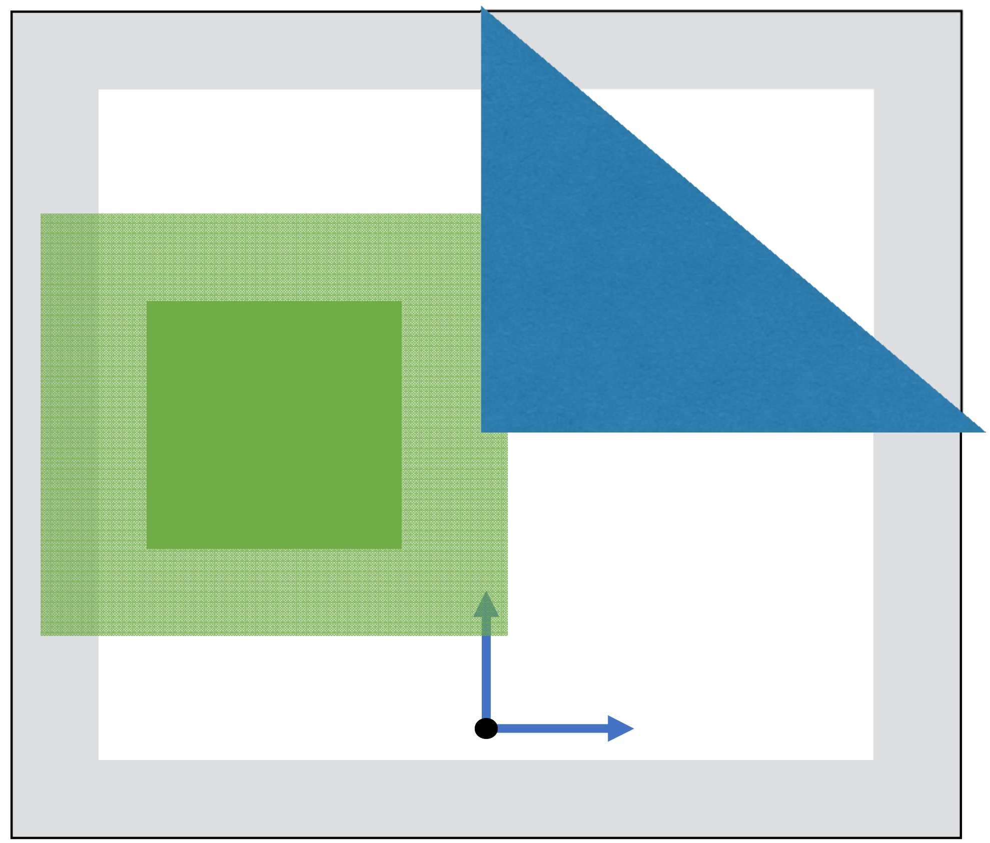
Configuration Space



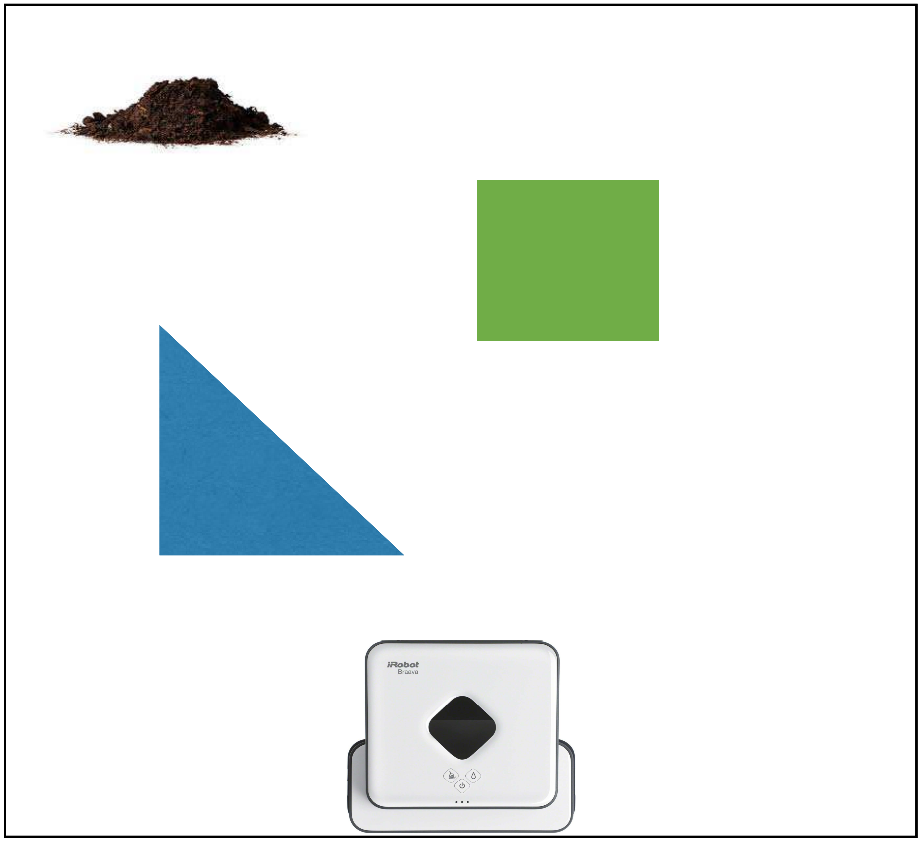
Q: What would the configuration space look like for this robot?

Configuration Space



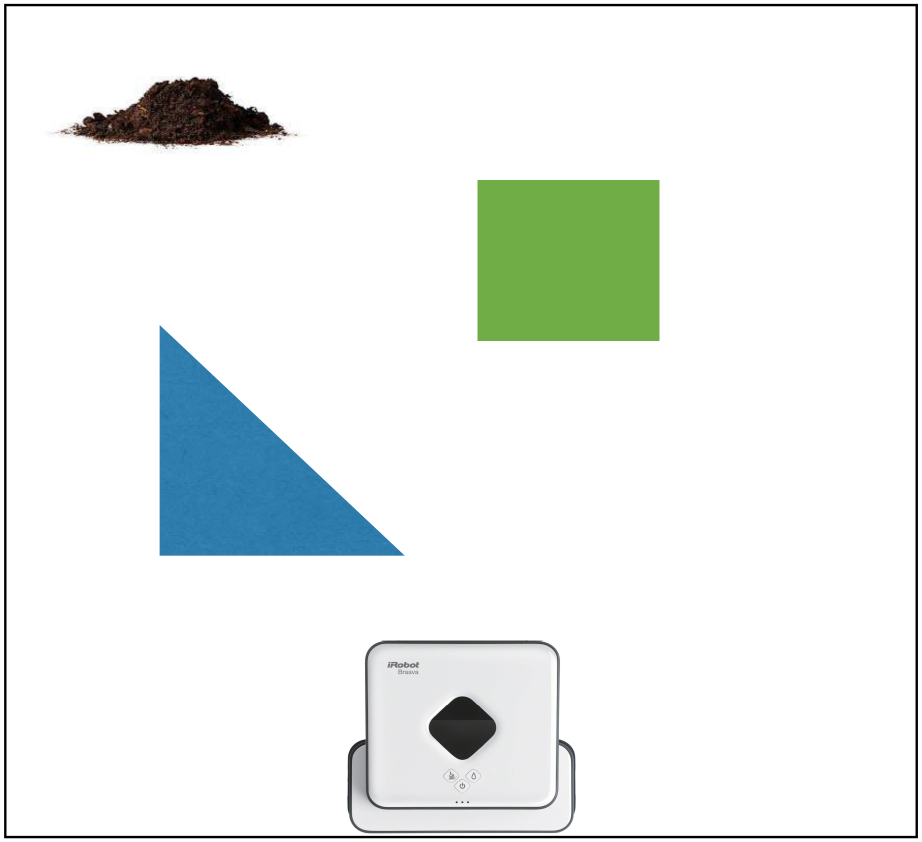


Configuration Space



Q: What about this square robot?

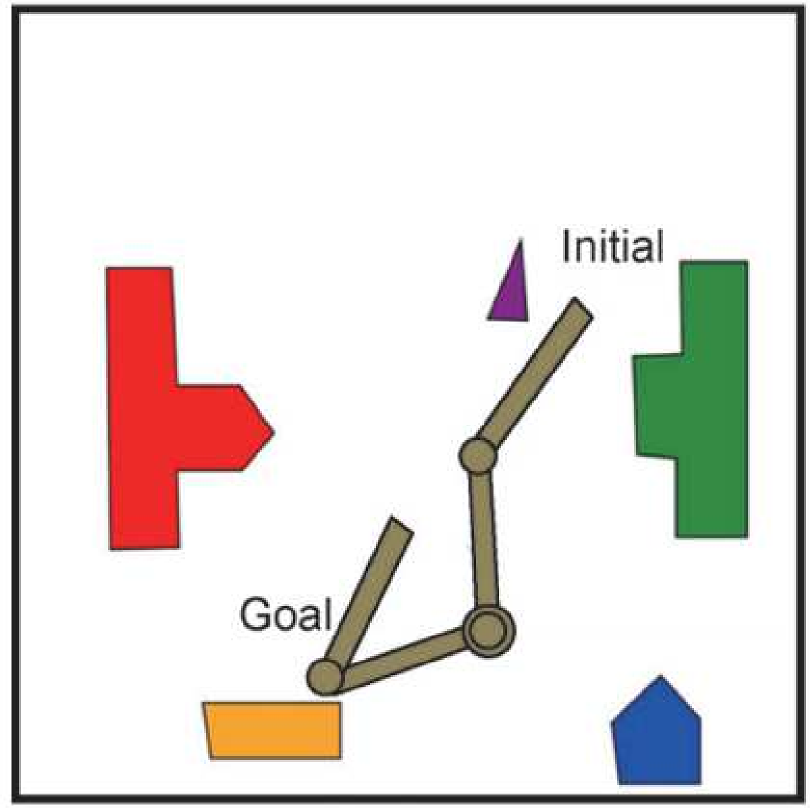
Configuration Space



- Well for the Square robot the obstacle clearance depends on rotation too!
- Configuration space is 3-dimensional *(x, y, rotation)*

Configuration Space

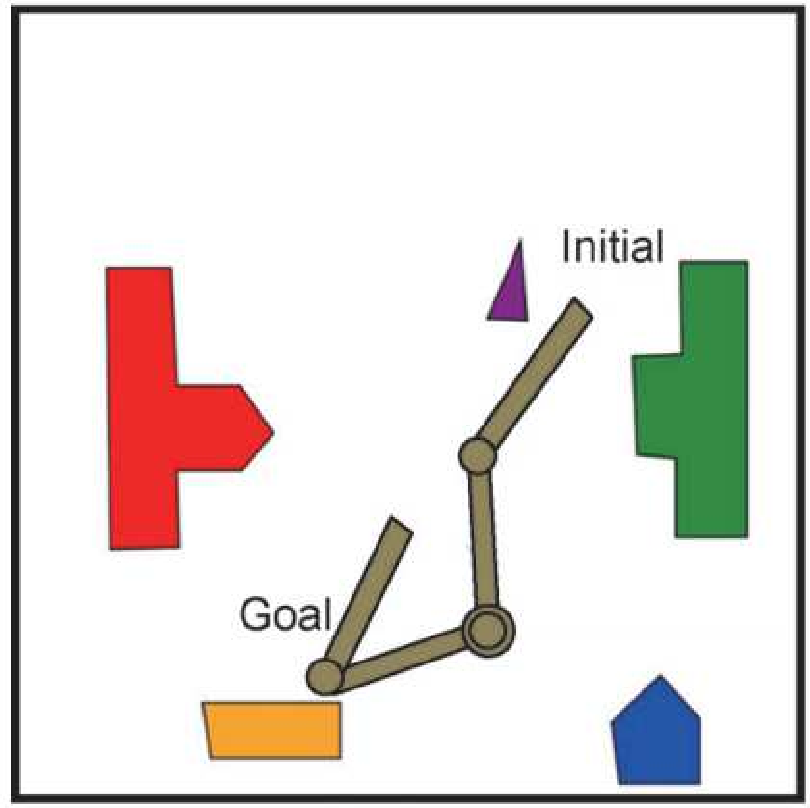
- Consider a simple 2-link robot arm in the **task space (x,y)** shown on the right.
- How could we instead think of the **configuration space**? What would uniquely determine the end effector position?



Workspace

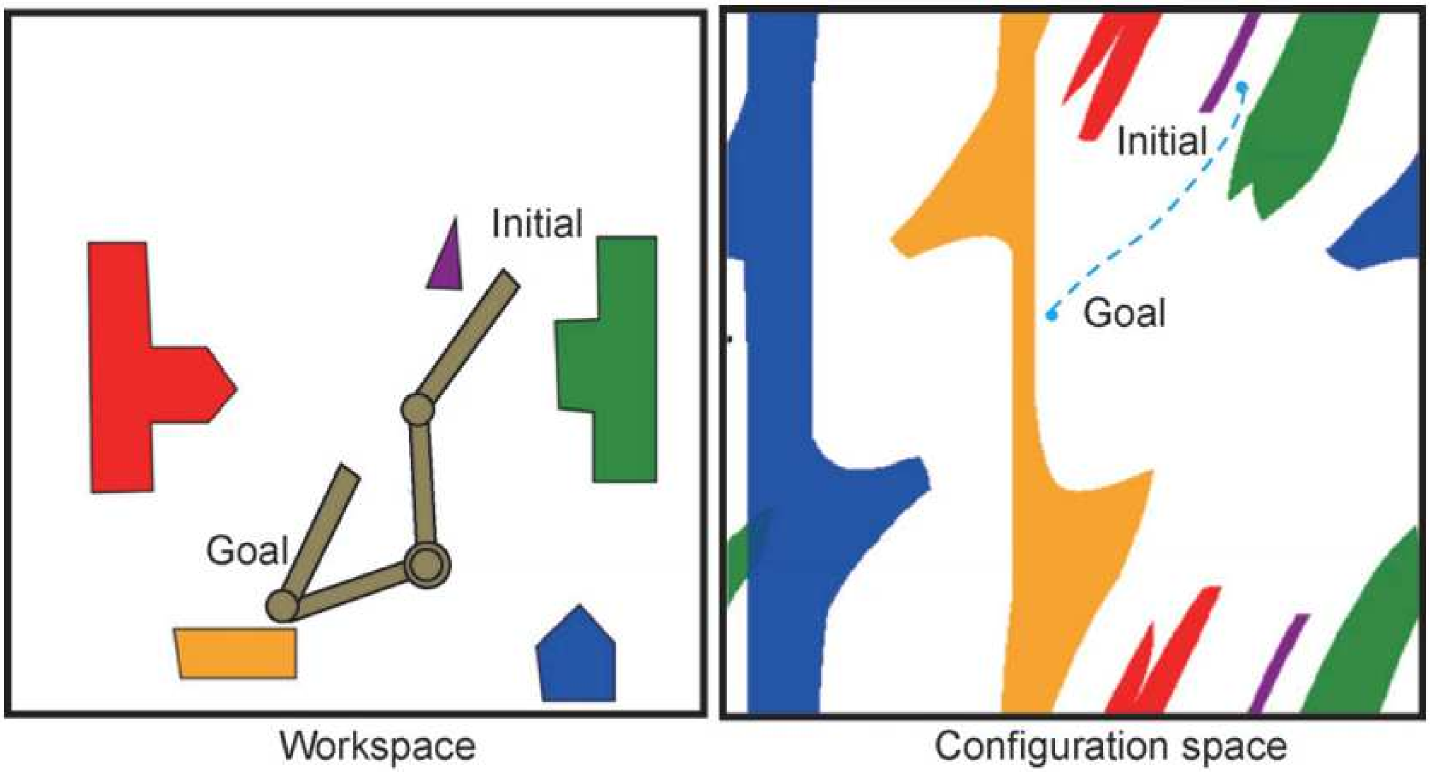
Configuration Space

- Consider a simple 2-link robot arm in the **task space (x,y)** shown on the right.
- How could we instead think of the **configuration space**? What would uniquely determine the end effector position?
- Well if we consider the two joint angles of the arm we can uniquely determine the position of the end-effector so lets make our configuration space $(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2)$



Workspace

Configuration Space

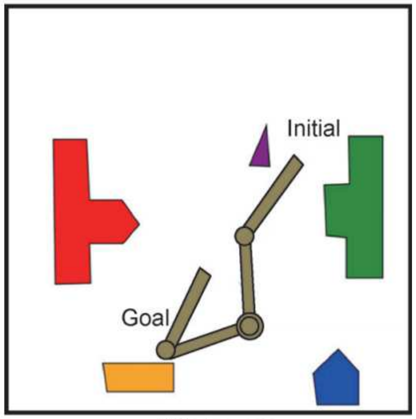


Hmmm this is getting complex quite fast...

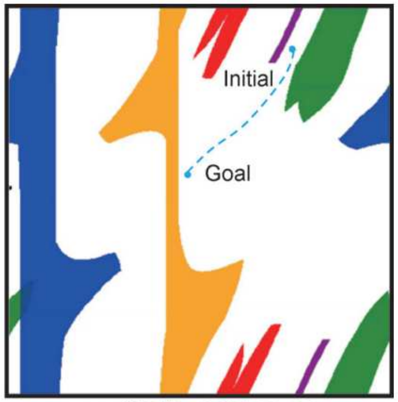
How to use configuration space in practice

If we map the obstacles into configuration space we can check whether the configuration point, q, is in an obstacle and we have a **unique plan** for the robot

- **Problem:** mapping obstacles into configuration space is hard



Workspace



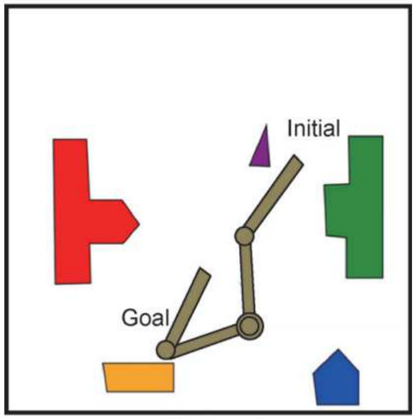
Configuration space

How to use configuration space in practice

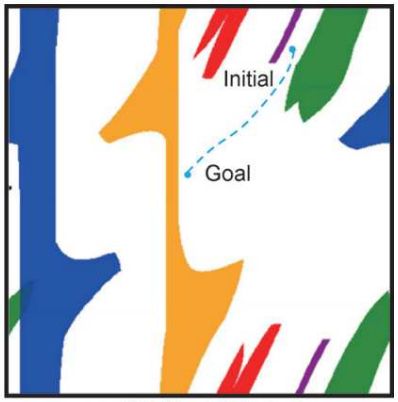
If we map the obstacles into configuration space we can check whether the configuration point, q, is in an obstacle and we have a **unique plan** for the robot

- **Problem:** mapping obstacles into configuration space is hard

Better approach: **use forward kinematics** to check task space obstacle collisions!



Workspace



Configuration space

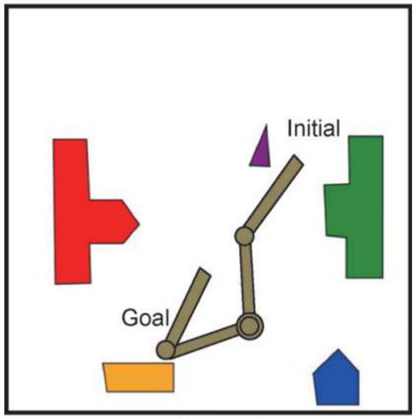
How to use configuration space in practice

If we map the obstacles into configuration space we can check whether the configuration point, *q*, is in an obstacle and we have a **unique plan** for the robot

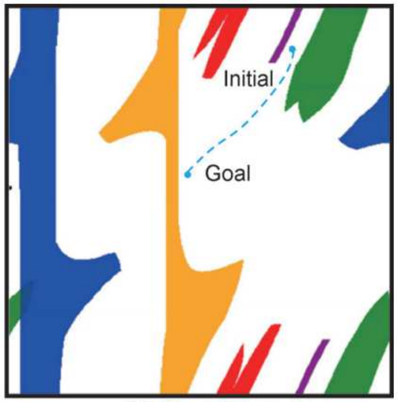
- **Problem:** mapping obstacles into configuration space is hard

Better approach: **use forward kinematics** to check task space obstacle collisions!

- **No free lunch** – Now each collision check requires full kinematics and not a simple lookup



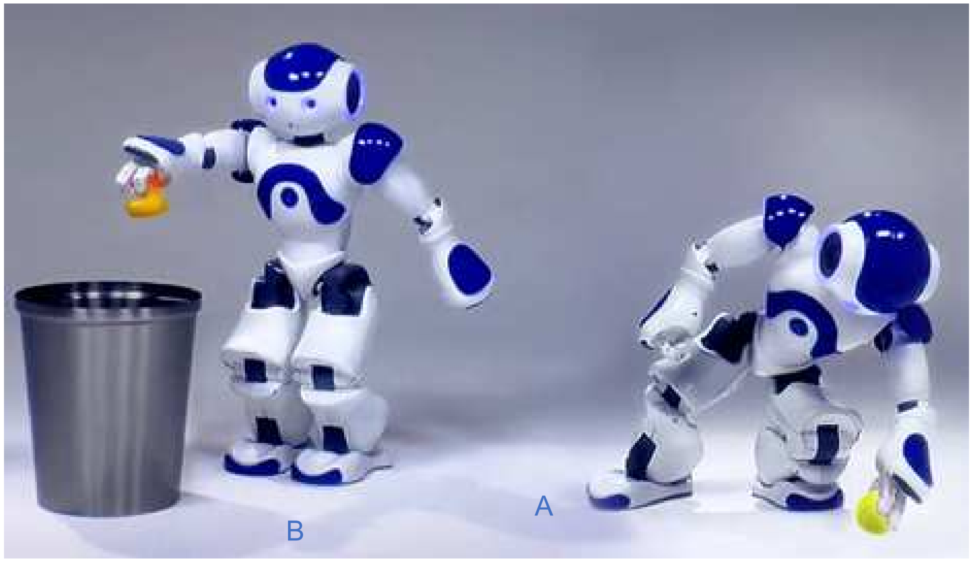
Workspace



Configuration space

Planning in Configuration Space

Suppose we have a configuration space representation of our planning problem



Planning in Configuration Space

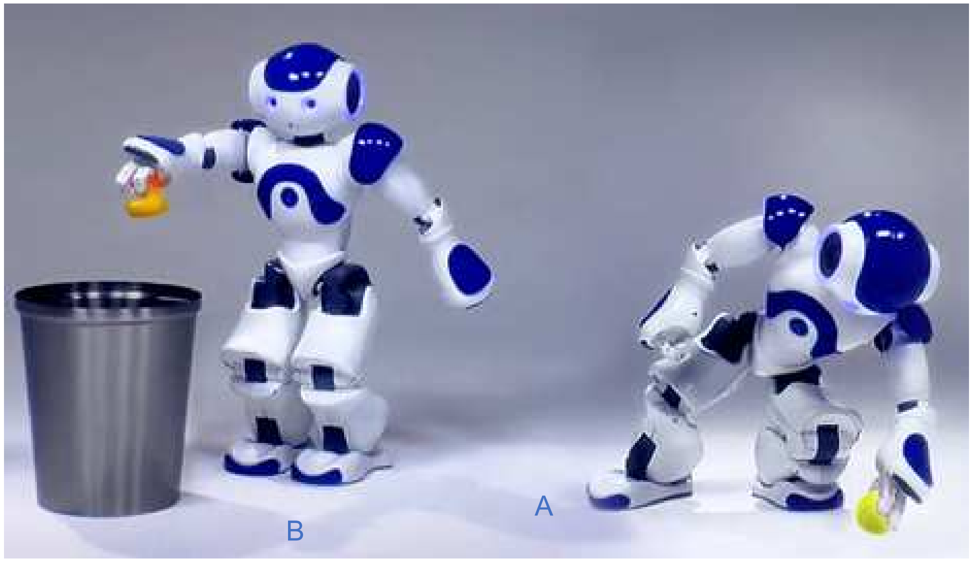
Suppose we have a configuration space representation of our planning problem

Goal: Find shortest collision-free path from configuration *A* to *B*

States: configurations $q \in \mathcal{R}^{\sim 20}$

Actions: Δq

Transition: $q' \leftarrow q + \Delta q$



Planning in Configuration Space

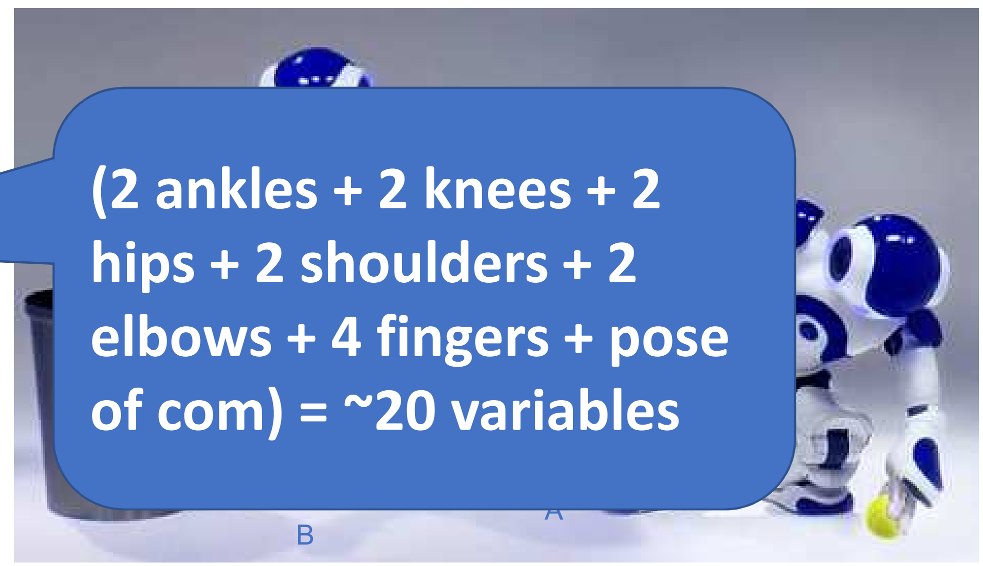
Suppose we have a configuration space representation of our planning problem

Goal: Find shortest collision-free path from configuration *A* to *B*

States: configurations $q \in \mathcal{R}^{\sim 20}$

Actions: Δq

Transition: $q' \leftarrow q + \Delta q$



(2 ankles + 2 knees + 2 hips + 2 shoulders + 2 elbows + 4 fingers + pose of com) = ~20 variables

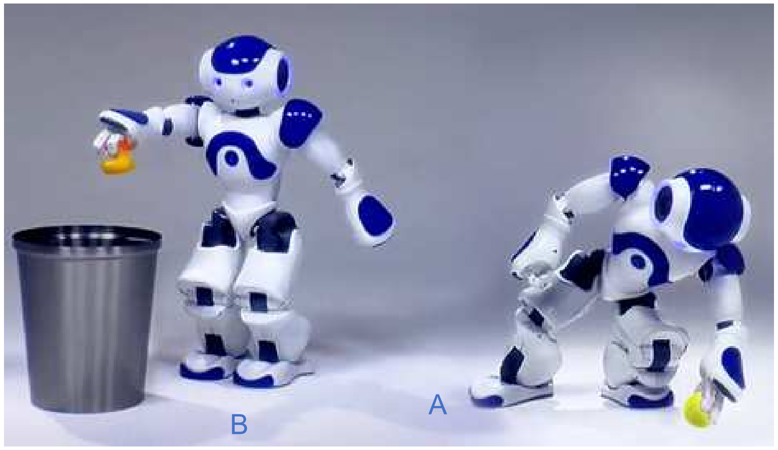
Planning in Configuration Space

Suppose we have a configuration space representation of our planning problem

Goal: Find shortest collision-free path from configuration *A* to *B*

States: configurations $q \in \mathcal{R}^{20}$ **Actions:** Δq **Transition:** $q' \leftarrow q + \Delta q$

If we **discretize** states and actions (e.g., 10 positions per joint) can we use a graph search algorithm like **A***?



Planning in Configuration Space

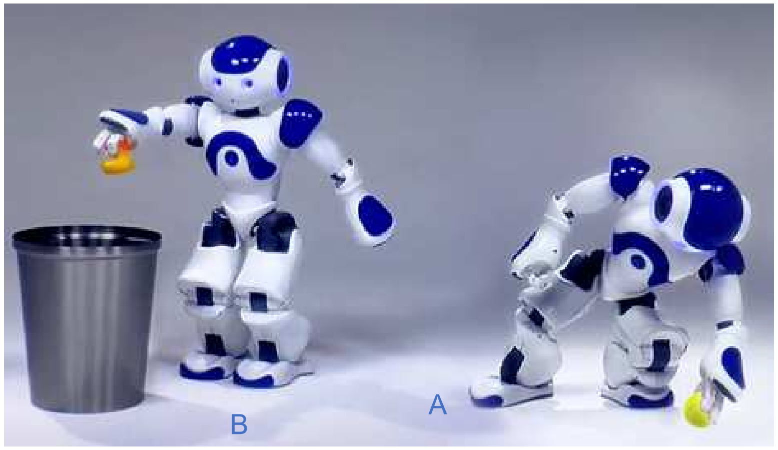
Suppose we have a configuration space representation of our planning problem

Goal: Find shortest collision-free path from configuration *A* to *B*

States: configurations $q \in \mathcal{R}^{20}$ **Actions:** Δq **Transition:** $q' \leftarrow q + \Delta q$

If we **discretize** states and actions (e.g., 10 positions per joint) can we use a graph search algorithm like **A***?

Sure but: $|S| = |A| = 10^{20}$



Planning in Configuration Space

Suppose we have a configuration space representation of our planning problem

Goal: Find shortest collision-free path from configuration *A* to *B*

States: configurations $q \in \mathcal{R}^{20}$ **Actions:** Δq **Transition:** $q' \leftarrow q + \Delta q$

If we **discretize** states and actions (e.g., 10 positions per joint) can we use a graph search algorithm like **A***?

Sure but: $|S| = |A| = 10^{20}$

…curse of dimensionality!

A Naive Random Approach

Well if we can't explore the whole graph at once what if we **incrementally build up a graph** of reachable configurations?

A Naive Random Approach

Well if we can't explore the whole graph at once what if we **incrementally build up a graph** of reachable configurations?

Algorithm (input: s_0, s_{goal}, initial state graph G)

- Pick a random state $s \in G$
- Apply random action a
- Add resulting state s' to G
- Repeat until G has a path from s_0 to s_{goal}

A Naive Random Approach

Well if we can’t explore the whole graph at once what if we **incrementally build up a graph** of reachable configurations?

Algorithm (input: s_0, s_{goal}, initial state graph G)

- Pick a random state $s \in G$
- Apply random action a
- Add resulting state s' to G
- Repeat until G has a path from s_0 to s_{goal}

Probabilistically complete: As iterations go to infinity, probability that G contains a solution goes to 1!

A Naive Random Approach

Well if we can't explore the whole graph at once what if we **incrementally build up a graph** of reachable configurations?

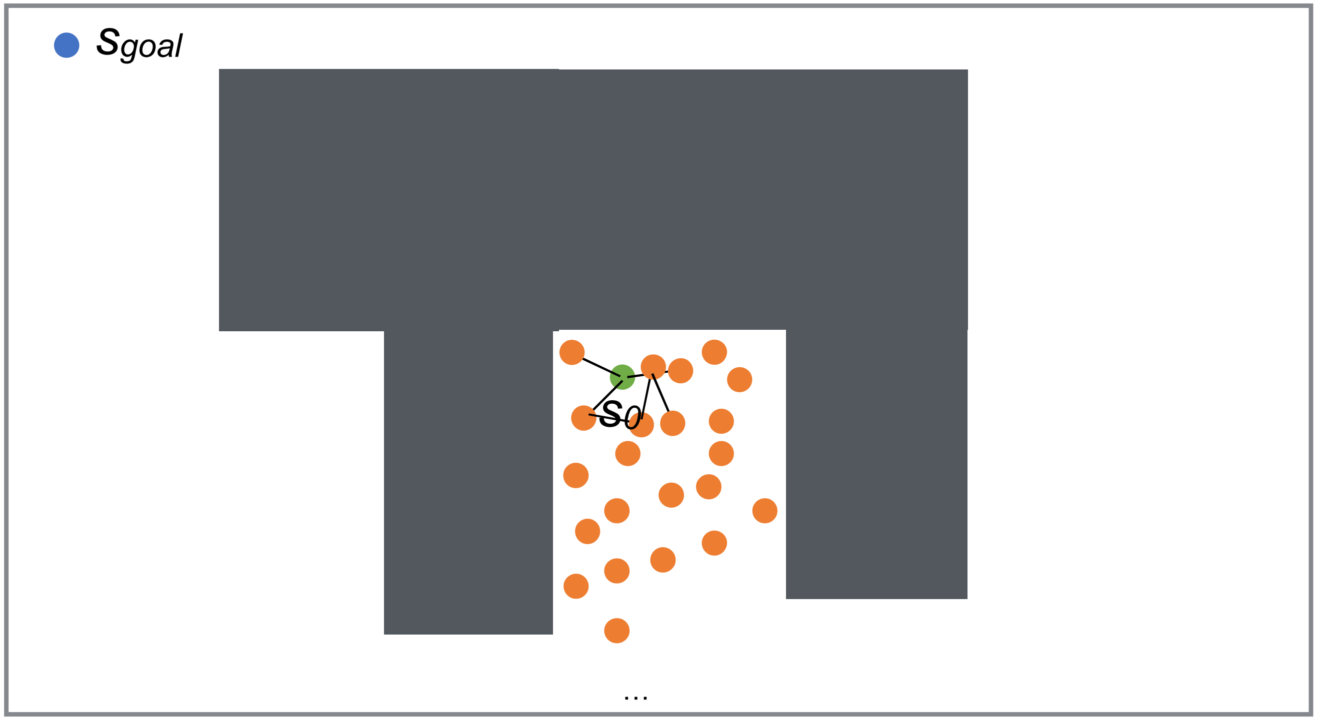
Algorithm (input: s_0, s_{goal}, initial state graph G)

- Pick a random state $s \in G$
- Apply random action a
- Add resulting state s' to G
- Repeat until G has a path from s_0 to s_{goal}

Probabilistically complete: As iterations go to infinity, probability that G contains a solution goes to 1!

Q: What's the problem with this?

Naive Action Sampling



Lots of samples close to your initial state —> slow!

Rapidly Exploring Random Trees

Consider the following tweak to the naive approach called **Rapidly Exploring Random Trees (RRTs)** [Lavalle & Kuffner]

Algorithm (input: s_0, s_{goal}, initial state tree T)

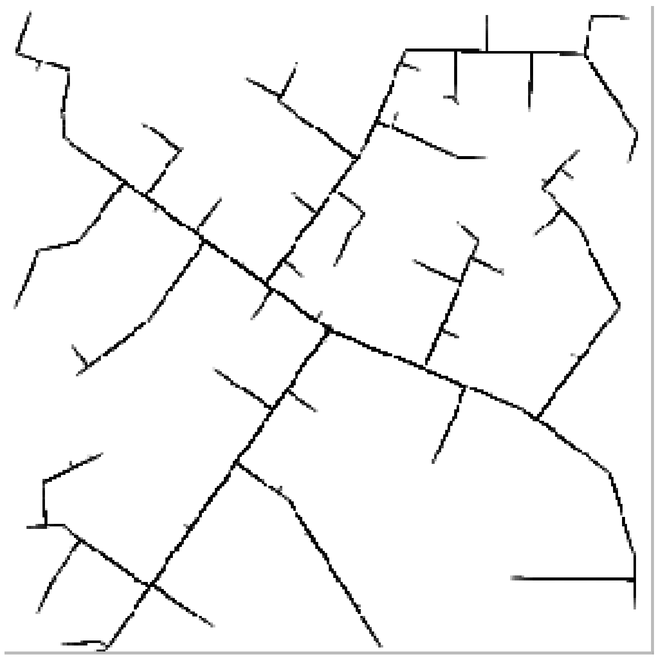
- Sample states $s \in S = R^{20}$ until s is collision-free
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}

Rapidly Exploring Random Trees

Consider the following tweak to the naive approach called **Rapidly Exploring Random Trees (RRTs)** [Lavalle & Kuffner]

Algorithm (input: s_0, s_{goal}, initial state tree T)

- Sample states $s \in S = R^{20}$ until s is collision-free
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



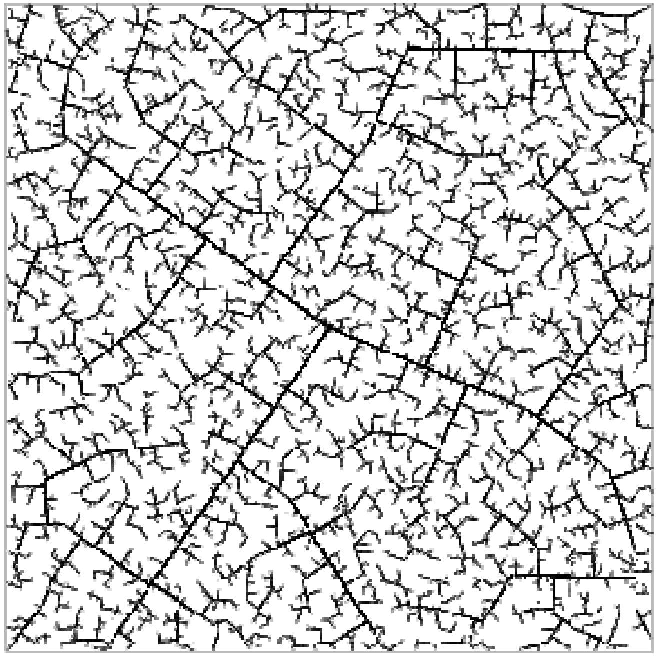
45 iterations

Rapidly Exploring Random Trees

Consider the following tweak to the naive approach called **Rapidly Exploring Random Trees (RRTs)** [Lavalle & Kuffner]

Algorithm (input: s_0, s_{goal}, initial state tree T)

- Sample states $s \in S = R^{20}$ until s is collision-free
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



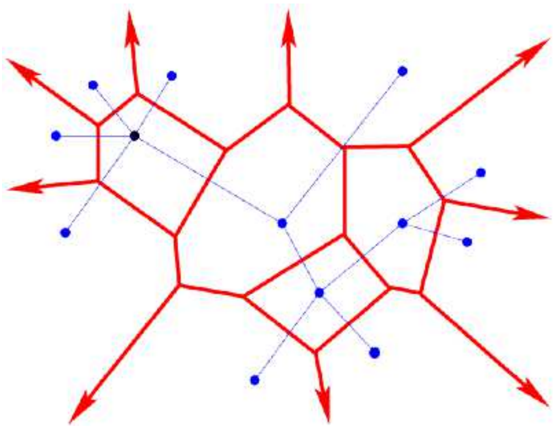
2345 iterations

Randomness encourages exploration

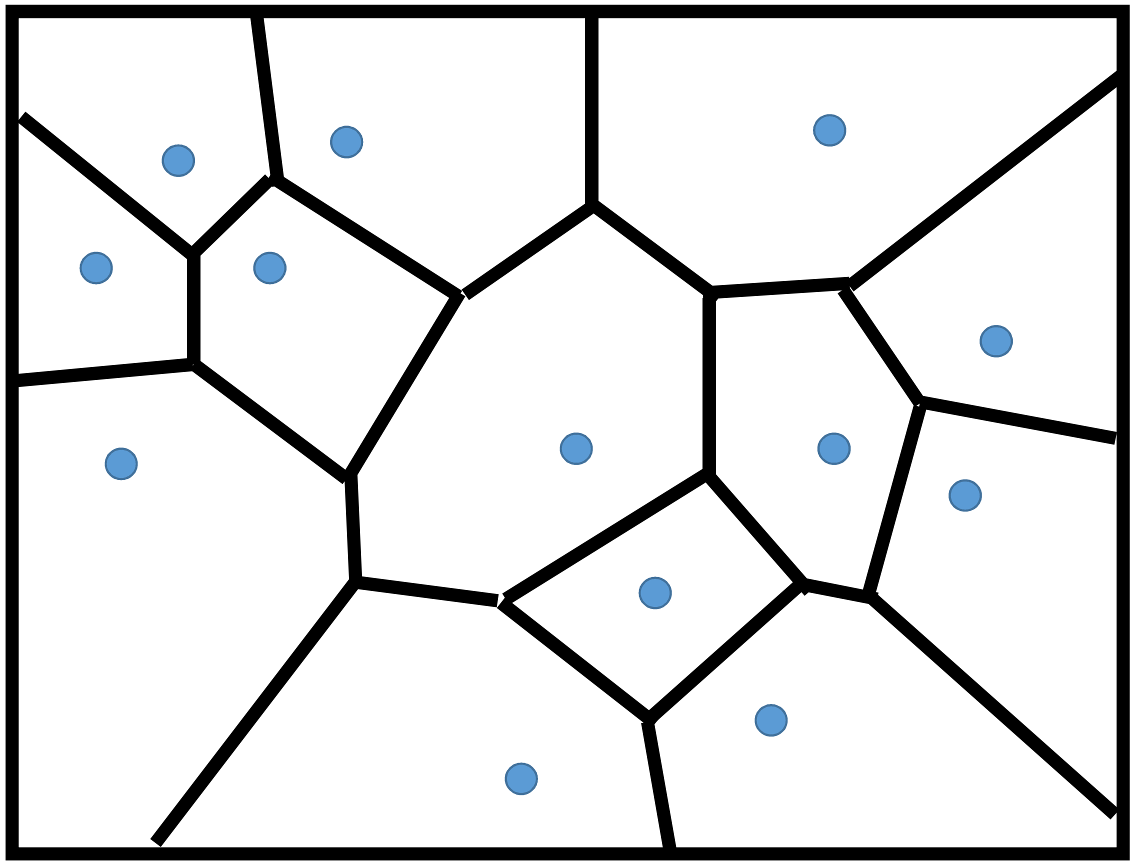
Key idea: uniform random sampling in configuration space is actually a heuristic that encourages exploration!

To see this we use ***Voronoi regions***

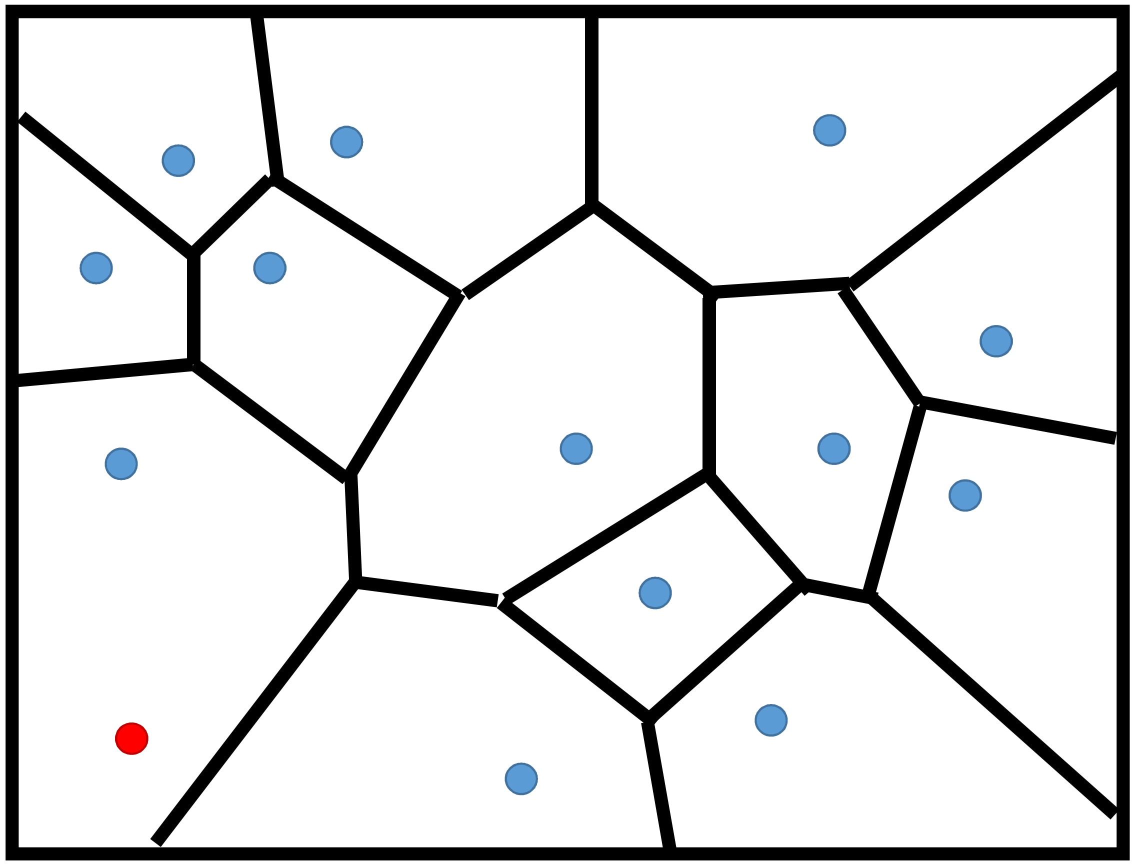
Def: Voronoi region is the set of points in space that are closest to a particular node in the tree:



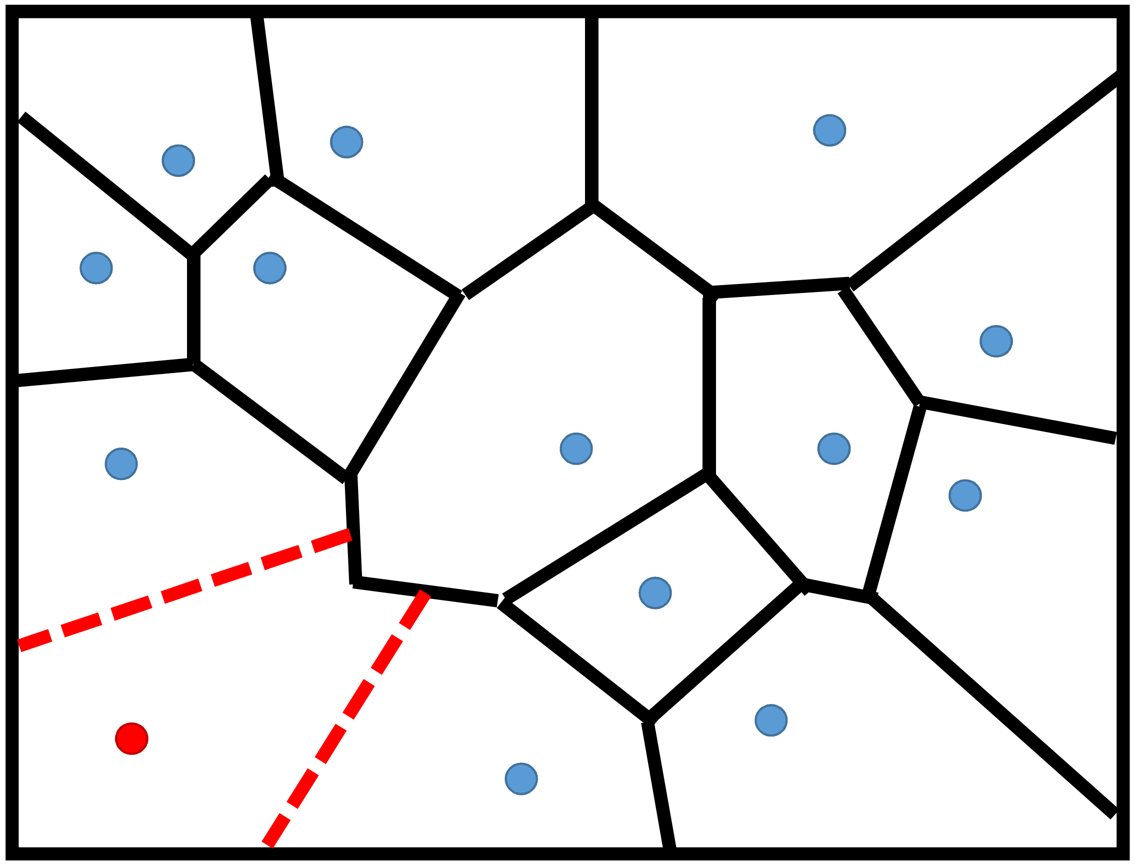
Randomness encourages exploration



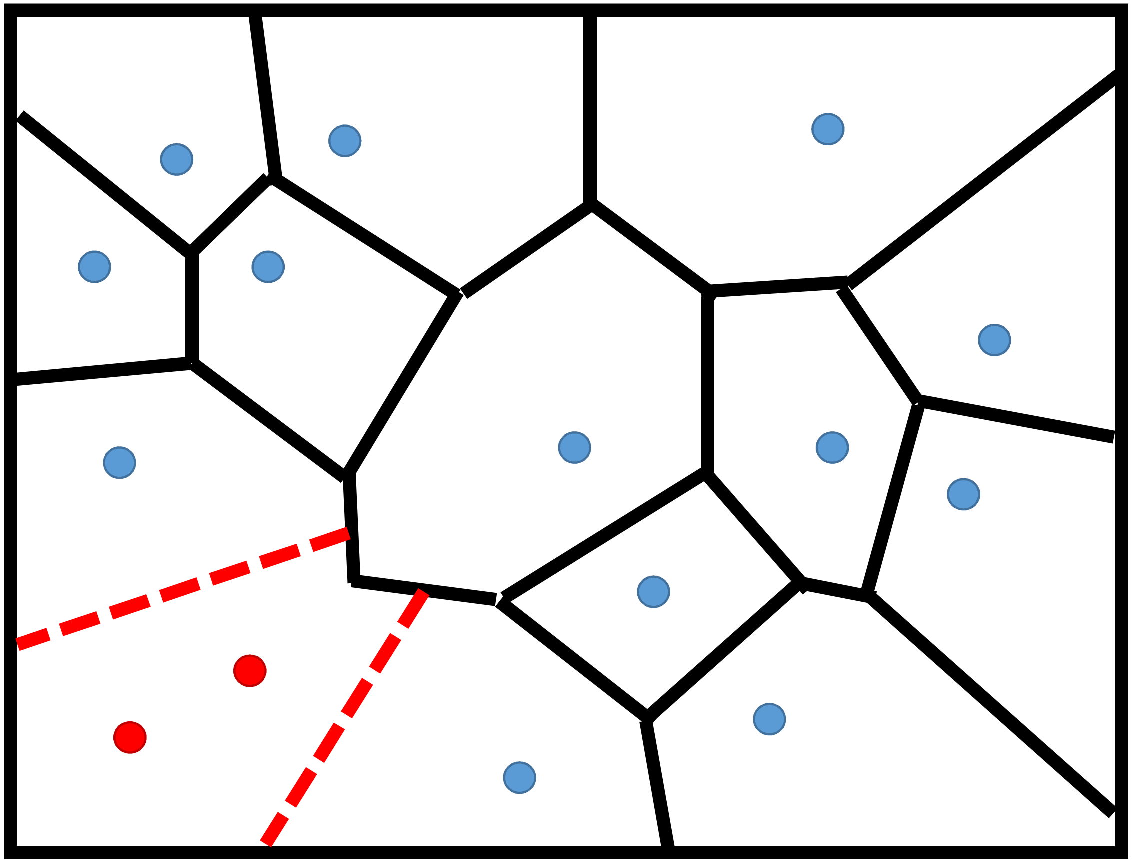
Randomness encourages exploration



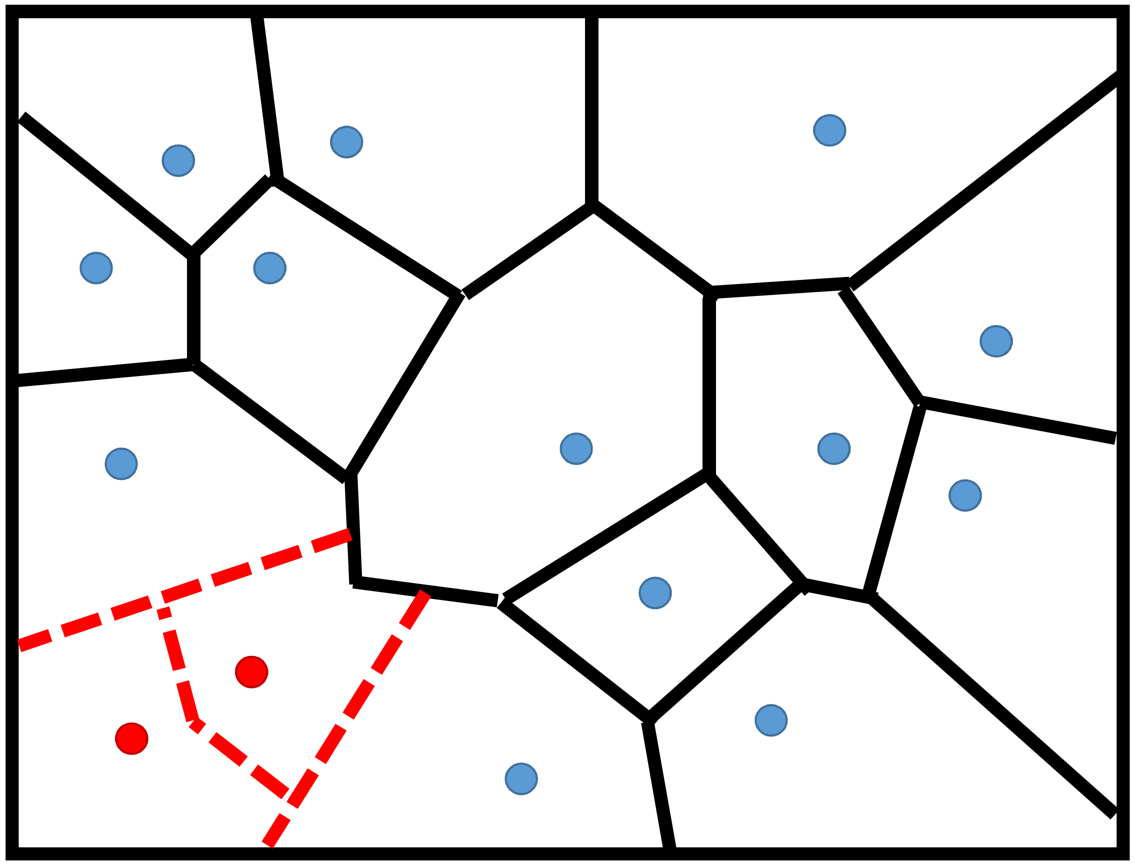
Randomness encourages exploration



Randomness encourages exploration



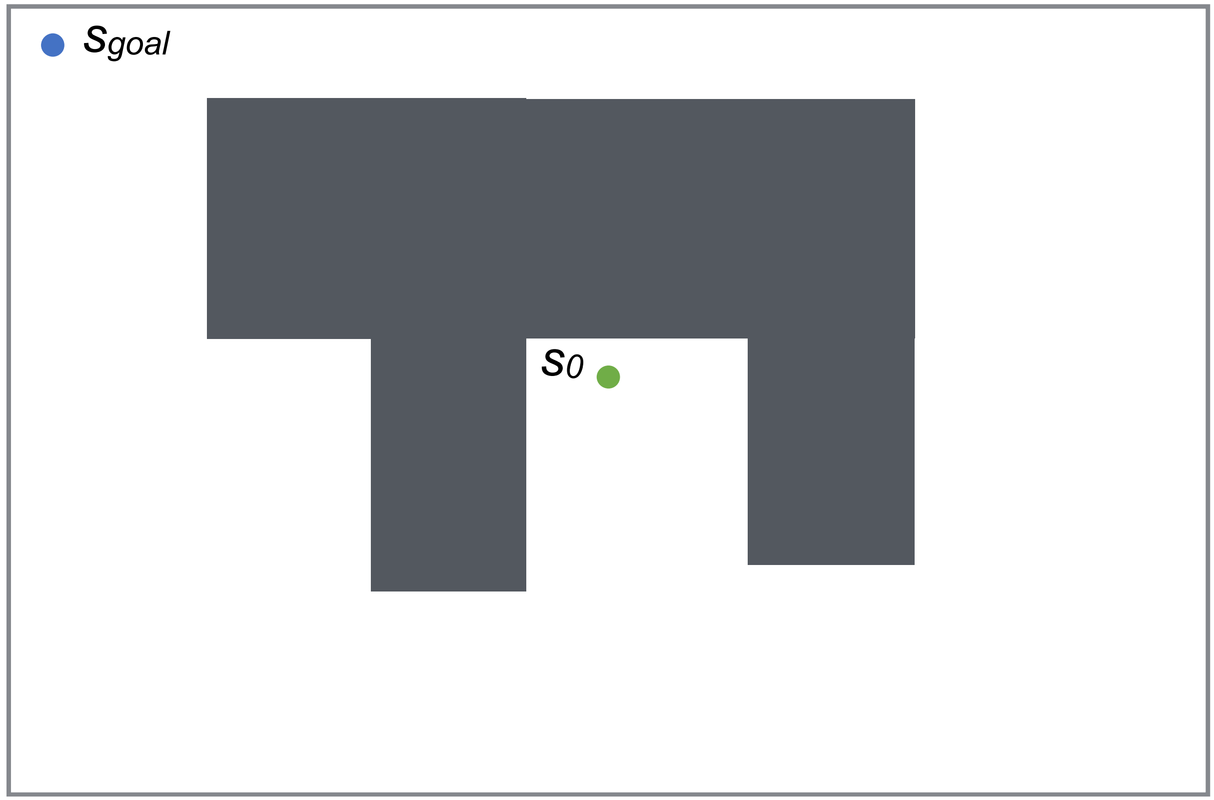
Randomness encourages exploration



Rapidly Exploring Random Trees

Algorithm (**input: s_0, s_{goal}, initial state tree T**)

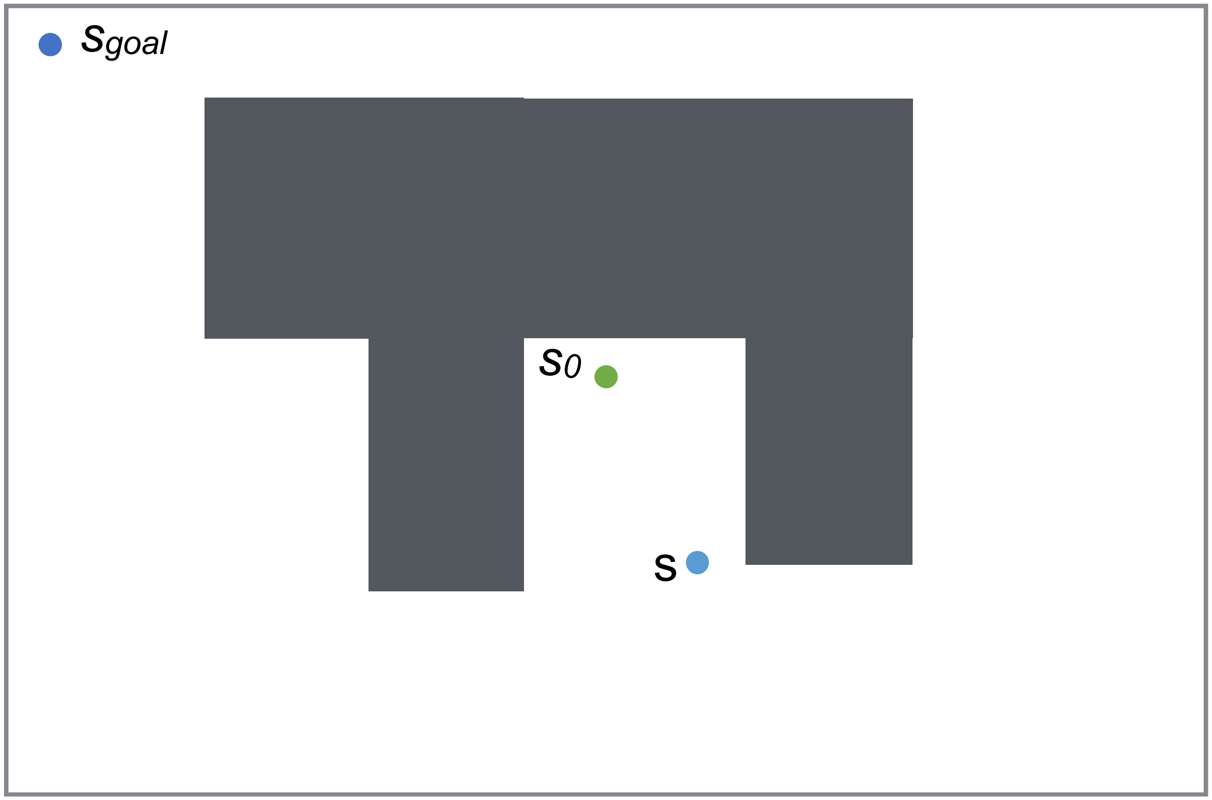
- Sample states $s \in S = R^{20}$ until s is collision-free
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



Rapidly Exploring Random Trees

Algorithm (input: s_0, s_{goal}, initial state tree T)

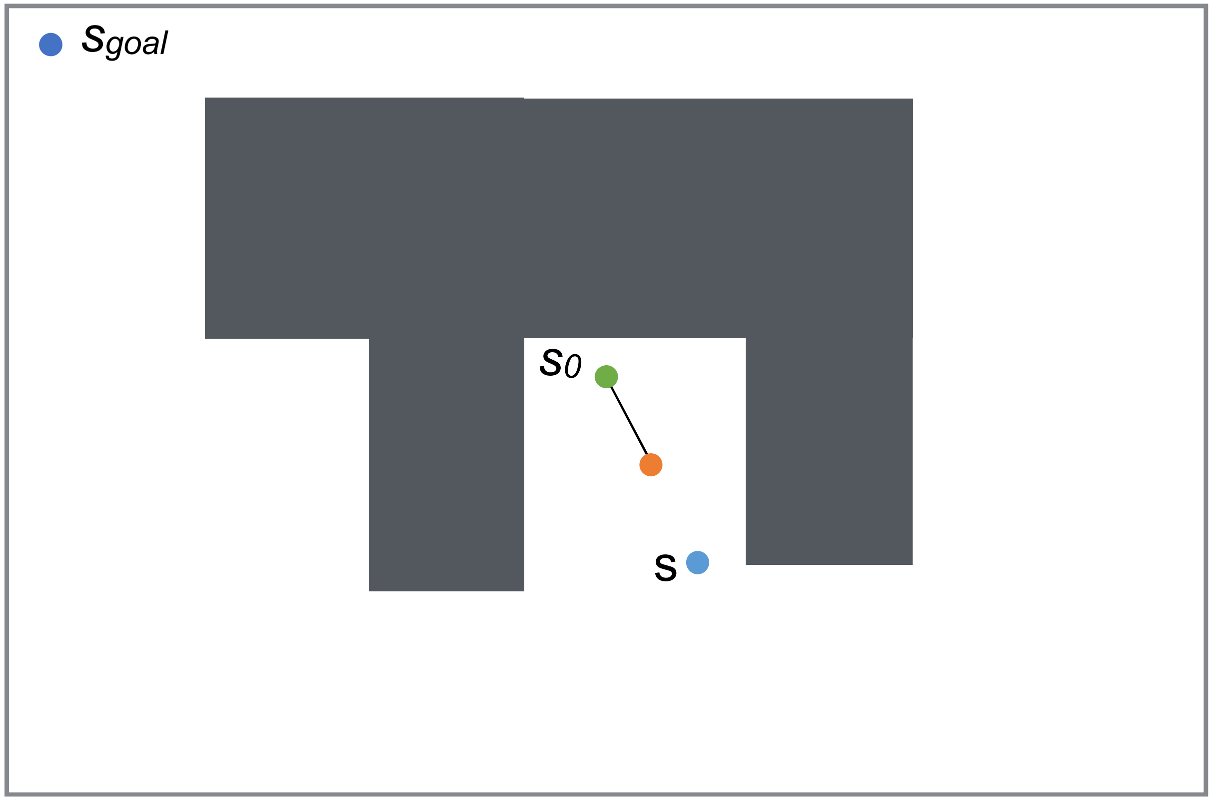
- **Sample states $s \in S = R^{20}$ until s is collision-free**
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



Rapidly Exploring Random Trees

Algorithm (input: s_0, s_{goal}, initial state tree T)

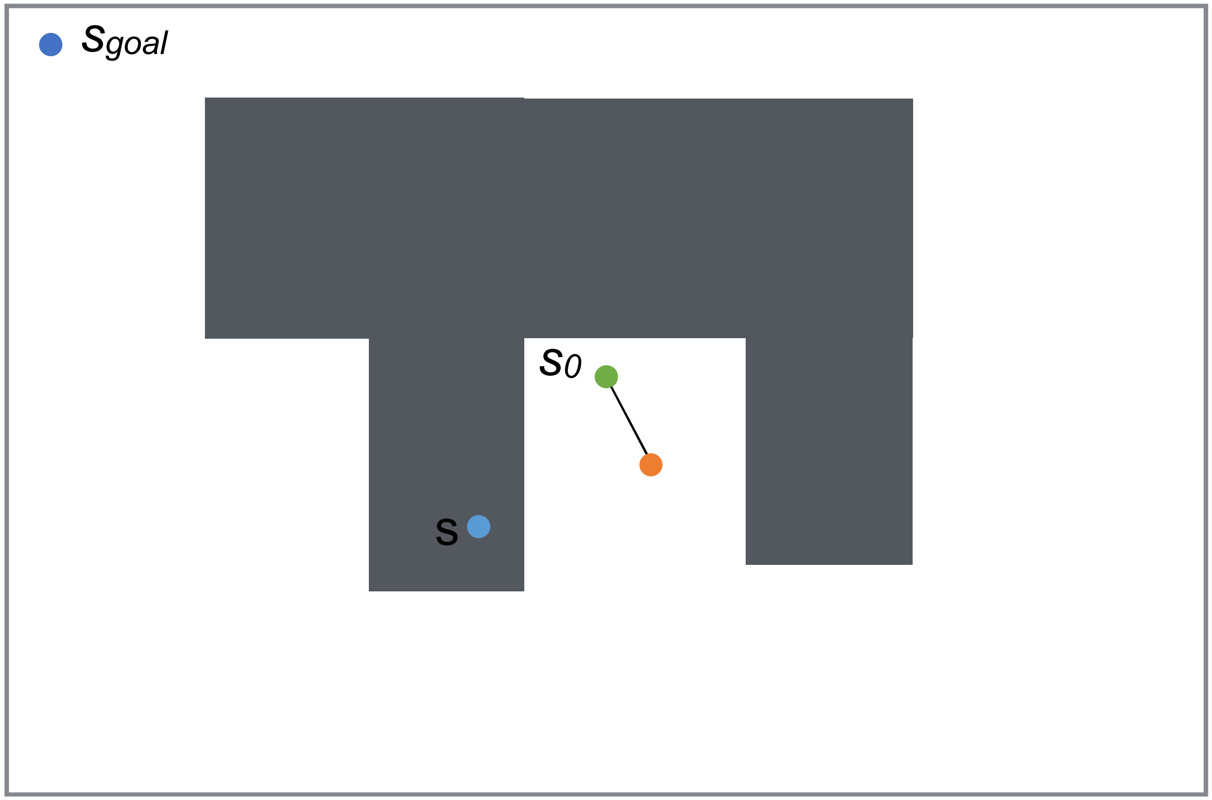
- Sample states $s \in S = R^{20}$ until s is collision-free
- **Find closest state $s_c \in T$**
- **Extend s_c toward s**
- **Add resulting state s' to T**
- Repeat until T contains a path from s_0 to s_{goal}



Rapidly Exploring Random Trees

Algorithm (input: s_0, s_{goal}, initial state tree T)

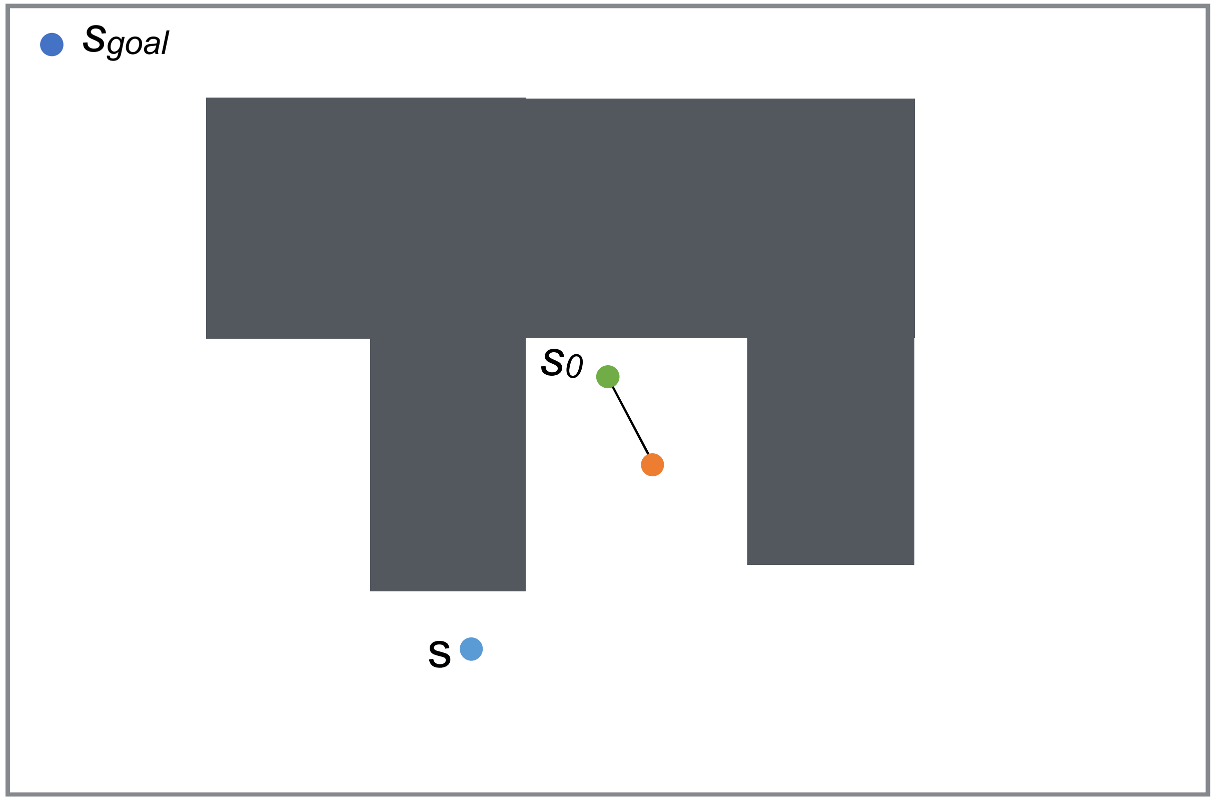
- **Sample states $s \in S = R^{20}$ until s is collision-free**
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



Rapidly Exploring Random Trees

Algorithm (input: s_0, s_{goal}, initial state tree T)

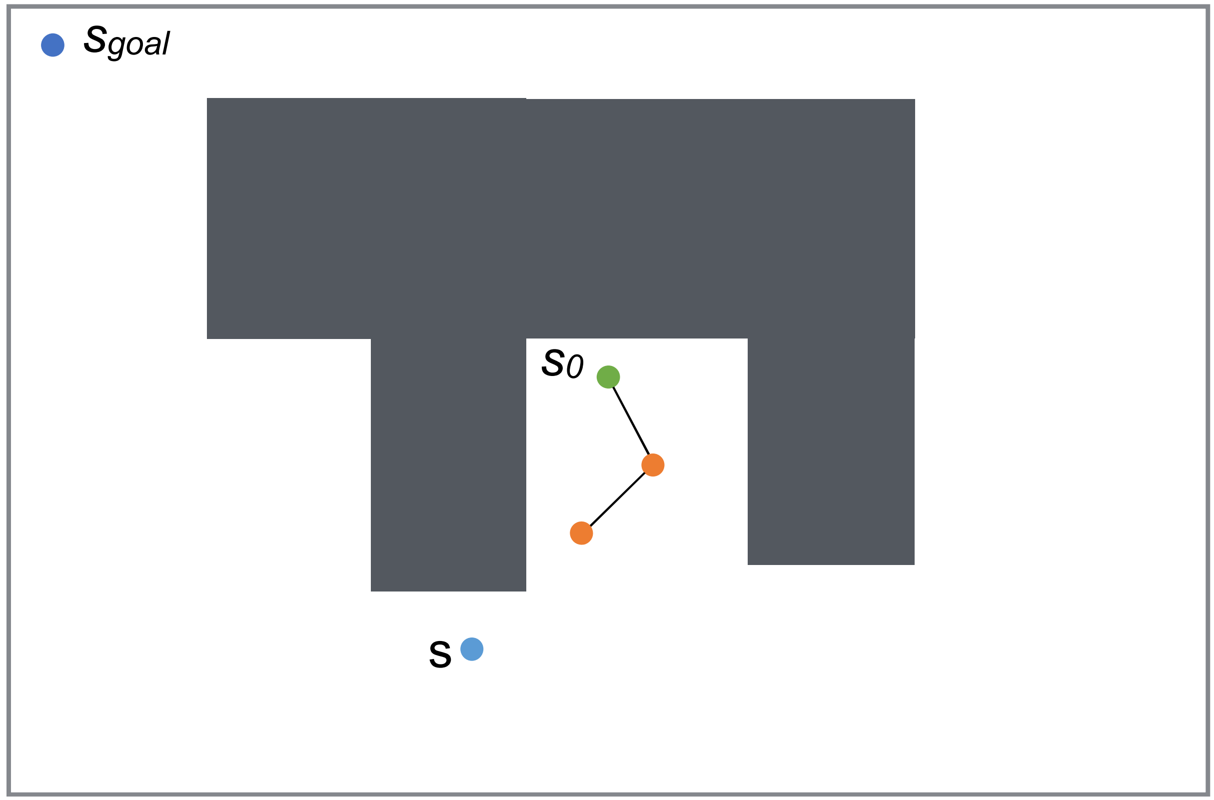
- **Sample states $s \in S = R^{20}$ until s is collision-free**
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



Rapidly Exploring Random Trees

Algorithm (input: s_0, s_{goal}, initial state tree T)

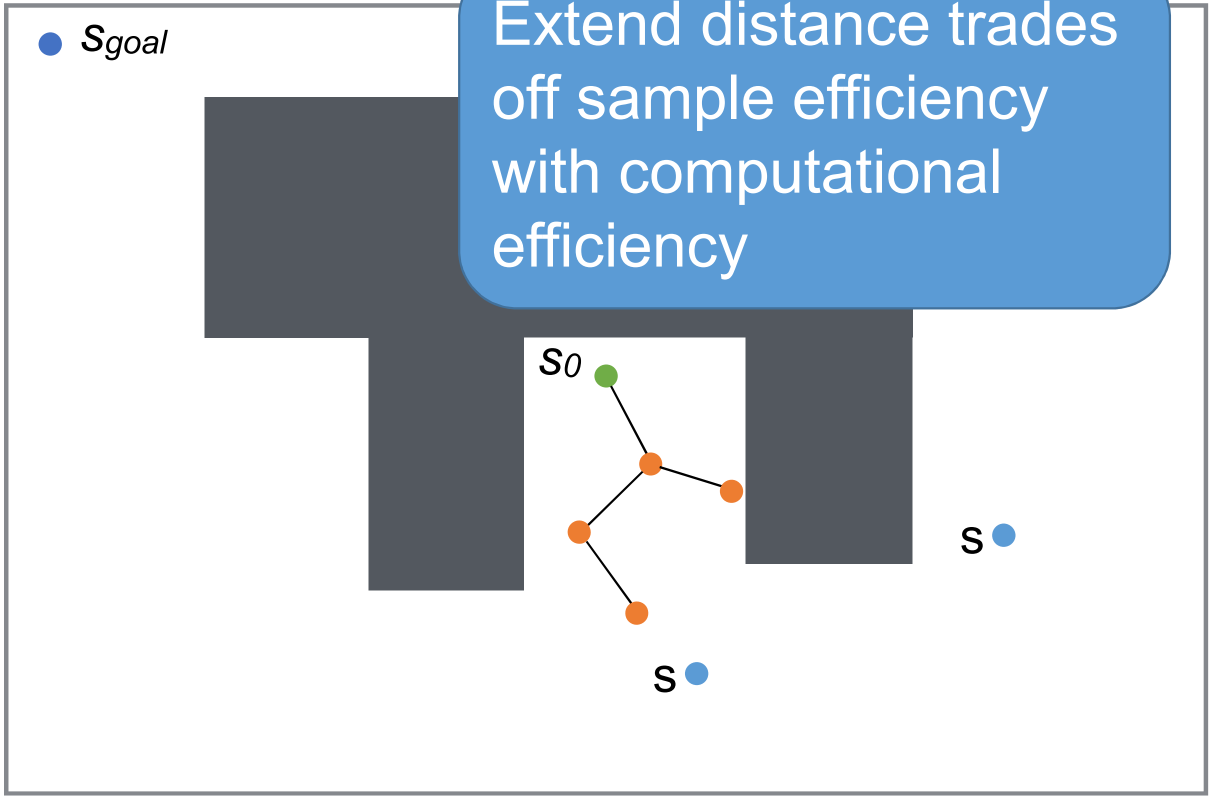
- Sample states $s \in S = R^{20}$ until s is collision-free
- **Find closest state $s_c \in T$**
- **Extend s_c toward s**
- **Add resulting state s' to T**
- Repeat until T contains a path from s_0 to s_{goal}



Uniform Sampling

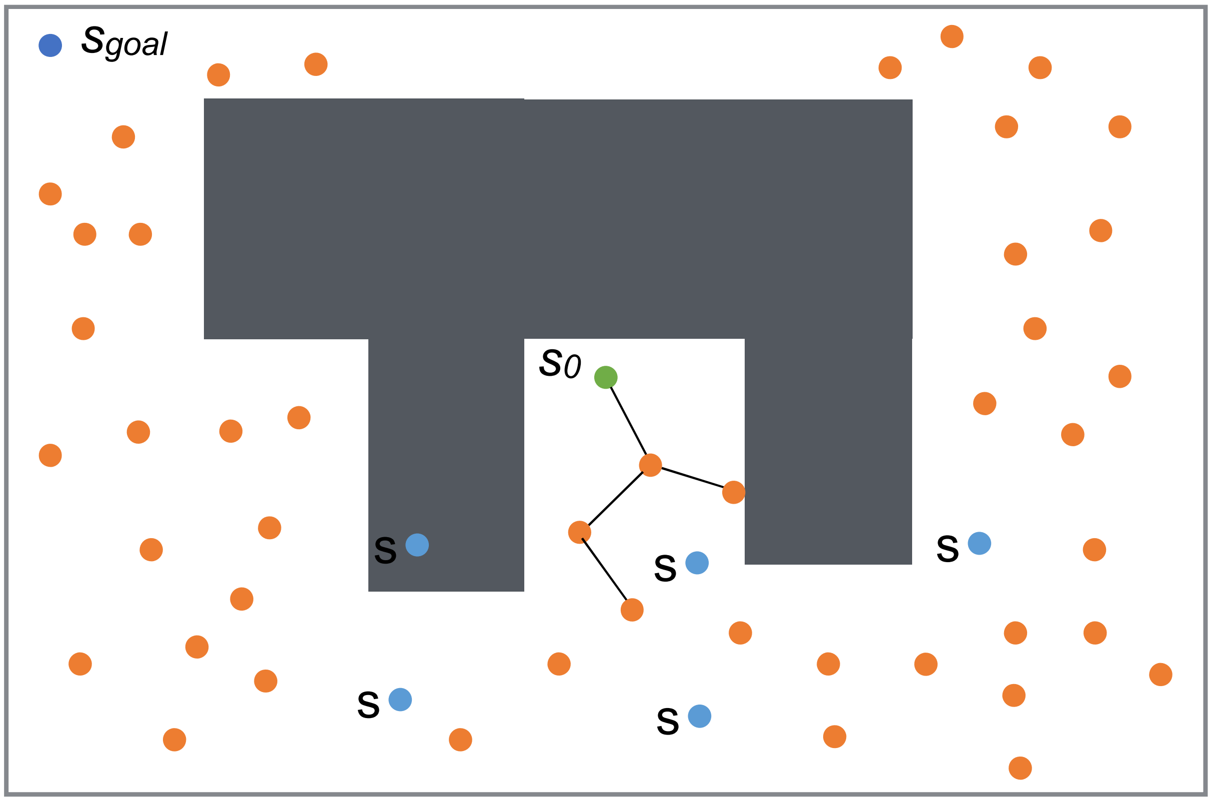
Algorithm (input: s_0, s_{goal}, initial state tree T)

- Sample states $s \in S = R^{20}$ until s is collision-free
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- **Repeat until T contains a path from s_0 to s_{goal}**



Extend distance trades off sample efficiency with computational efficiency

Uniform Sampling



Properties of RRT

Key idea: **random sampling** will naturally reduce the size of Voronoi regions, roughly prioritized by region size **encouraging exploration**

Properties of RRT

Key idea: **random sampling** will naturally reduce the size of Voronoi regions, roughly prioritized by region size **encouraging exploration**

RRT is **probabilistically complete**!

- If there's a solution it will find it eventually
- Can still be slow for some problems, but it is faster than naive action sampling approach

Properties of RRT

Key idea: **random sampling** will naturally reduce the size of Voronoi regions, roughly prioritized by region size **encouraging exploration**

RRT is **probabilistically complete**!

- If there's a solution it will find it eventually
- Can still be slow for some problems, but it is faster than naive action sampling approach

Q: Is this algorithm optimal?

Properties of RRT

Key idea: **random sampling** will naturally reduce the size of Voronoi regions, roughly prioritized by region size **encouraging exploration**

RRT is **probabilistically complete**!

- If there’s a solution it will find it eventually
- Can still be slow for some problems, but it is faster than naive action sampling approach

Not optimal (cost of paths are not considered)

- This is an example of “**feasible motion planning**”: find a path

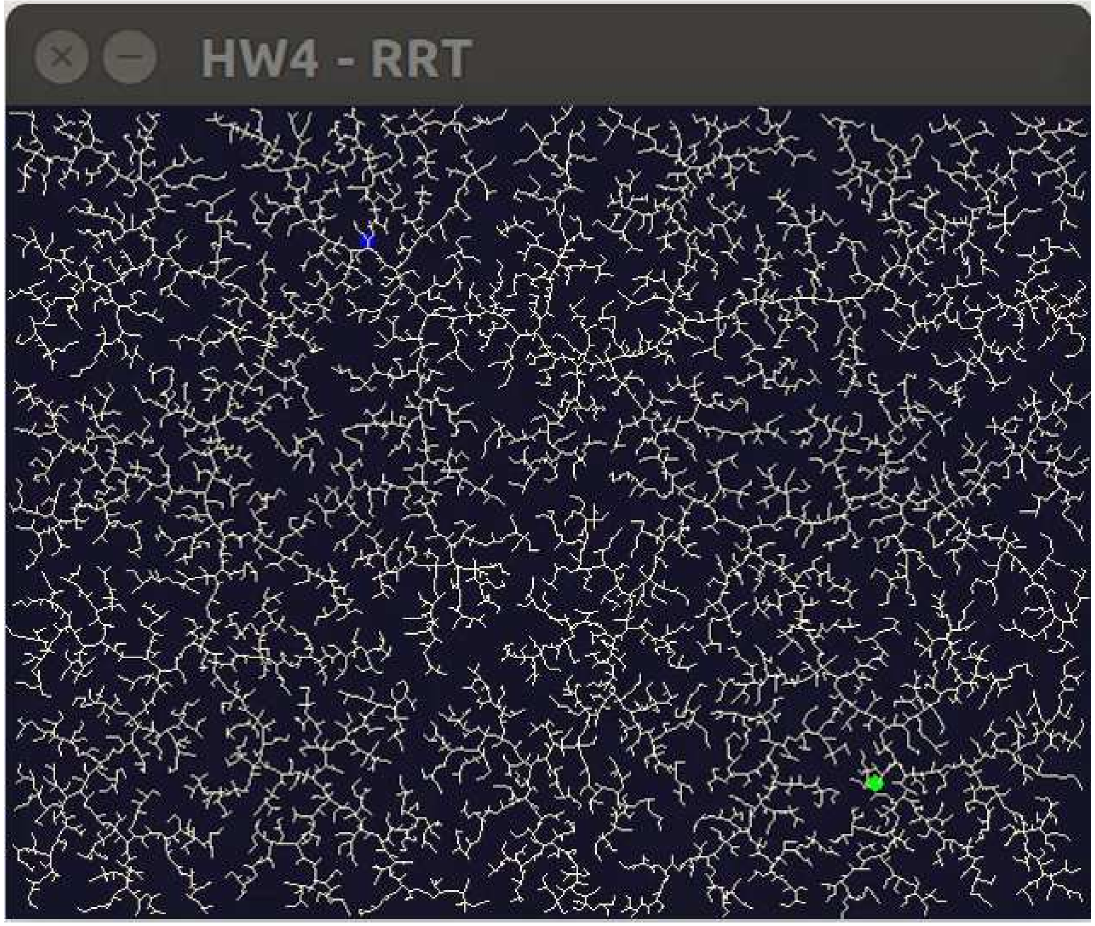
Rapidly Exploring Random Trees – Variants

Standard RRT (input: s_0, s_{goal}, initial state tree T)

- Sample states $s \in S = R^{20}$ until s is collision-free
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}

Rapidly Exploring Random Trees – Variants

Rapidly Exploring Random Trees – Variants



Q: What can we change to make this better?

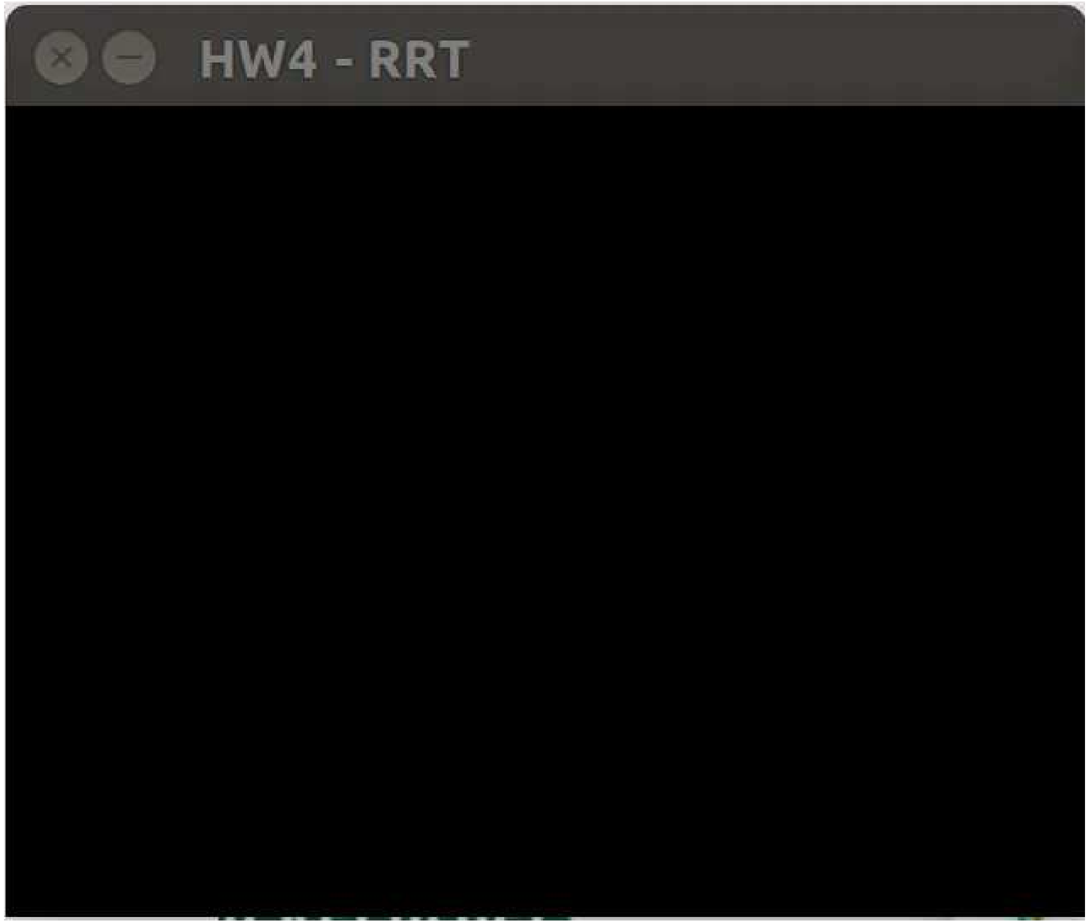
Rapidly Exploring Random Trees – Variants

Goal Directed Sampling (input: s_0, s_{goal}, initial state tree T)

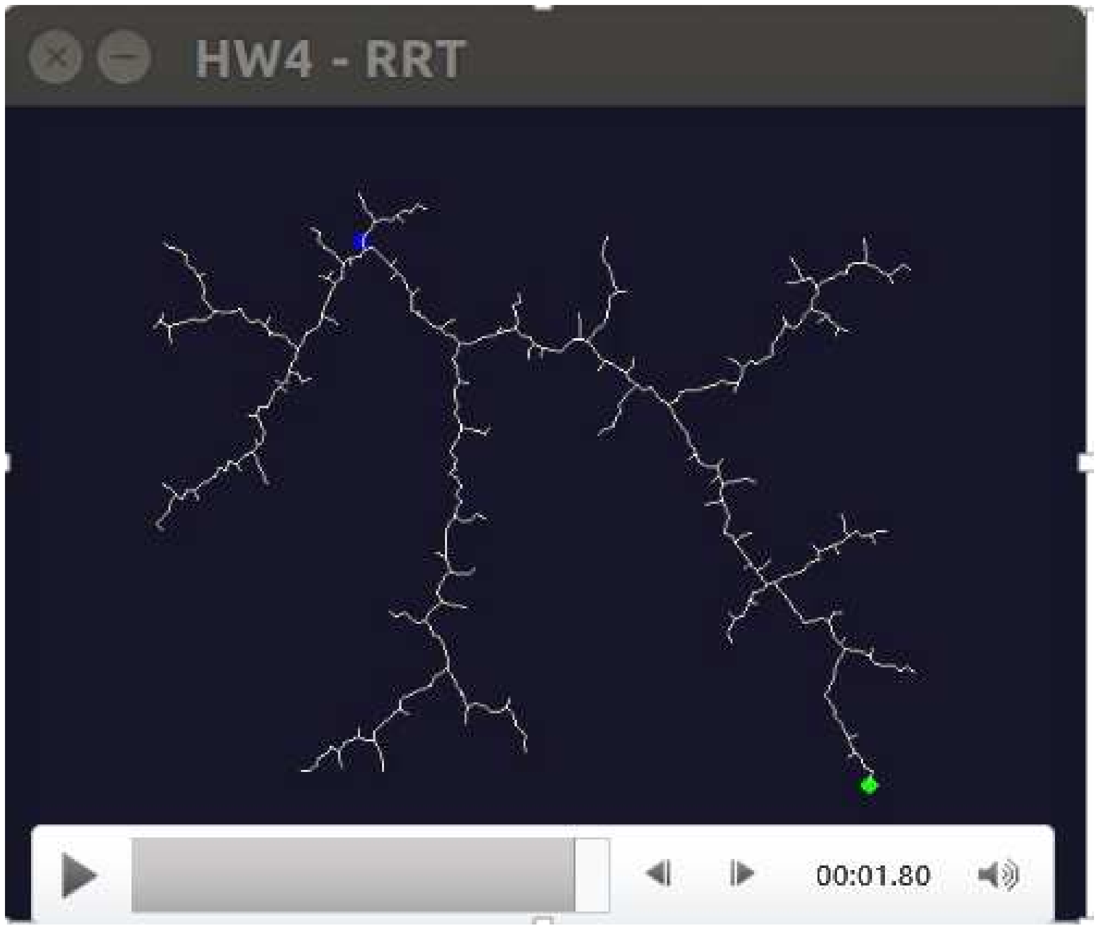
- Sample states $s \in S = R^{20}$ until s is collision-free **but with probability p sample the goal instead of a random point**
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}

Intuition: instead of "stumbling" upon the solution, bias the tree growth in the goal direction

Rapidly Exploring Random Trees – Variants



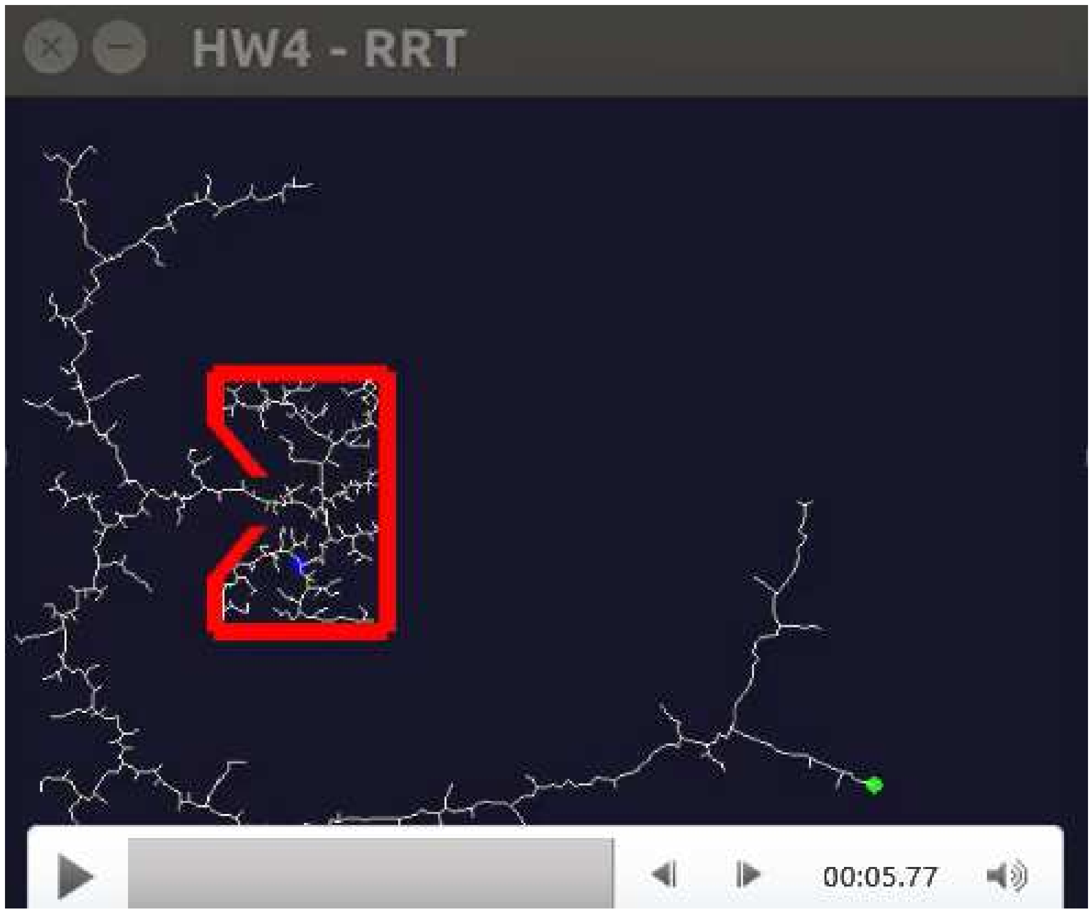
Rapidly Exploring Random Trees – Variants



Rapidly Exploring Random Trees – Variants

Q: How could we avoid this problem?

Rapidly Exploring Random Trees – Variants

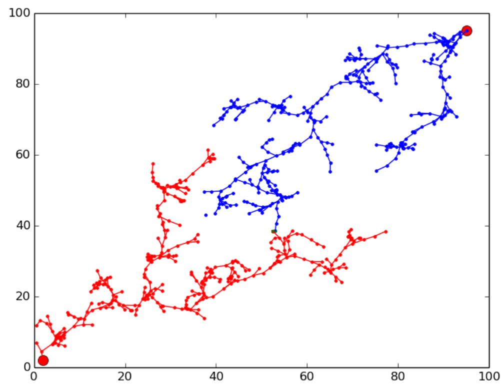


Q: How could we avoid this problem?

Rapidly Exploring Random Trees – Variants

Bidirectional RRT (input: s_0, s_{goal}, initial state trees T_1, T_2)

- Sample states $s \in S = R^{20}$ until s is collision-free
- Find closest state $s_c \in T_1$
- Extend s_c toward s
- Add resulting state s' to T_1

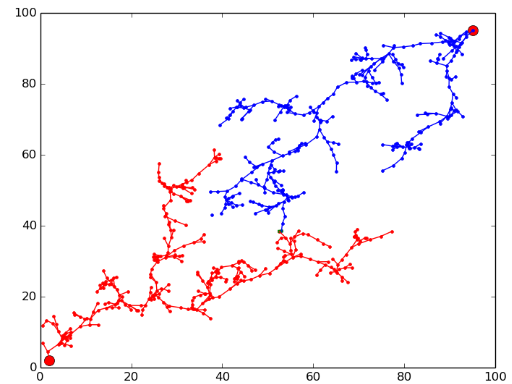


Intuition: search from one direction is sometimes easier than the other

Rapidly Exploring Random Trees – Variants

Bidirectional RRT (input: s_0, s_{goal}, initial state trees T_1, T_2)

- Sample states $s \in S = R^{20}$ until s is collision-free
- Find closest state $s_c \in T_1$
- Extend s_c toward s
- Add resulting state s' to T_1
- **Find closest state $s_{c2} \in T_2$ to s'**
- **Extend s_{c2} toward s'**
- **Add resulting state s'' to T_2**
- **If $s'' == s'$ and return a path from s_0 to s_{goal}**
- **Else *Swap(T_1, T_2)***



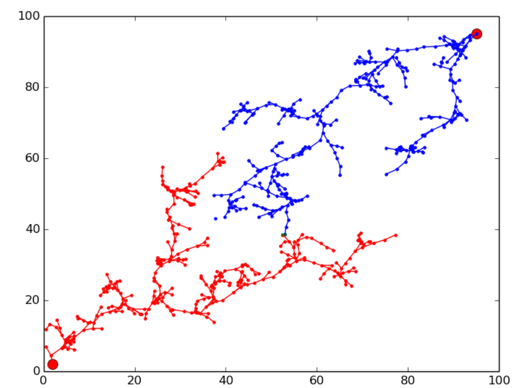
Intuition: search from one direction is sometimes easier than the other

Rapidly Exploring Random Trees – Variants

Bidirectional RRT (input: s_0, s_{goal}, initial state trees T_1, T_2)

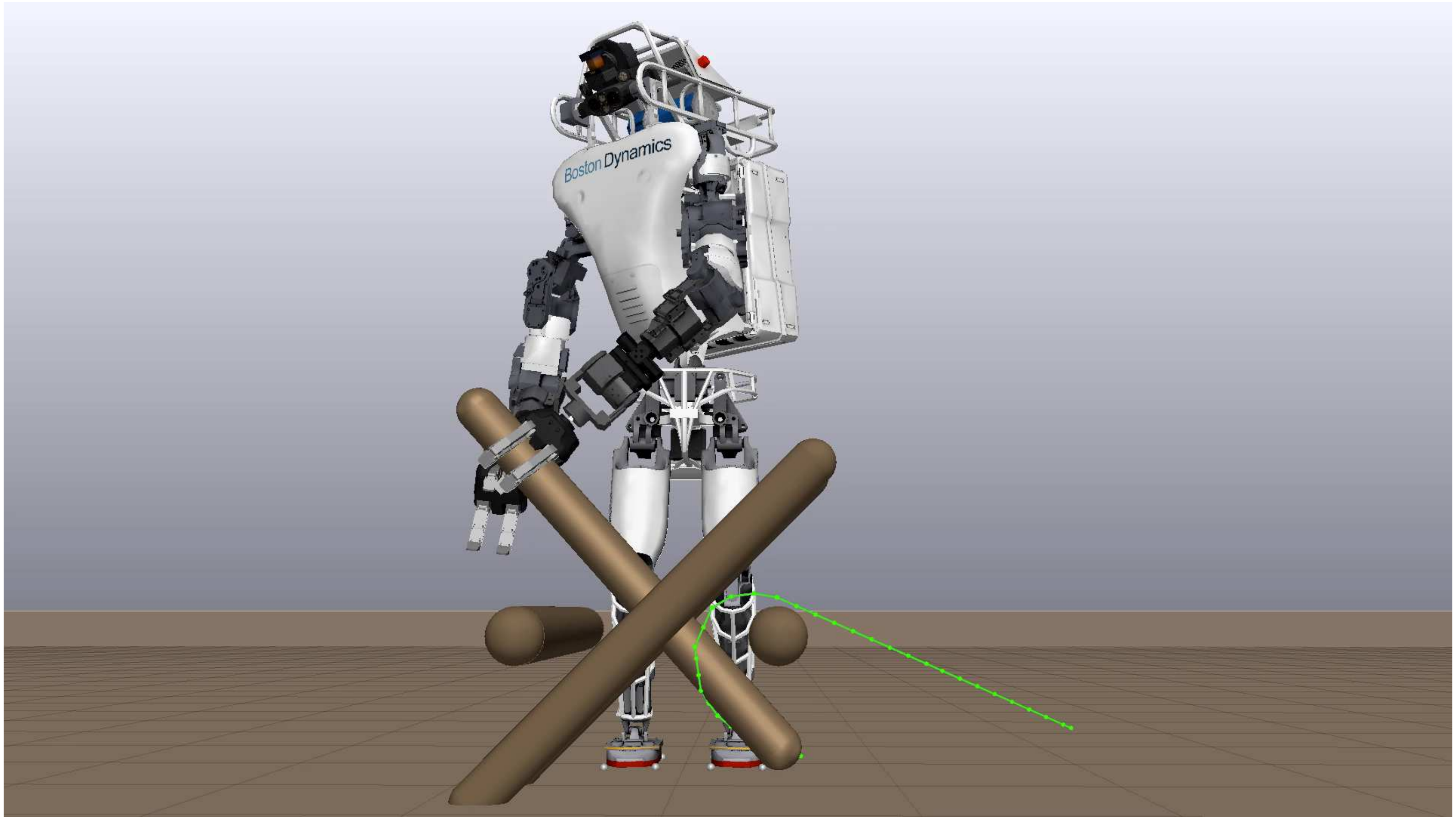
- Sample states $s \in S = R^{20}$ until s is collision-free
- Find closest state $s_c \in T_1$
- Extend s_c toward s
- Add resulting state s' to T_1
- **Find closest state $s_{c2} \in T_2$ to s'**
- **Extend s_{c2} toward s'**
- **Add resulting state s'' to T_2**
- **If $s'' == s'$ and return a path from s_0 to s_{goal}**
- **Else *Swap(T_1, T_2)***

Can also "balance" trees by swapping T_1, T_2 based on size

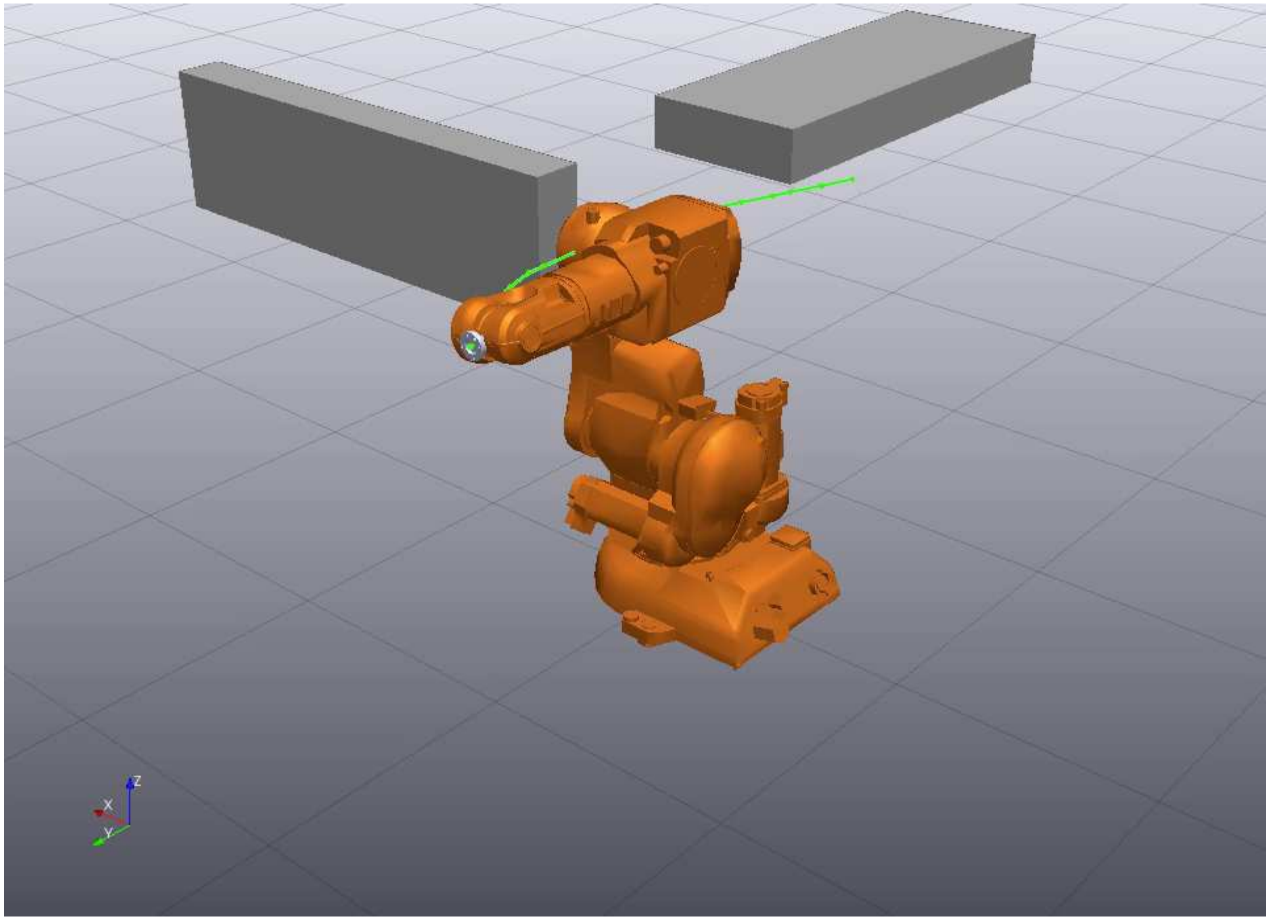


Intuition: search from one direction is sometimes easier than the other

RRT often works really well in practice



RRT often works really well in practice



Sometimes Paths are Weird

What if we search the same state space repeatedly?

RRT (a “**single-query**” algorithm) would become very inefficient as we would “forget” all of the possible connections we learned in the previous iteration

What if we search the same state space repeatedly?

RRT (a "**single-query**" algorithm) would become very inefficient as we would "forget" all of the possible connections we learned in the previous iteration

What if instead of building a tree every time we want to move, we build a reusable graph ***G*** of sampled states?

What if we search the same state space repeatedly?

RRT (a "**single-query**" algorithm) would become very inefficient as we would "forget" all of the possible connections we learned in the previous iteration

What if instead of building a tree every time we want to move, we build a reusable graph ***G*** of sampled states?

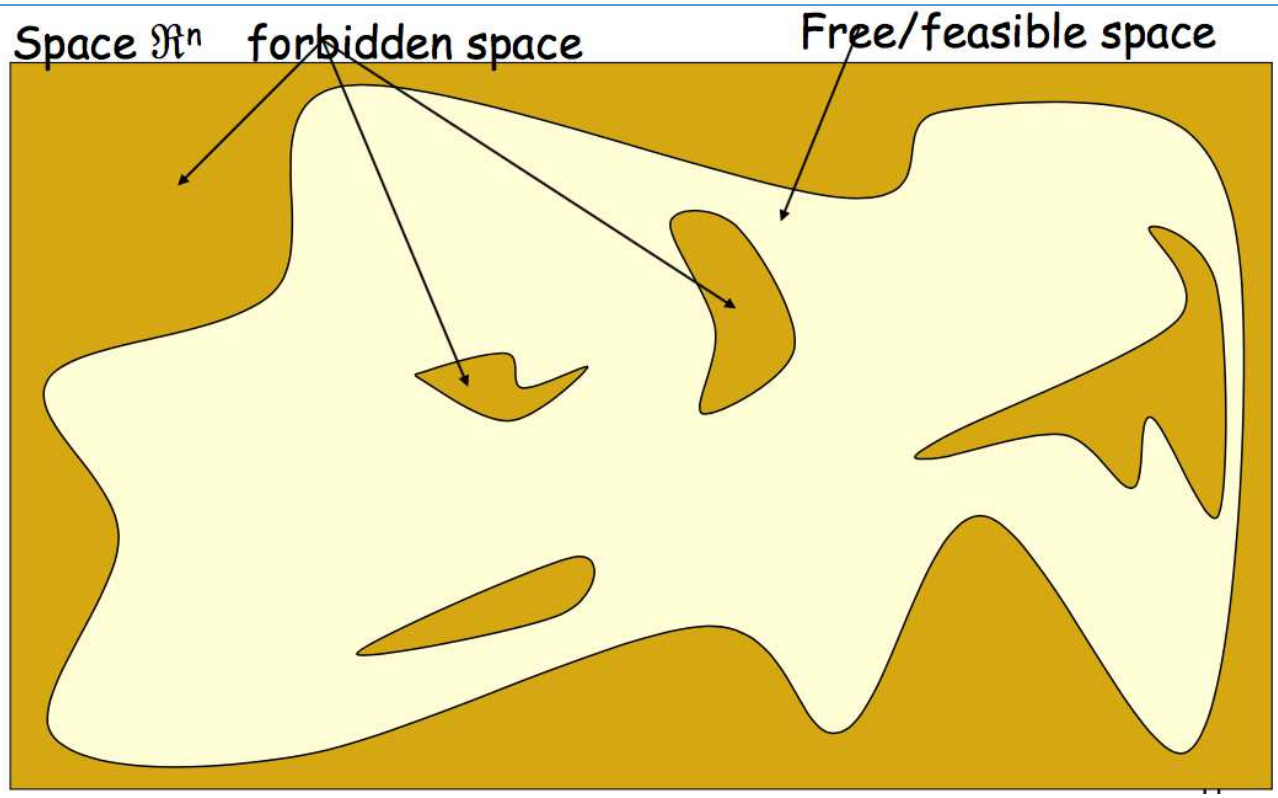
This "**multi-query**" approach is called **Probabilistic Roadmaps (PRMs)**

Probabilistic Roadmaps (PRMs) leverage an offline and an online computation phase

Step 1: Offline build a random graph ***G*** that covers the state space

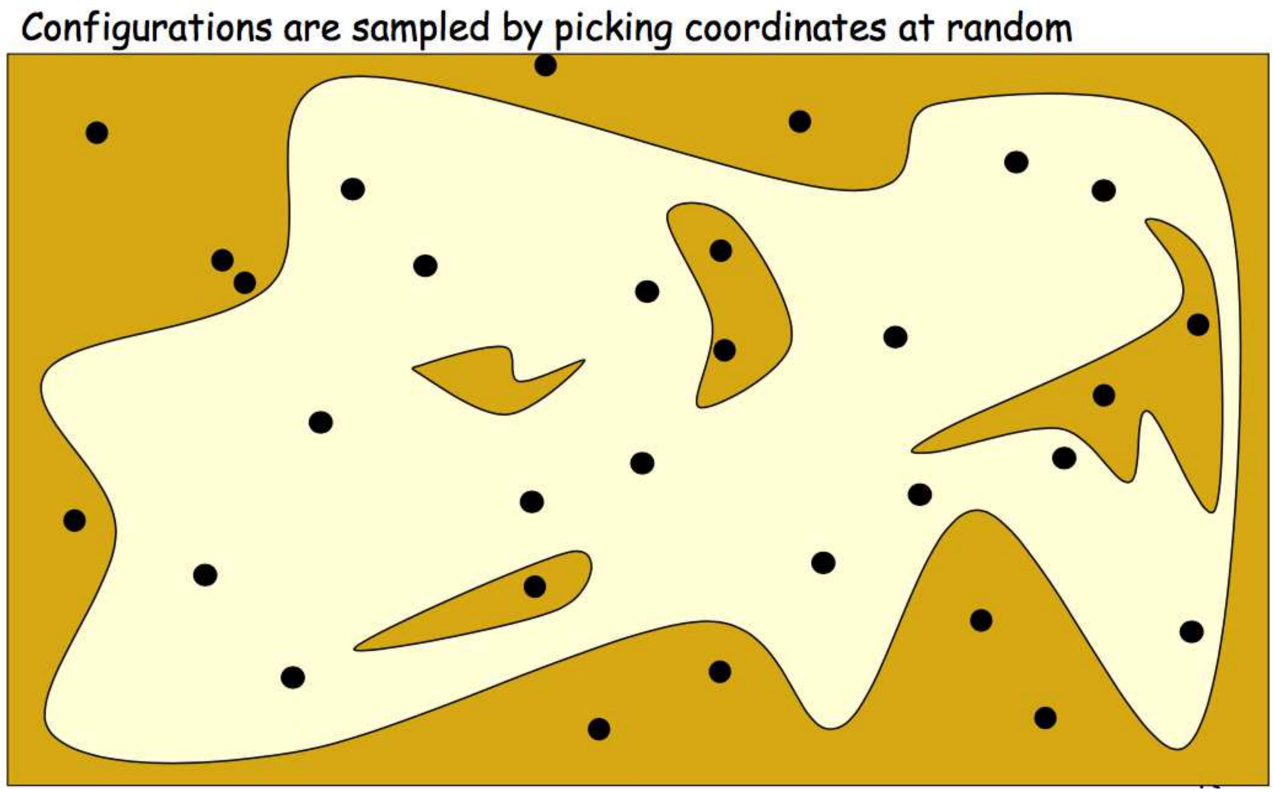
Probabilistic Roadmaps (PRMs) leverage an offline and an online computation phase

Step 1: Offline build a random graph ***G*** that covers the state space



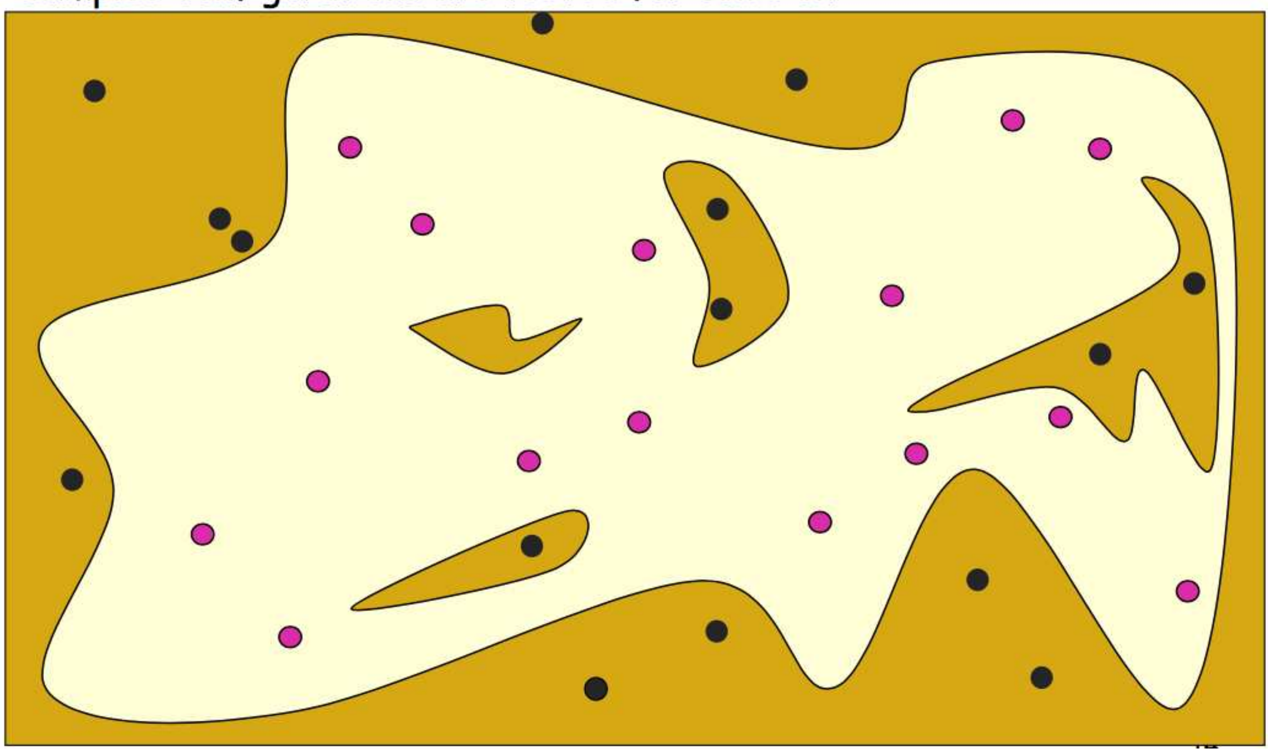
Probabilistic Roadmaps (PRMs) leverage an offline and an online computation phase

Step 1: Offline build a random graph ***G*** that covers the state space



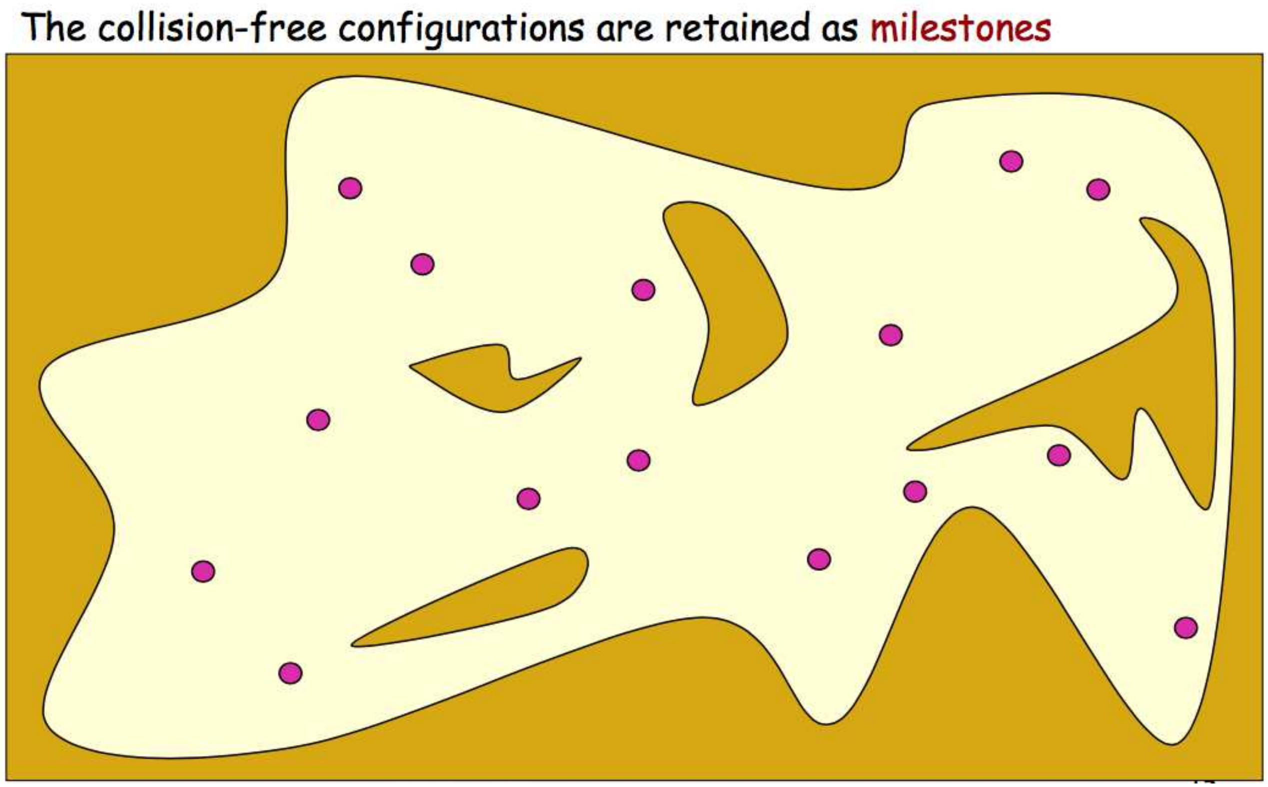
Probabilistic Roadmaps (PRMs) leverage an offline and an online computation phase

Step 1: Offline build a random graph ***G*** that covers the state space



Probabilistic Roadmaps (PRMs) leverage an offline and an online computation phase

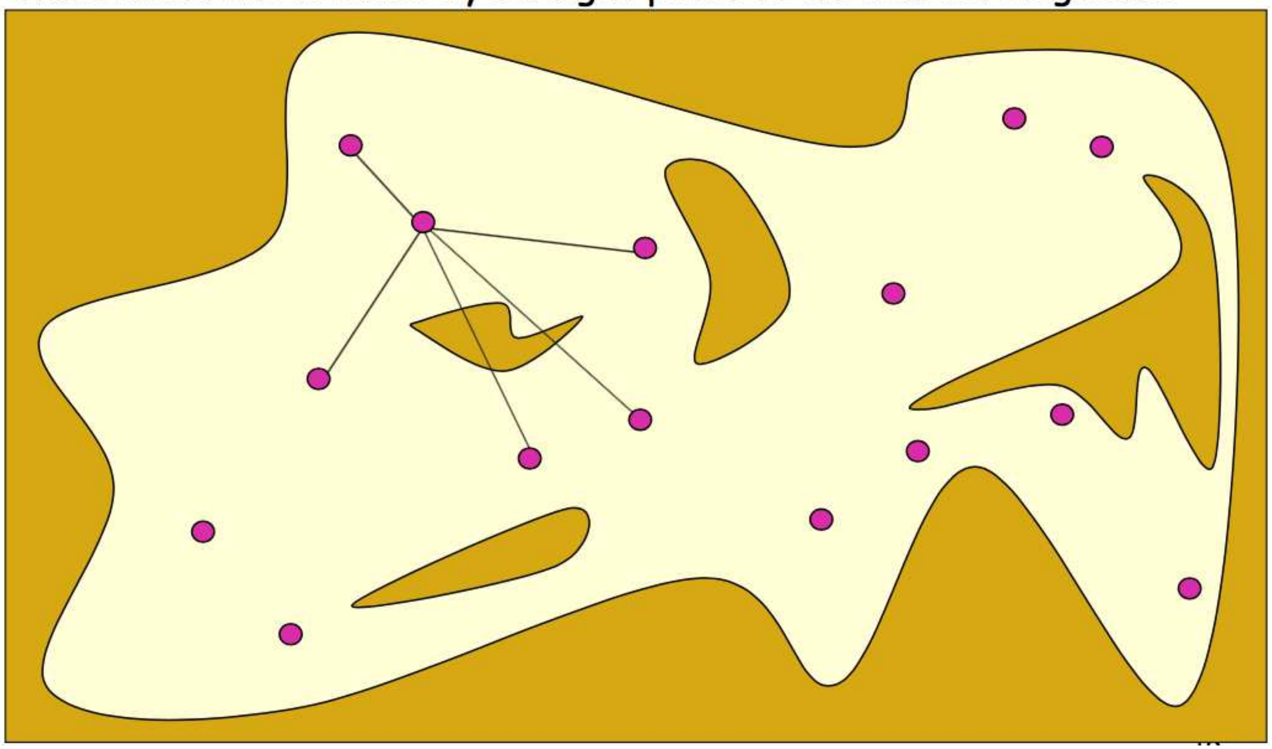
Step 1: Offline build a random graph ***G*** that covers the state space



Probabilistic Roadmaps (PRMs) leverage an offline and an online computation phase

Step 1: Offline build a random graph ***G*** that covers the state space

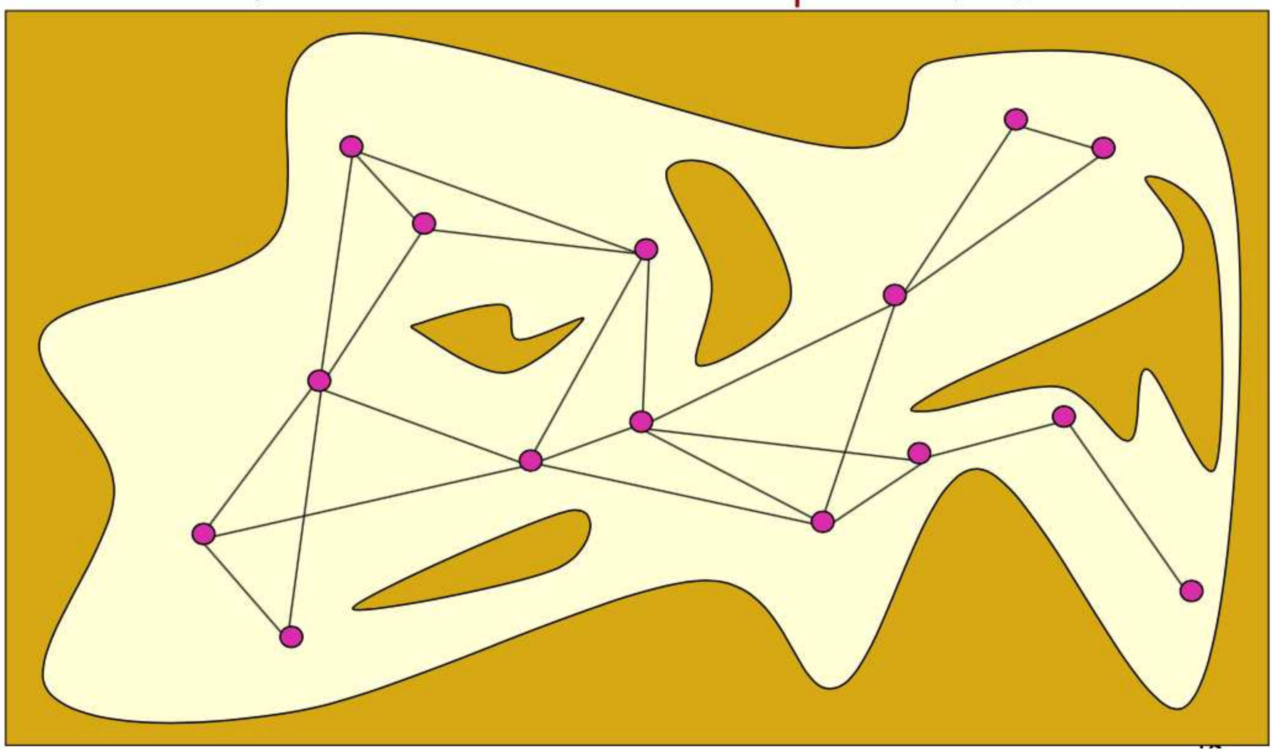
Each milestone is linked by straight paths to its nearest neighbors



Probabilistic Roadmaps (PRMs) leverage an offline and an online computation phase

Step 1: Offline build a random graph ***G*** that covers the state space

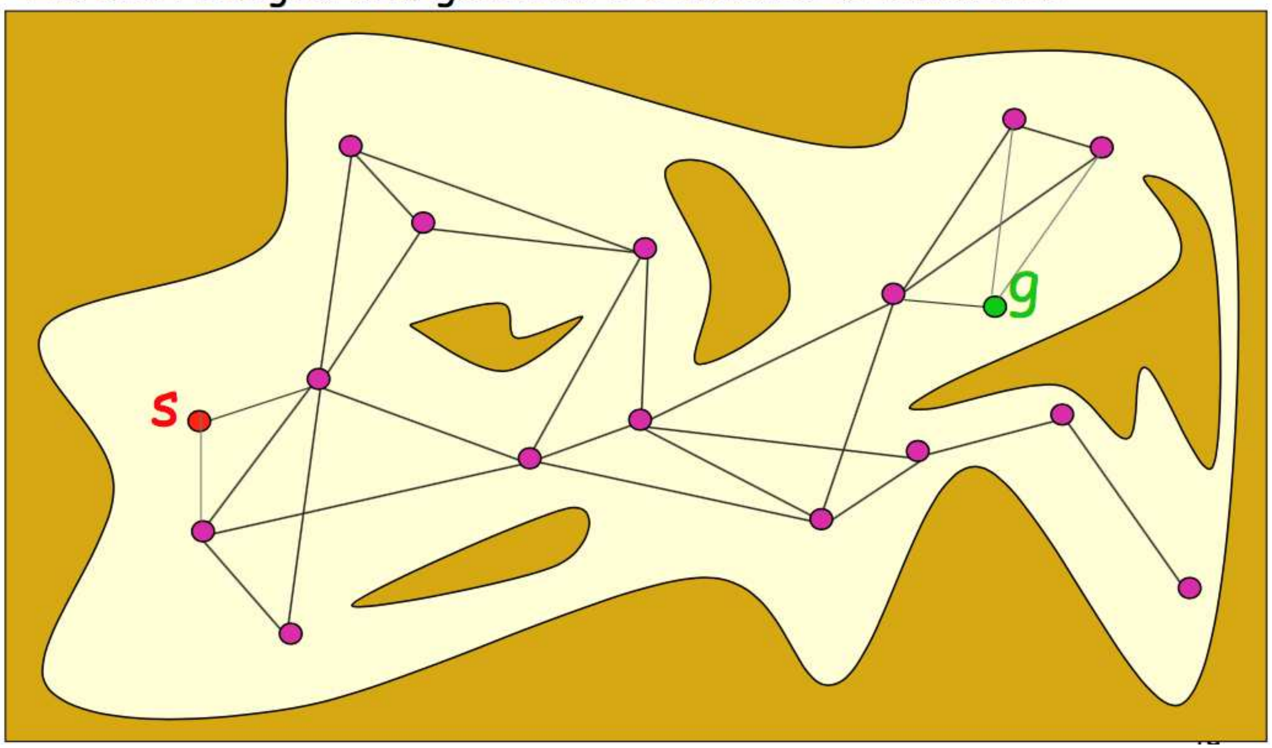
The collision-free links are retained as local paths to form the PRM



Probabilistic Roadmaps (PRMs) leverage an offline and an online computation phase

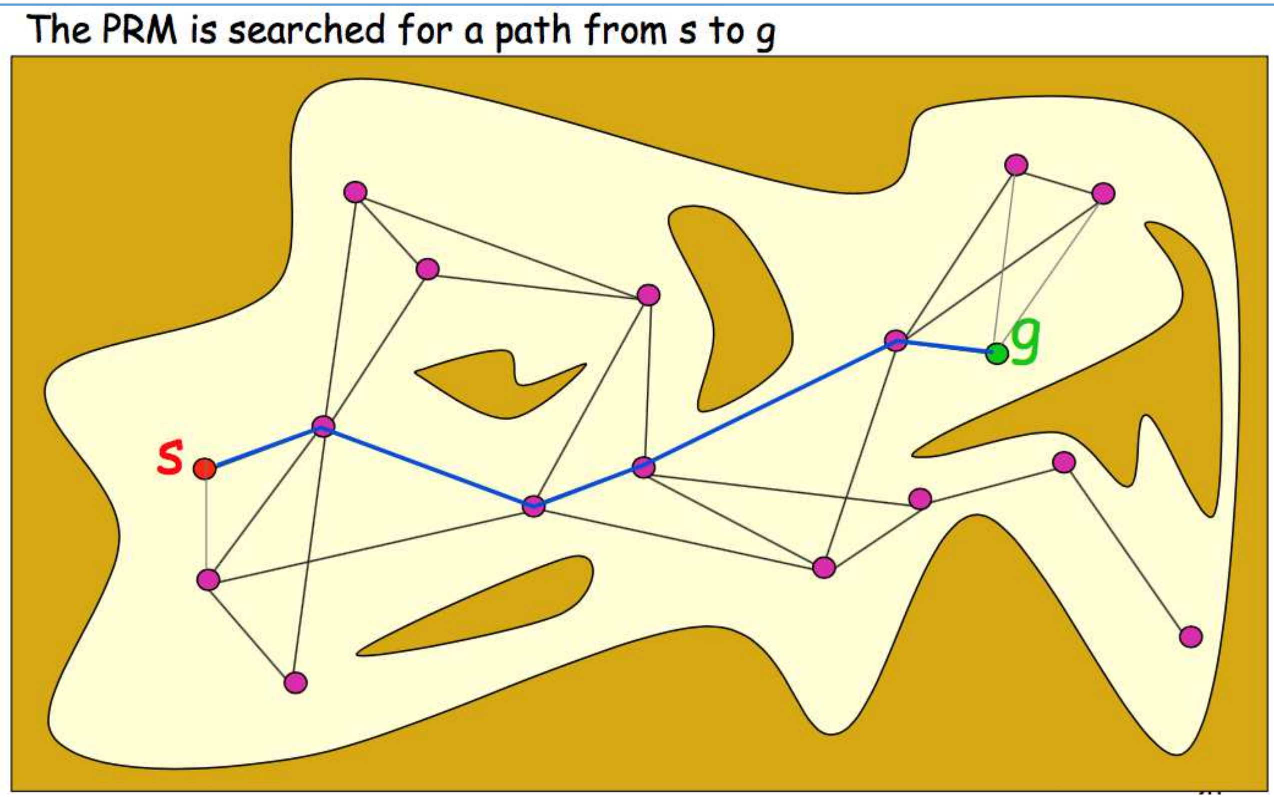
Step 2: Online connect the start and goal nodes and run graph search

The start and goal configurations are included as milestones



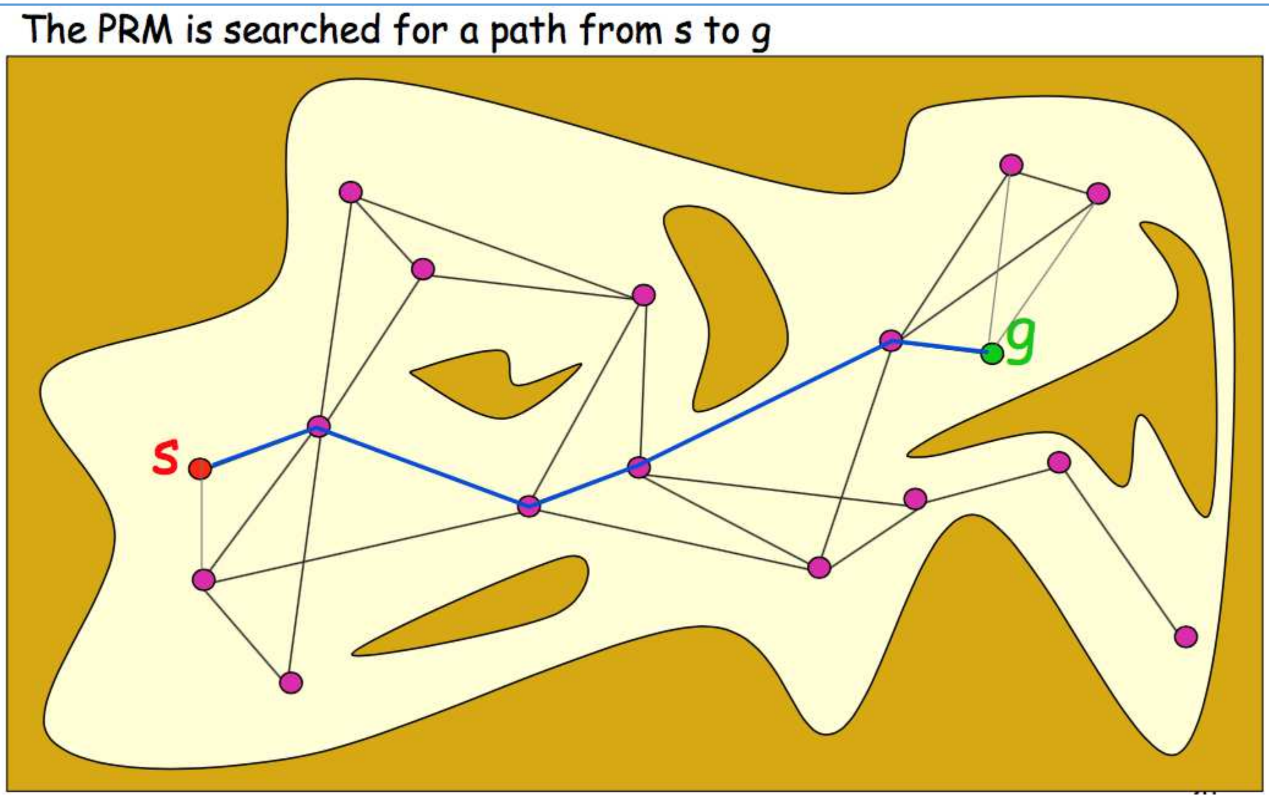
Probabilistic Roadmaps (PRMs) leverage an offline and an online computation phase

Step 2: Online connect the start and goal nodes and run graph search



Probabilistic Roadmaps (PRMs) leverage an offline and an online computation phase

Step 2: Online connect the start and goal nodes and run graph search



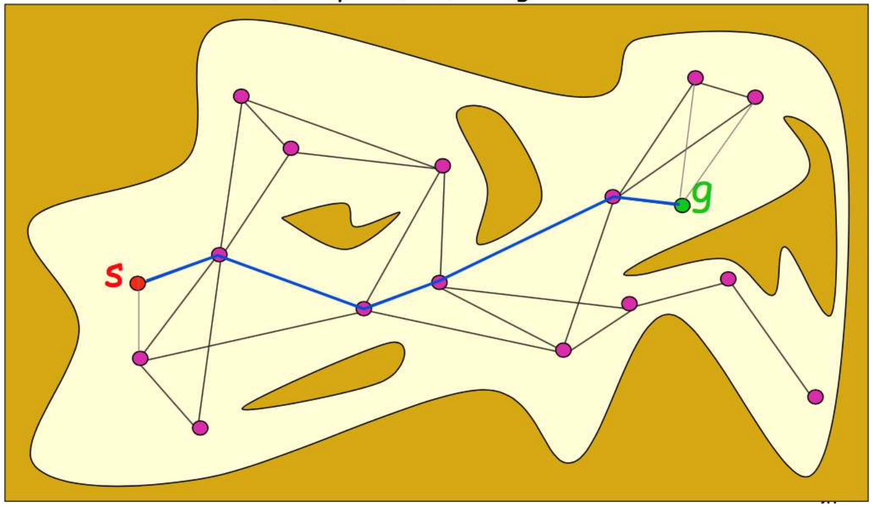
Q: Is this optimal?
Is this complete?

PRM Considerations

What if it fails?

- Maybe the roadmap was not adequate
- Could spend more time in the sampling/graph-building phase
- Could do another sampling phase and reuse *G*
- Sampling and query phases don't have to be executed sequentially

The PRM is searched for a path from s to g

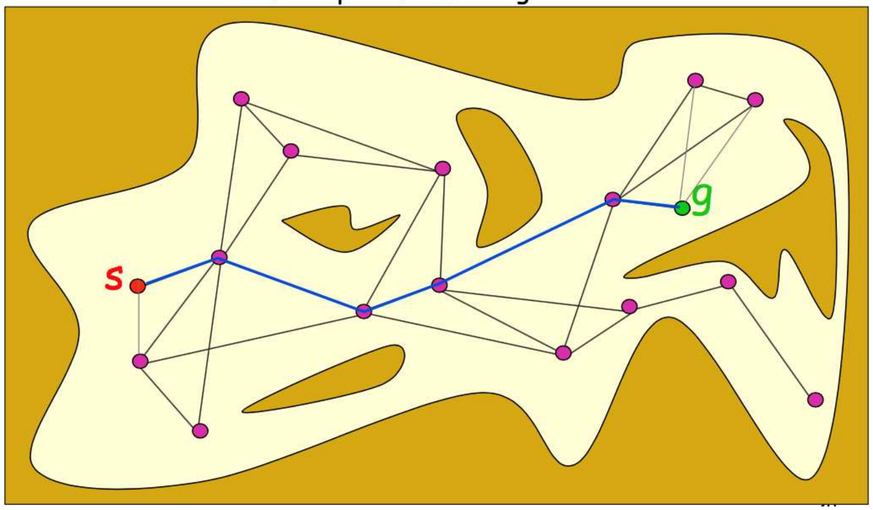


PRM Considerations

What if it fails?

- Maybe the roadmap was not adequate
- Could spend more time in the sampling/graph-building phase
- Could do another sampling phase and reuse *G*
- Sampling and query phases don't have to be executed sequentially

The PRM is searched for a path from s to g



Inherent tradeoff between offline and online computational effort!

Challenges with RRTs & PRMs

1. **Sampling effectively is hard**
   - Sometimes uniform coverage of the state space isn't what we want (e.g., if there are many unreachable regions)

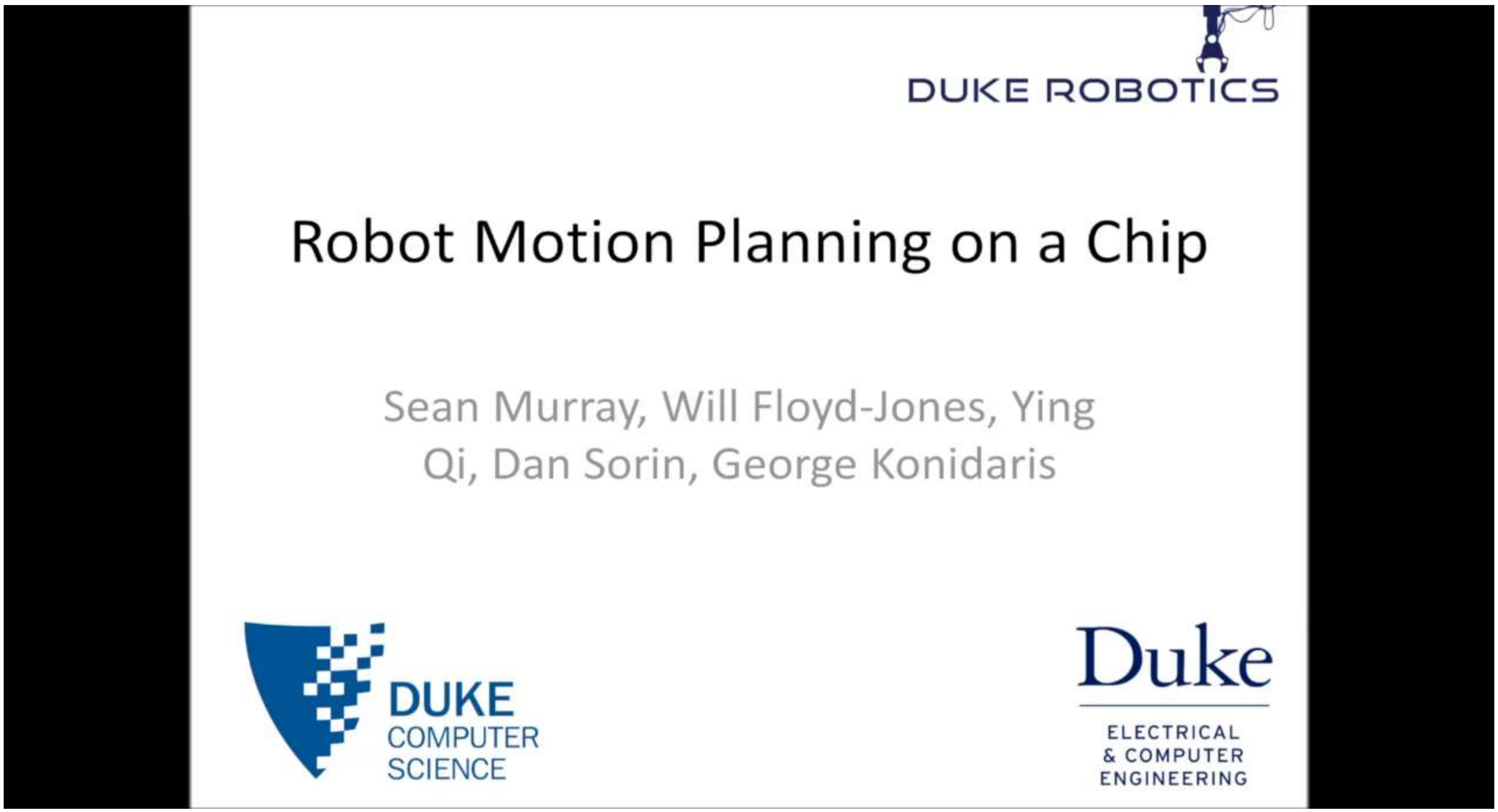
Challenges with RRTs & PRMs

1. **Sampling effectively is hard**
   - Sometimes uniform coverage of the state space isn't what we want (e.g., if there are many unreachable regions)
2. **Connecting neighboring points can get complicated**
   - Remember from earlier we need to use forward kinematics to check task space obstacle collisions! And complex geometries make this even harder!

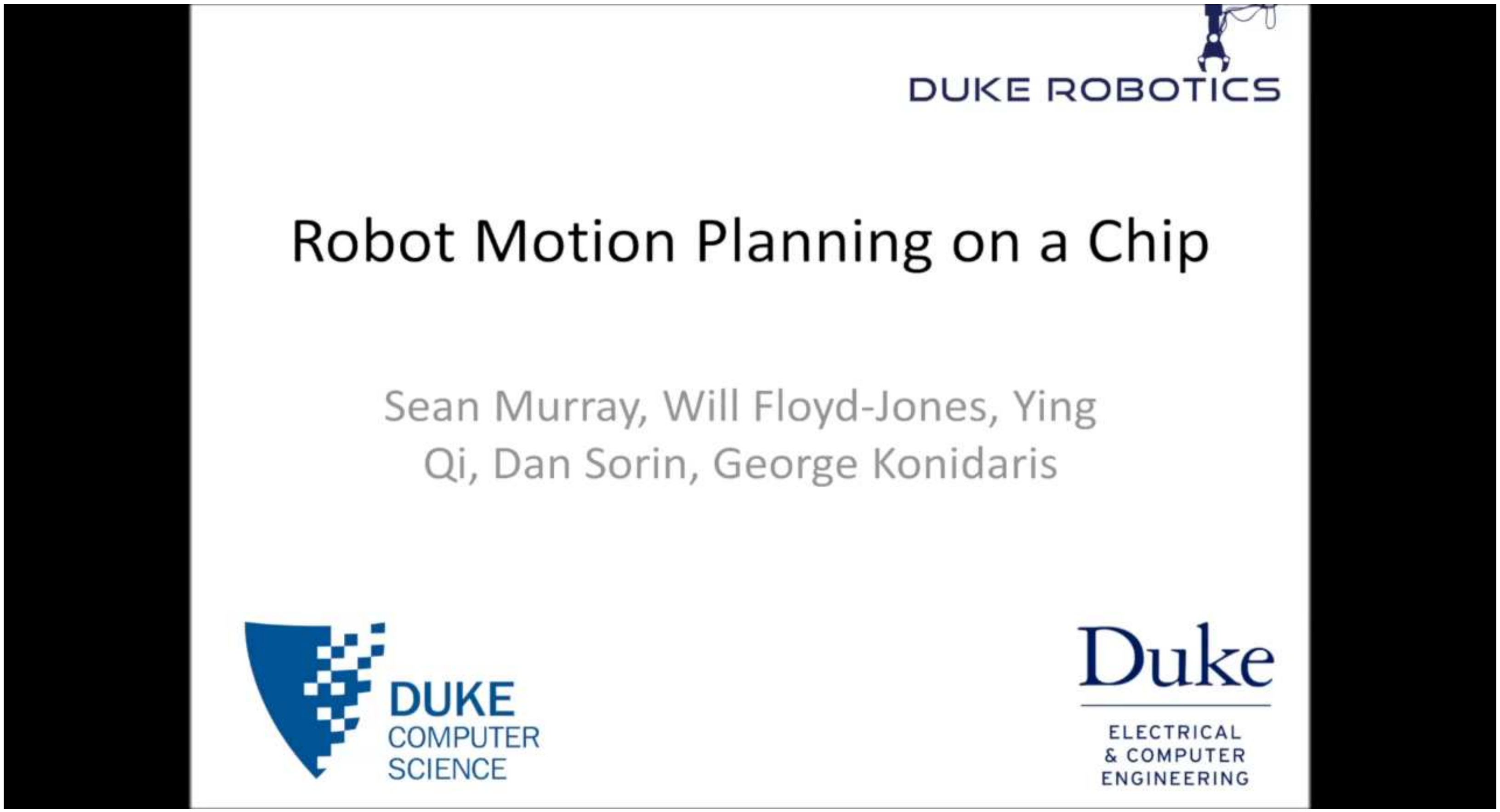
Challenges with RRTs & PRMs

1. **Sampling effectively is hard**
   - Sometimes uniform coverage of the state space isn't what we want (e.g., if there are many unreachable regions)
2. **Connecting neighboring points can get complicated**
   - Remember from earlier we need to use forward kinematics to check task space obstacle collisions! And complex geometries make this even harder!
   - If you can't simply draw straight lines between sample configurations, this step could involve a whole other optimization!

Solving part of the collision checking problem will get you your own startup!



Solving part of the collision checking problem will get you your own startup!

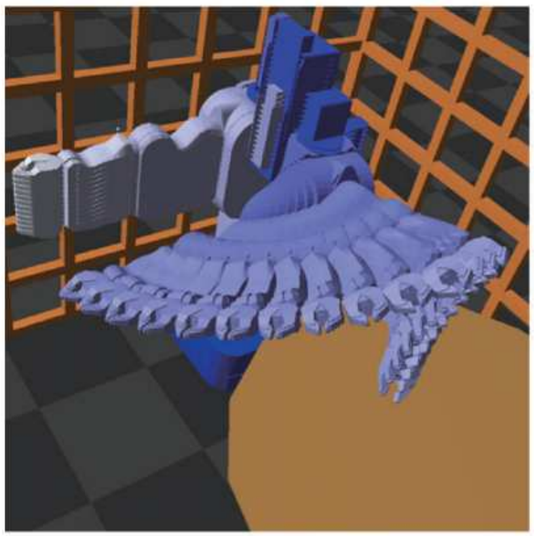


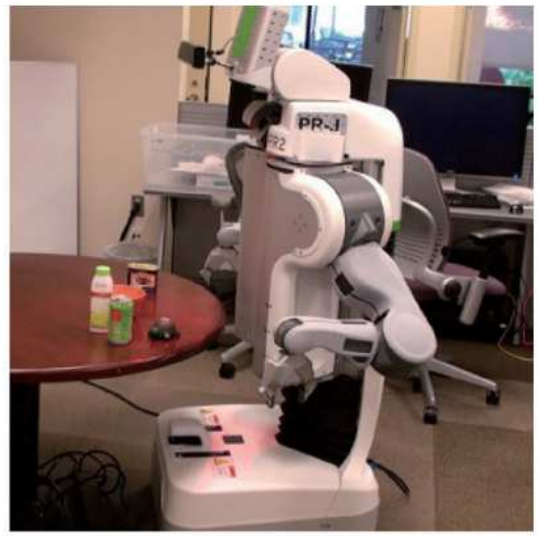
Summary

1. **Policies** are not feasible for most robots, so we **plan** instead
2. Robot planning usually involves thinking about both **task and configuration spaces**
3. **RRTs and PRMs**: powerful tools based on very simple ideas
   - **Probabilistically complete**
   - Hundreds of papers introducing variants and improvements to the basic idea
   - **Single-query (RRT) vs. Multi-query (PRM)**
4. For many real problems, **collision checking** can be expensive

CS182: Artificial Intelligence

Lecture 13: Robot Motion Planning II





Brian Plancher
Harvard University
Fall 2018

Slides adapted from
Scott Kuindersma

Announcements

- **Midterm 1 is in 1 week (10/29) during class in the normal classroom**
  - **Covers L1-L11, P1-P3, S1-S6**
  - **Midterm review** (no section this week)
    - Tuesday 4:30-6:30 SC Hall E
    - Sunday 12:00-2:00 in Pierce 301
  - If you have an AEO letter for extra time or have a conflict with the midterm you need to **let us know today** so we can ensure that we figure out appropriate accommodations!
- The Robotics material is on midterm 2 and Wednesday's guest lecture will have a problem on P4 so come!

Final Project Information is on Canvas!

	Aspect	Deadline
5%	Project Proposal	11/12, 11:59 PM
5%	Status Update	11/26, 11:59 PM
5%	Posters to Printer	12/7, 7:00 AM
	Poster Presentations	12/11, 12:00PM-3:00PM
80%	Final Project Report	12/18, 11:59 PM

Final Project Information is on Canvas!

- **Proposal – 5%**
  - Describe the problem
  - Identify the course related topics (aka what algorithms)
  - List your intended experiments
  - List papers / resources / outside code you intend to integrate with
  - How are you dividing the work?
  - Think of this as the first sections of your paper (abstract, background, motivation, related work)
- **Update – 5%**
- **Poster – 5%**
- **Report and Code – 85%**

Final Project Information is on Canvas!

- **Proposal – 5%**
- **Update – 5%**
  - How are you addressing your proposal feedback?
  - How have things been going? Any changes from the proposal?
- **Poster – 5%**
- **Report and Code – 85%**

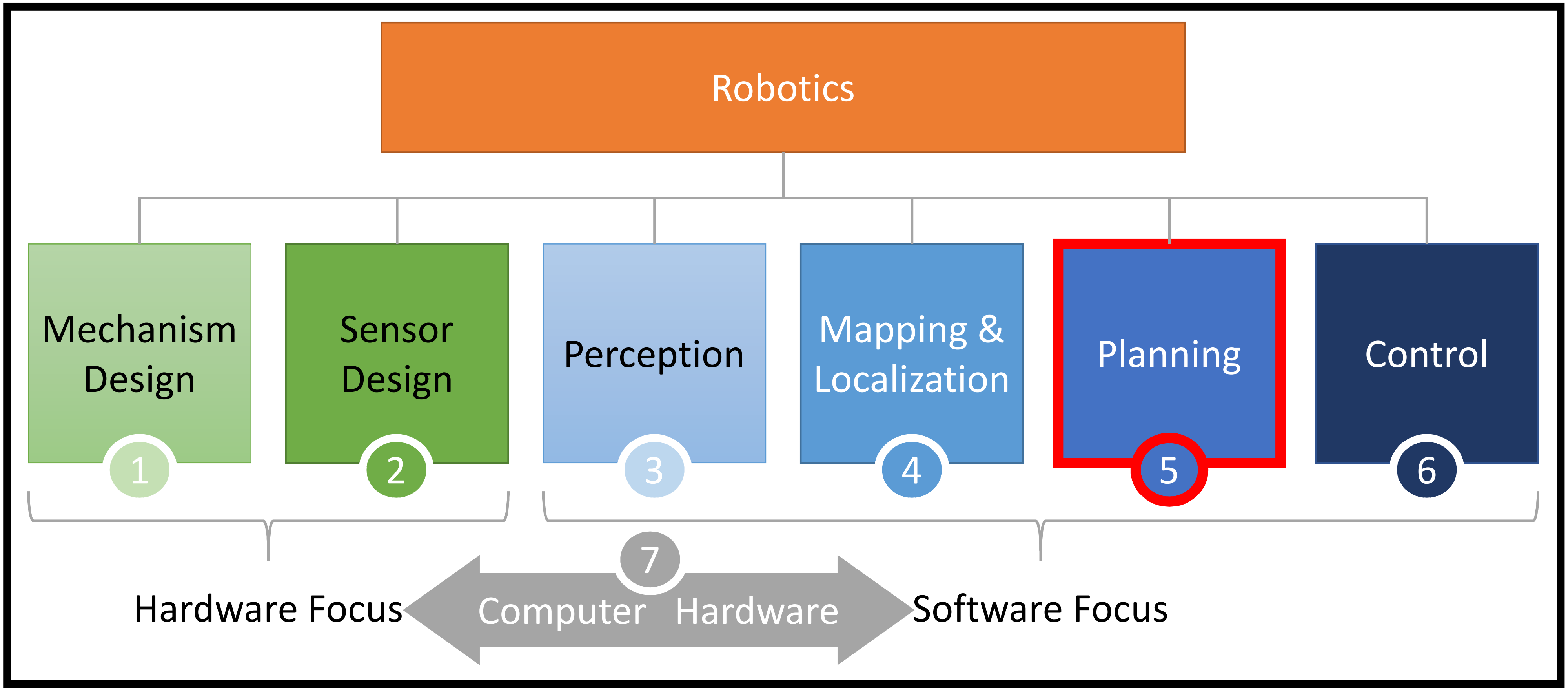
Final Project Information is on Canvas!

- **Proposal – 5%**
- **Update – 5%**
- **Poster – 5%**
  - Think of it as a way to walk the course staff through your coming paper
    - Algorithms explained, Graphs of experiments, Future work, etc.
  - Last chance to get feedback from the course staff and make sure you are on the right track for your final paper
  - Posters must be sent to MCB by 7am on Friday Dec 7th. Hard deadline.
    - Note: Midterm 2 is Dec 5th and presentation is Tuesday Dec 11th
    - Make sure to include all sections in the template (but can make prettier)
- **Report and Code – 85%**

Final Project Information is on Canvas!

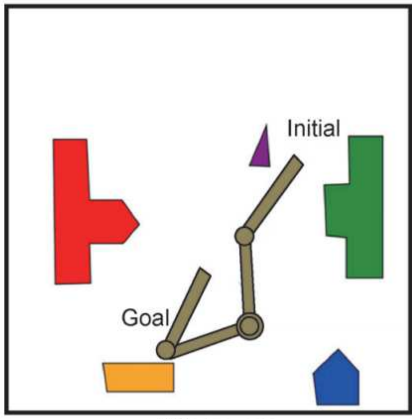
- **Proposal – 5%**
- **Update – 5%**
- **Poster – 5%**
- **Report and Code – 85%**
  - The bulk of your grade
  - Think of it as a full research paper
    - Abstract, Background, Motivation, Related Work from proposal
    - Algorithms explained, Graphs of experiments from Poster
    - Wrapped up in a coherent paper
  - Your code needs to work but the **VAST MAJORITY** of your grade is based on your paper so make sure you have AI contributions written up

From last time: Robotics is a **BIG** space

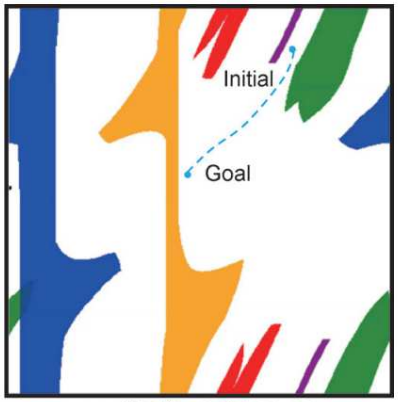


From last time: Spaces and Transformations

- **Task space**: the 3D workspace of the robot
  - E.g., the **pose** (x,y,z,roll,pitch,yaw) of the robot's hand or an object
- **Configuration space**: the *n*-dimensional space of joint angles + robot world position
  - Vector $q \in \mathbb{R}^n$
- **Forward kinematics**: maps *q* to outputs in task space (e.g. hand position)
- **Inverse kinematics**: maps task space poses to configuration space



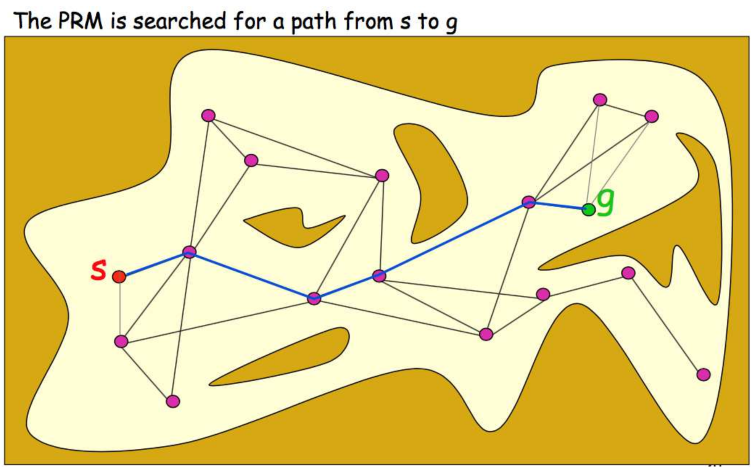
Workspace

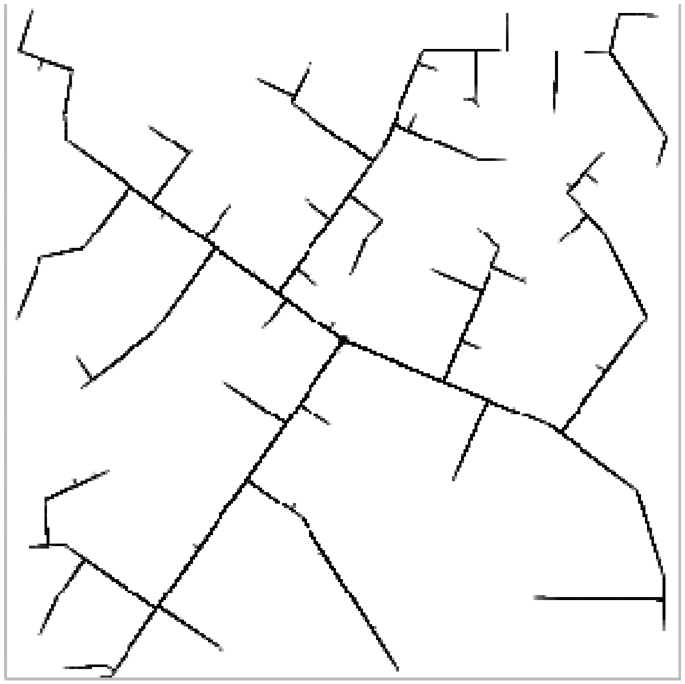


Configuration space

From last time: RRTs and PRMs

- **Single-query (RRT) vs. Multi-query (PRM)**
- **Probabilistically complete**
- **Computes feasible paths**
- Hundreds of papers introducing variants





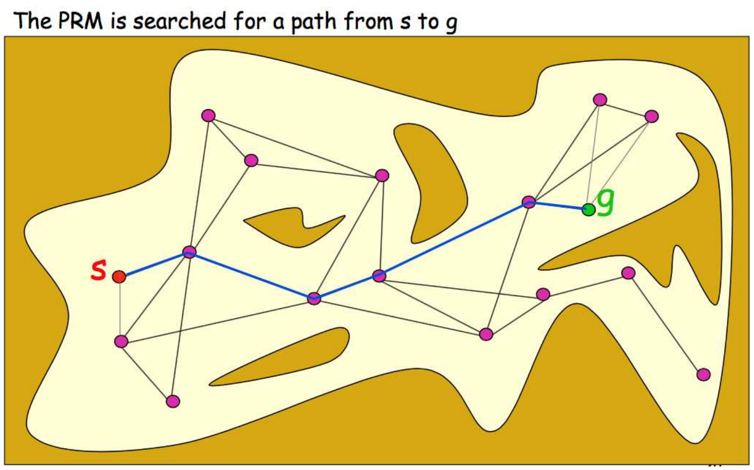
45 iterations

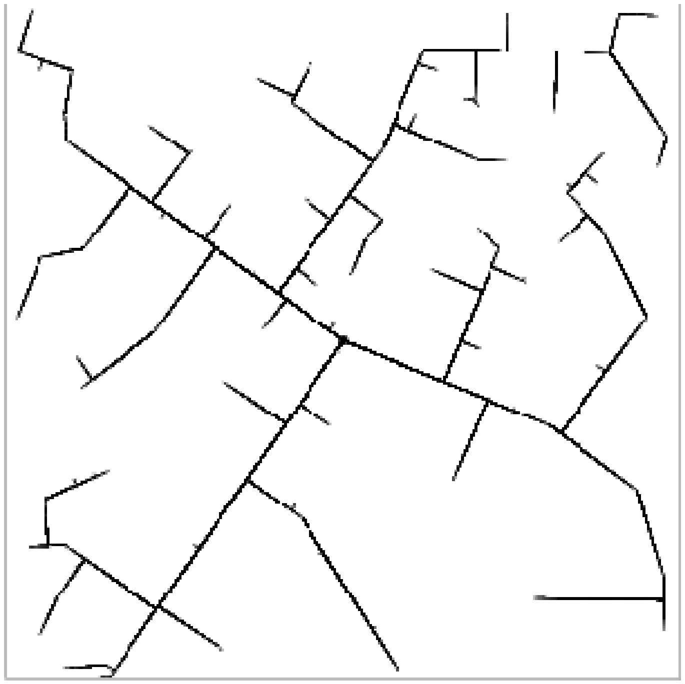
From last time: RRTs and PRMs

Neither is Optimal! (Unless infinite samples PRM)

Collision checking can be expensive!

- **Single-query (RRT) vs. Multi-query (PRM)**
- **Probabilistically complete**
- **Computes feasible paths**
- Hundreds of papers introducing variants



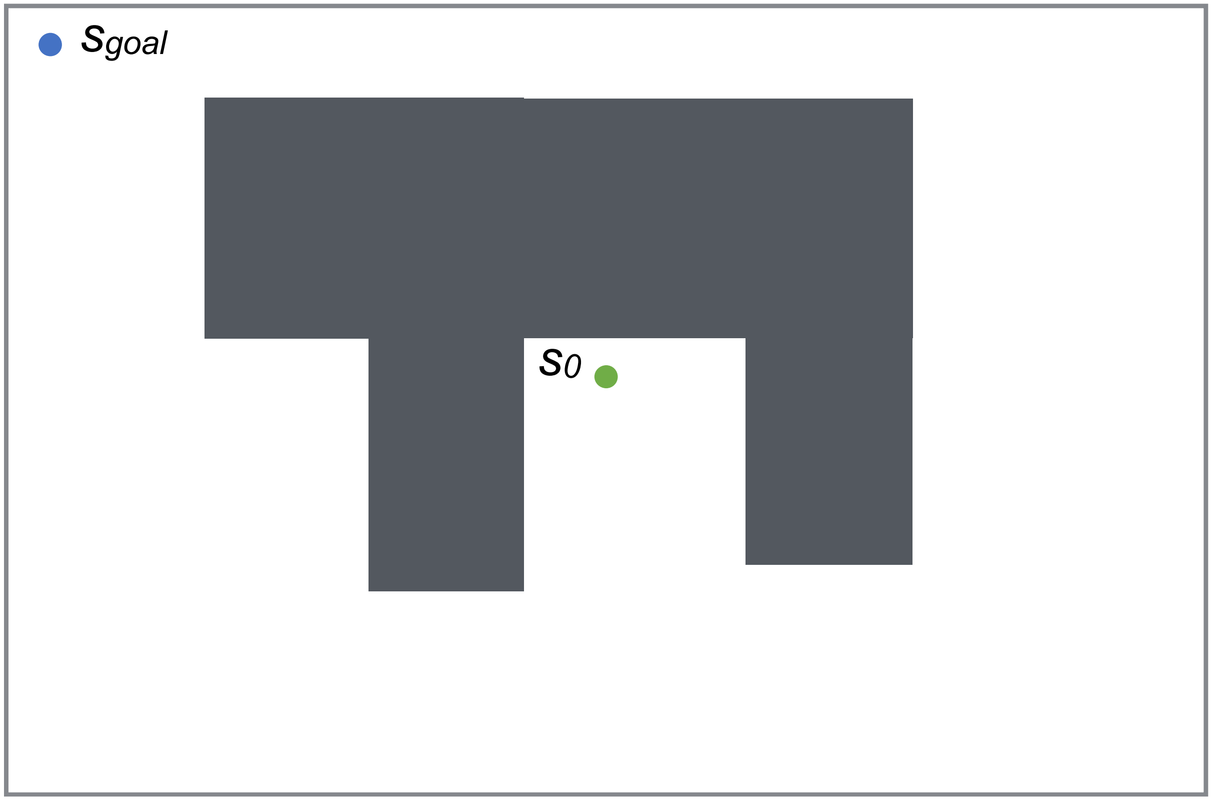


45 iterations

From last time: RRTs in action

Algorithm (**input: s_0, s_{goal}, initial state tree T**)

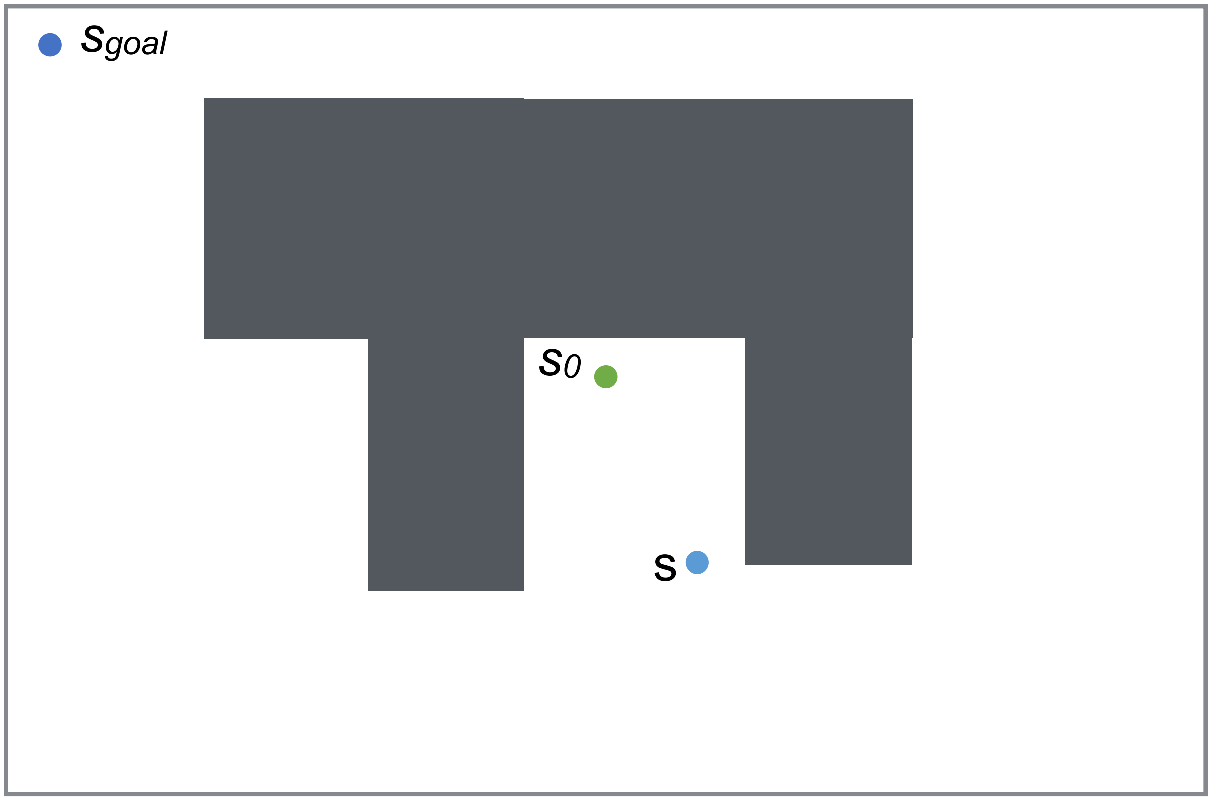
- Sample states $s \in S = R^{20}$ until s is collision-free
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



From last time: RRTs in action

Algorithm (input: s_0, s_{goal}, initial state tree T)

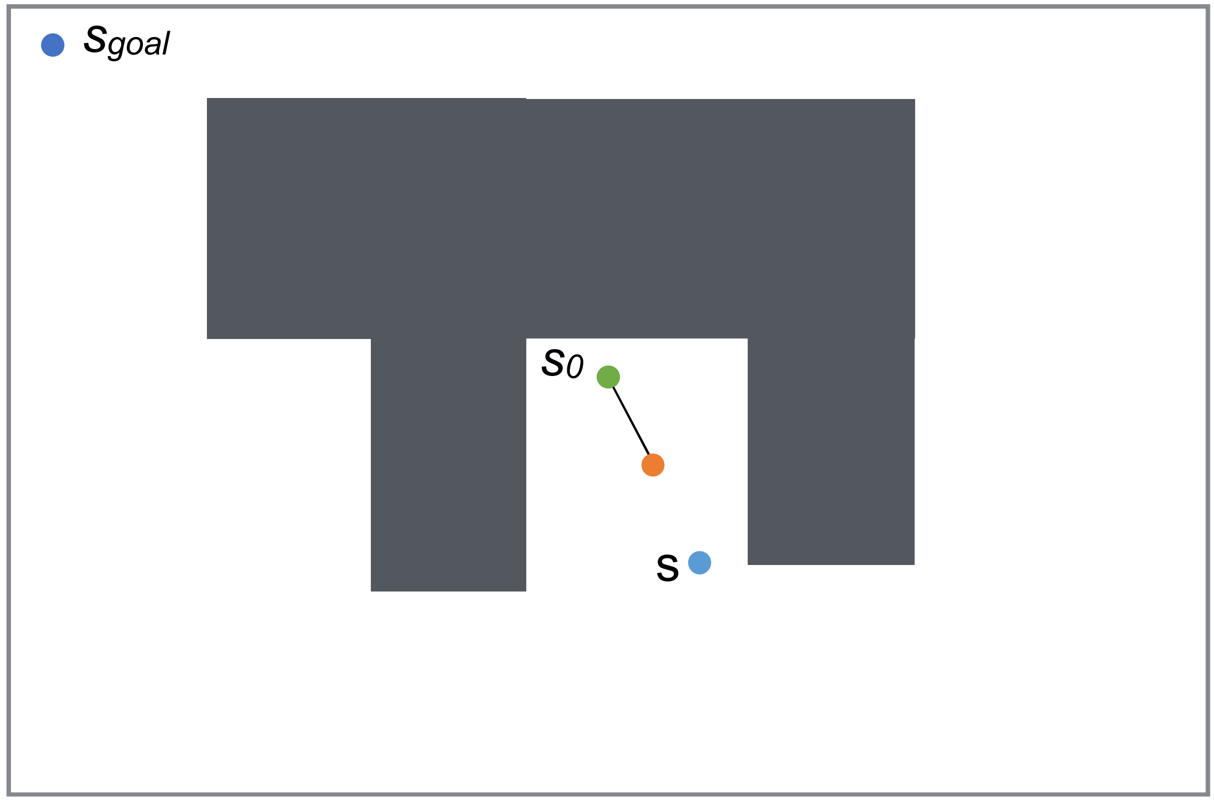
- **Sample states $s \in S = R^{20}$ until s is collision-free**
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



From last time: RRTs in action

Algorithm (input: s_0, s_{goal}, initial state tree T)

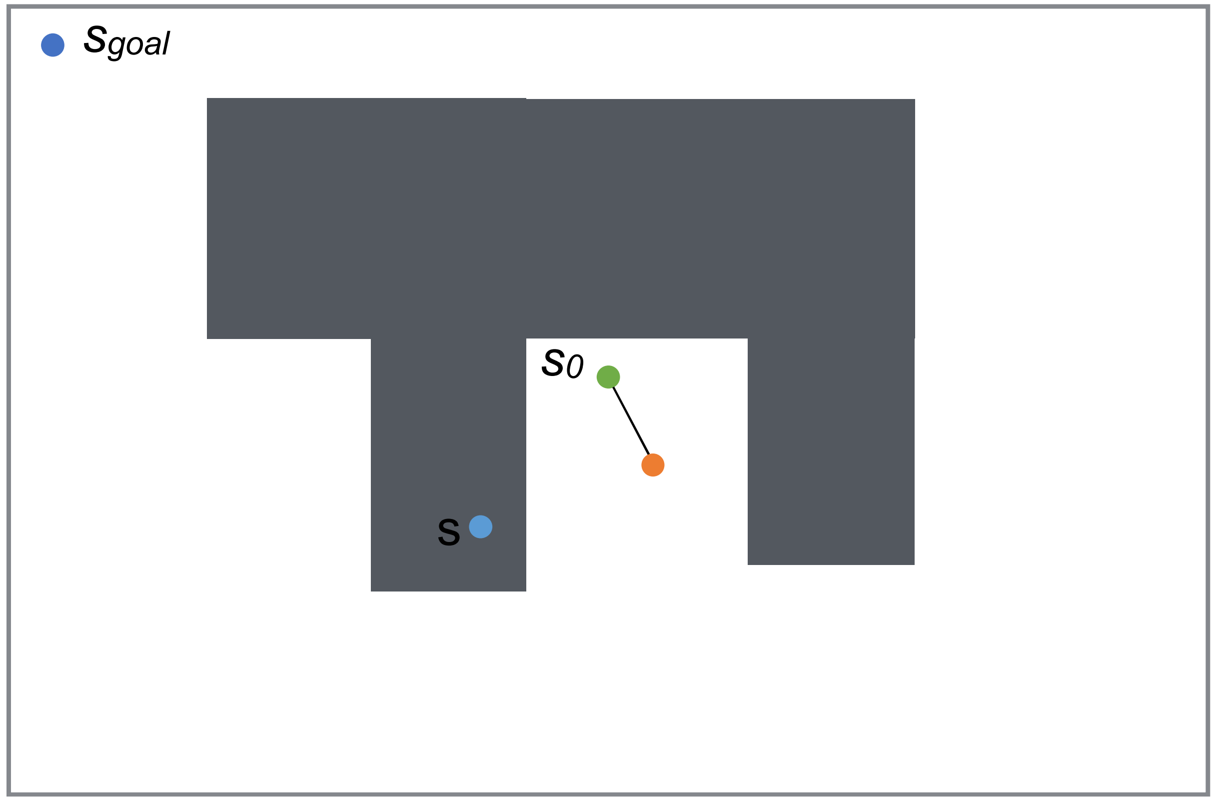
- Sample states $s \in S = R^{20}$ until s is collision-free
- **Find closest state $s_c \in T$**
- **Extend s_c toward s**
- **Add resulting state s' to T**
- Repeat until T contains a path from s_0 to s_{goal}



From last time: RRTs in action

Algorithm (input: s_0, s_{goal}, initial state tree T)

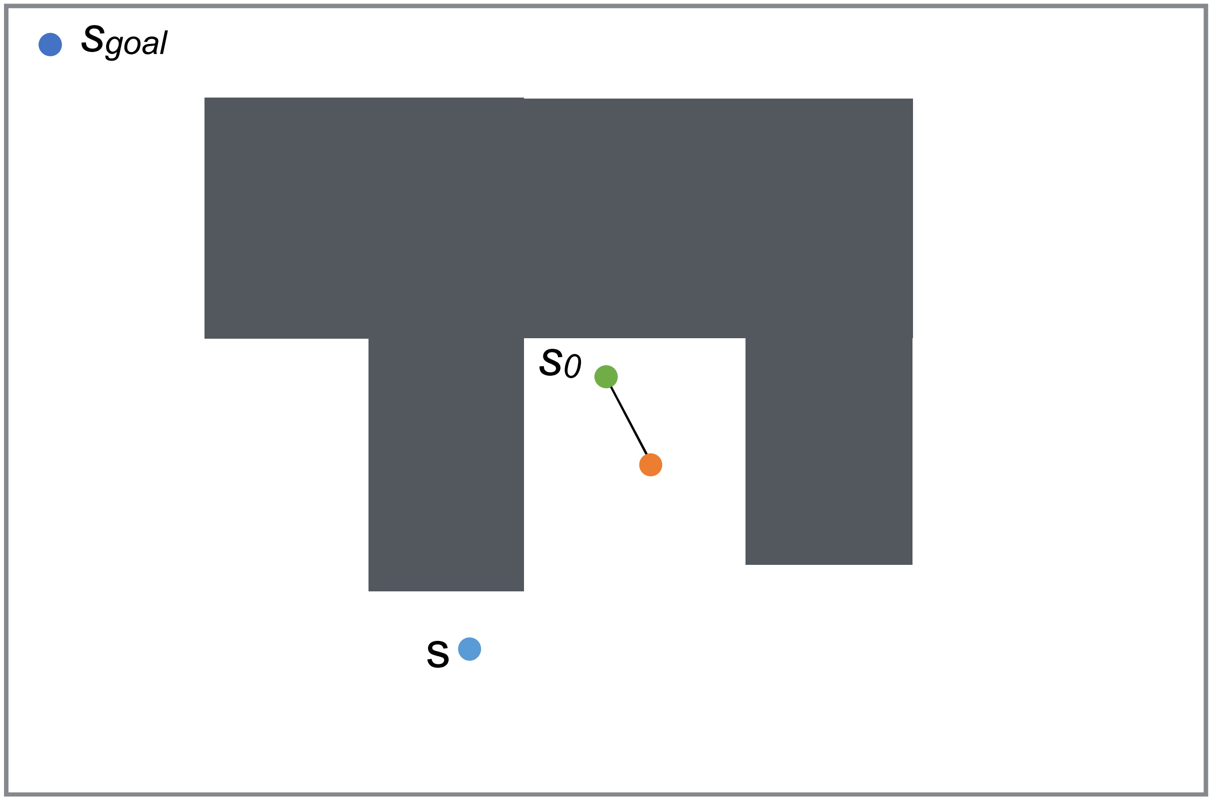
- **Sample states $s \in S = R^{20}$ until s is collision-free**
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



From last time: RRTs in action

Algorithm (input: s_0, s_{goal}, initial state tree T)

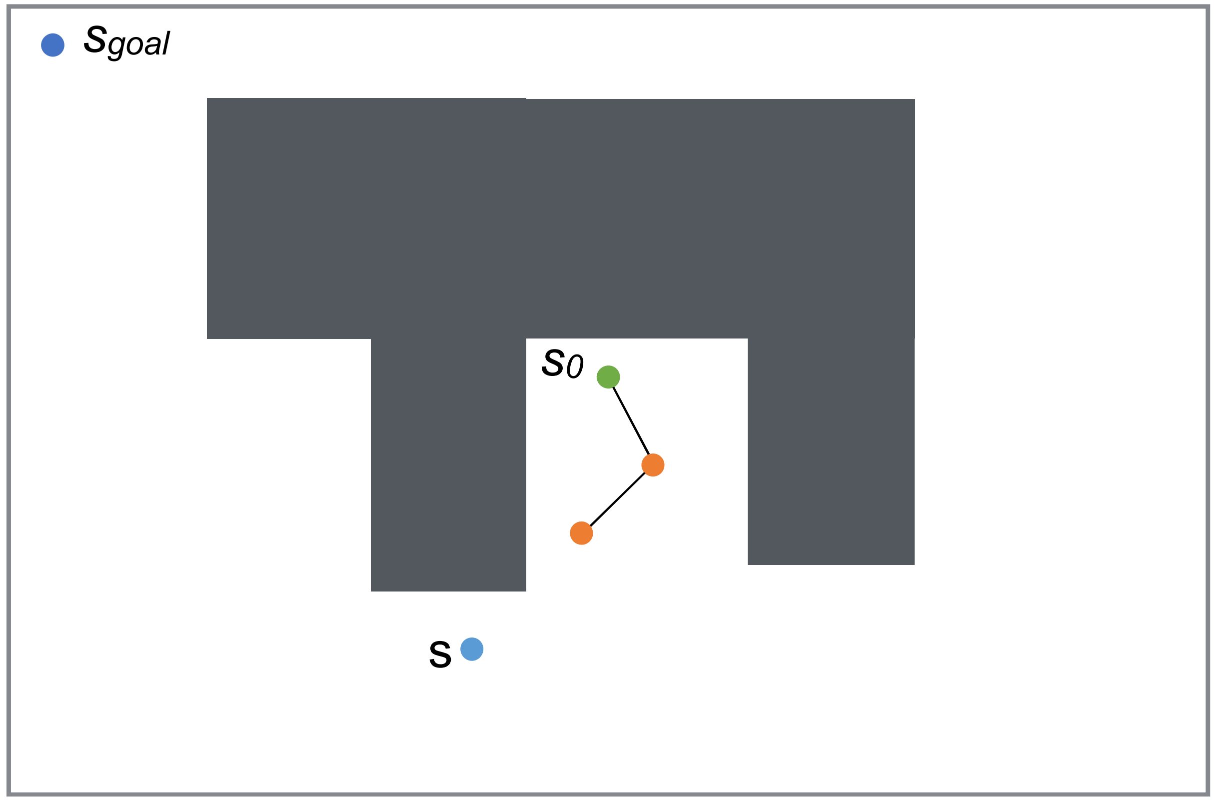
- **Sample states $s \in S = R^{20}$ until s is collision-free**
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



From last time: RRTs in action

Algorithm (input: s_0, s_{goal}, initial state tree T)

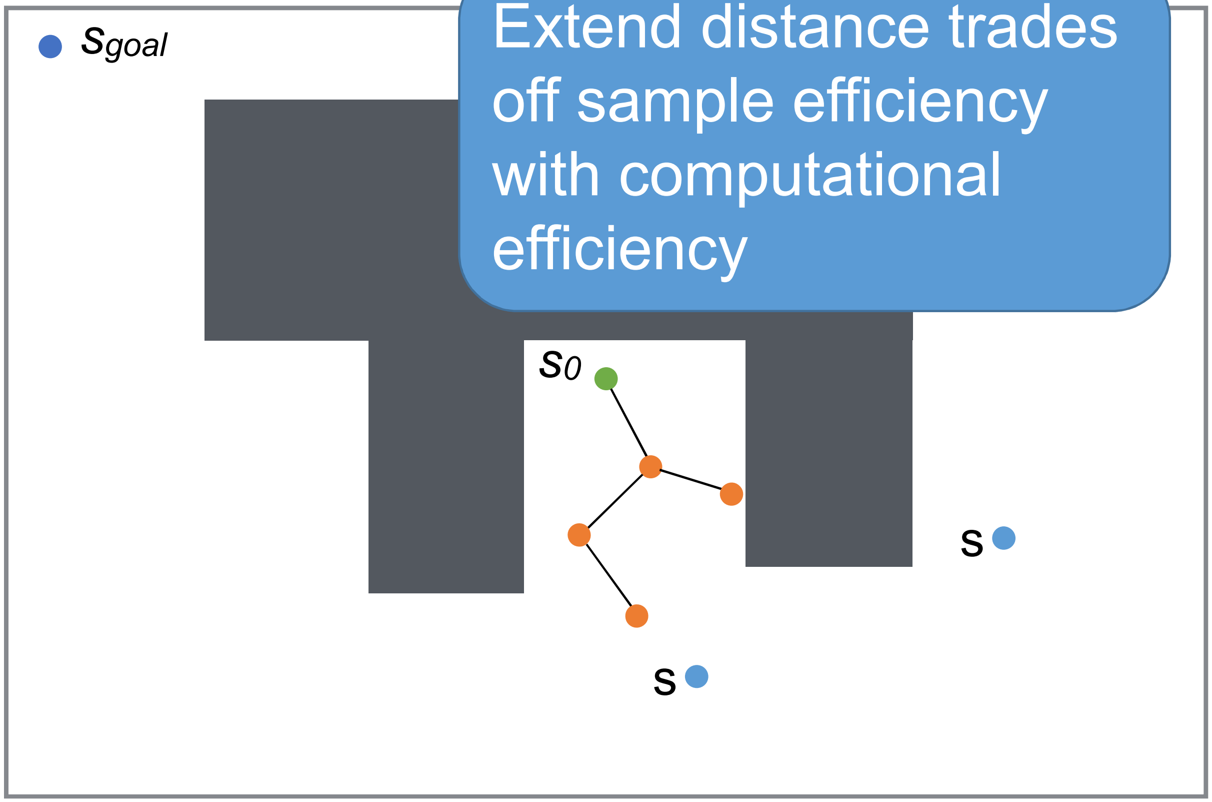
- Sample states $s \in S = R^{20}$ until s is collision-free
- **Find closest state $s_c \in T$**
- **Extend s_c toward s**
- **Add resulting state s' to T**
- Repeat until T contains a path from s_0 to s_{goal}



From last time: RRTs in action

Algorithm (input: s_0, s_{goal}, initial state tree T)

- Sample states $s \in S = R^{20}$ until s is collision-free
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- **Repeat until T contains a path from s_0 to s_{goal}**



Extend distance trades off sample efficiency with computational efficiency

What about optimality?

How do we modify the basic RRT algorithm to output optimal paths from s_0 to s_{goal}?

What about optimality?

How do we modify the basic RRT algorithm to output optimal paths from s_0 to s_{goal}?

- Change the sampling strategy?

What about optimality?

How do we modify the basic RRT algorithm to output optimal paths from s_0 to s_{goal}?

- Change the sampling strategy?
- Change the closest point logic?

What about optimality?

How do we modify the basic RRT algorithm to output optimal paths from s_0 to s_{goal}?

- Change the sampling strategy?
- Change the closest point logic?
- Incrementally “rewire” the tree?

RRT variant called RRT* does this!

RRT* Algorithm

RRT* (input: s_0, s_{goal}, initial state tree T)

- Sample states $s \in S = R^{15}$ until s is collision-free (often goal directed)
- Find closest state $s_c \in T$
- Extend s_c toward s resulting in state s'
- **STUFF GOES HERE**
- Repeat until **maximum iterations reached and** T contains a path from s_0 to s_{goal}

RRT* Algorithm

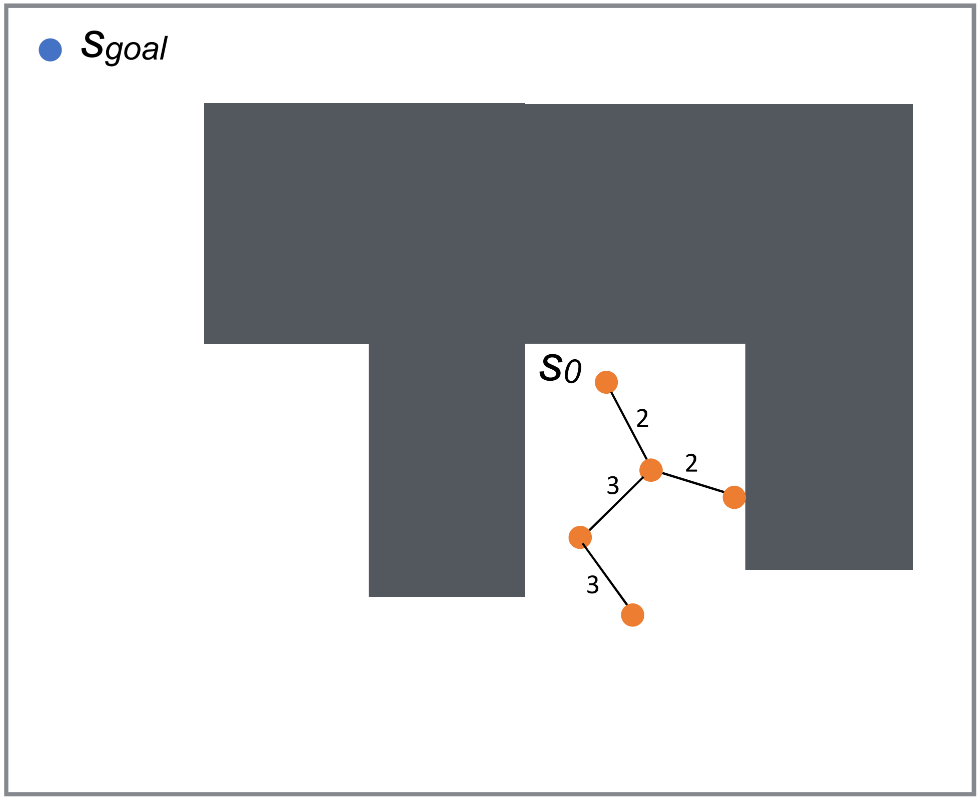
RRT* (input: s_0, s_{goal}, initial state tree T)

- Sample states $s \in S = R^{15}$ until s is collision-free (often goal directed)
- Find closest state $s_c \in T$
- Extend s_c toward s resulting in state s'
- **Find all $s_{near} \subseteq T$ within a distance d to s'**
- **Find $s_{min} \in s_{near}$, that has the lowest *path cost* to $s_0 \rightarrow s_{min} \rightarrow s'$**
- **Add edge $s_{min} \rightarrow s'$ to T**
- **Check path cost through s' to all states in $s \in s_{near}$, if any are lower than existing path cost to s, then "rewire" tree to include edge $s' \rightarrow s$**
- Repeat until **maximum iterations reached and** T contains a path from s_0 to s_{goal}

RRT* by example

RRT* (input: s_0, s_{goal}, initial state tree T)

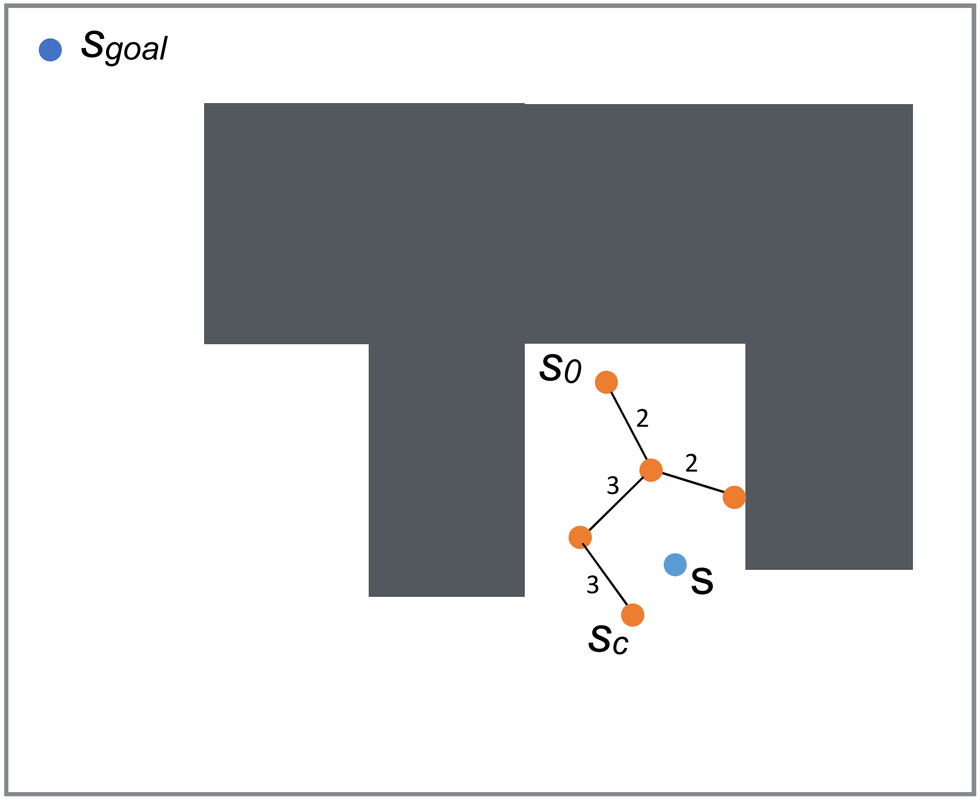
- Sample states $s \in S = R^{15}$ until s is collision-free (often goal directed)
- Find closest state $s_c \in T$
- Extend s_c toward s resulting in state s'
- Find all $s_{near} \subseteq T$ within a distance d to s'
- Find $s_{min} \in s_{near}$, that has the lowest *path cost* to s_0 -> s_{min} -> s'
- Add edge s_{min} -> s' to T
- Check path cost through s' to all states in $s \in s_{near}$, if any are lower than existing path cost to s, then "rewire" tree to include edge s' -> s
- Repeat until maximum iterations reached and T contains a path from s_0 to s_{goal}



RRT* by example

RRT* (input: s_0, s_{goal}, initial state tree T)

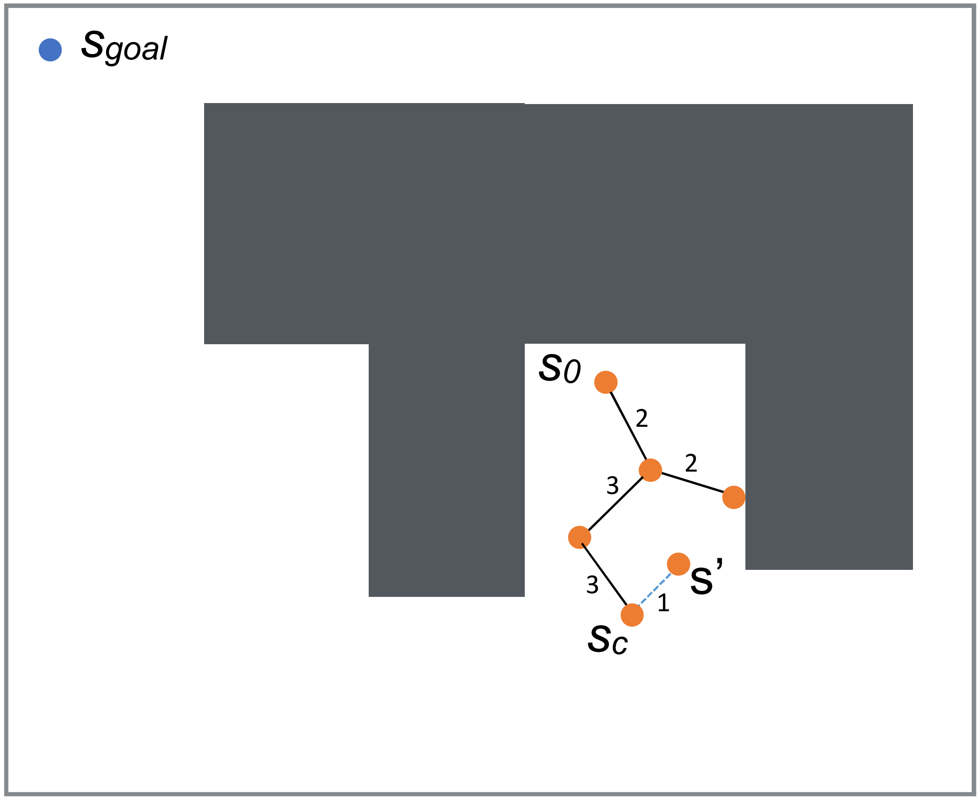
- **Sample states $s \in S = R^{15}$ until s is collision-free (often goal directed)**
- **Find closest state $s_c \in T$**
- Extend s_c toward s resulting in state s'
- Find all $s_{near} \subseteq T$ within a distance d to s'
- Find $s_{min} \in s_{near}$, that has the lowest *path cost* to s_0 -> s_{min} -> s'
- Add edge s_{min} -> s' to T
- Check path cost through s' to all states in $s \in s_{near}$, if any are lower than existing path cost to s, then "rewire" tree to include edge s' -> s
- Repeat until maximum iterations reached and T contains a path from s_0 to s_{goal}



RRT* by example

RRT* (input: s_0, s_{goal}, initial state tree T)

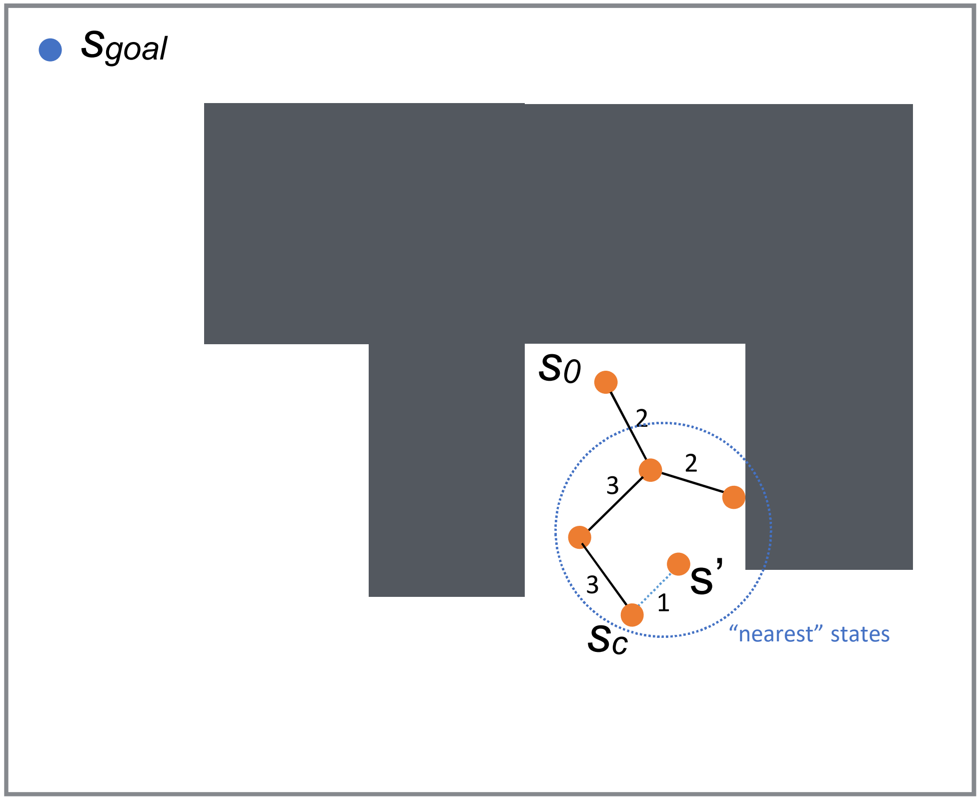
- Sample states $s \in S = R^{15}$ until s is collision-free (often goal directed)
- Find closest state $s_c \in T$
- **Extend s_c toward s resulting in state s'**
- Find all $s_{near} \subseteq T$ within a distance d to s'
- Find $s_{min} \in s_{near}$, that has the lowest *path cost* to s_0 -> s_{min} -> s'
- Add edge s_{min} -> s' to T
- Check path cost through s' to all states in $s \in s_{near}$, if any are lower than existing path cost to s, then "rewire" tree to include edge s' -> s
- Repeat until maximum iterations reached and T contains a path from s_0 to s_{goal}



RRT* by example

RRT* (input: s_0, s_{goal}, initial state tree T)

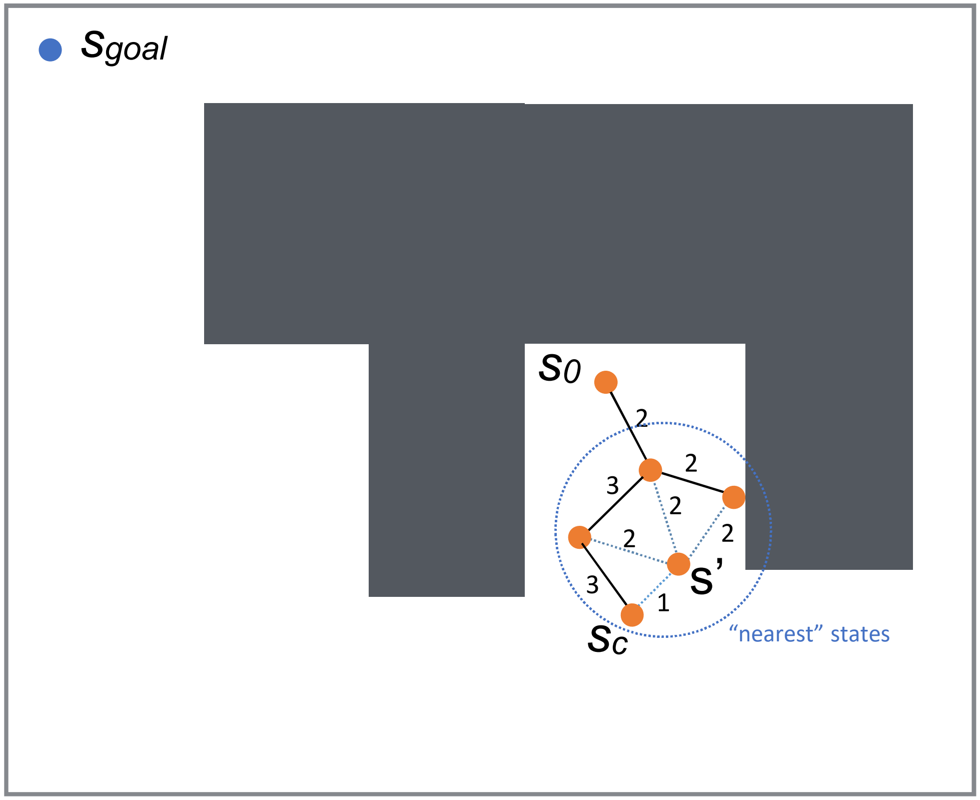
- Sample states $s \in S = R^{15}$ until s is collision-free (often goal directed)
- Find closest state $s_c \in T$
- Extend s_c toward s resulting in state s'
- **Find all $s_{near} \subseteq T$ within a distance d to s'**
- Find $s_{min} \in s_{near}$, that has the lowest *path cost* to s_0 -> s_{min} -> s'
- Add edge s_{min} -> s' to T
- Check path cost through s' to all states in $s \in s_{near}$, if any are lower than existing path cost to s, then "rewire" tree to include edge s' -> s
- Repeat until maximum iterations reached and T contains a path from s_0 to s_{goal}



RRT* by example

RRT* (input: s_0, s_{goal}, initial state tree T)

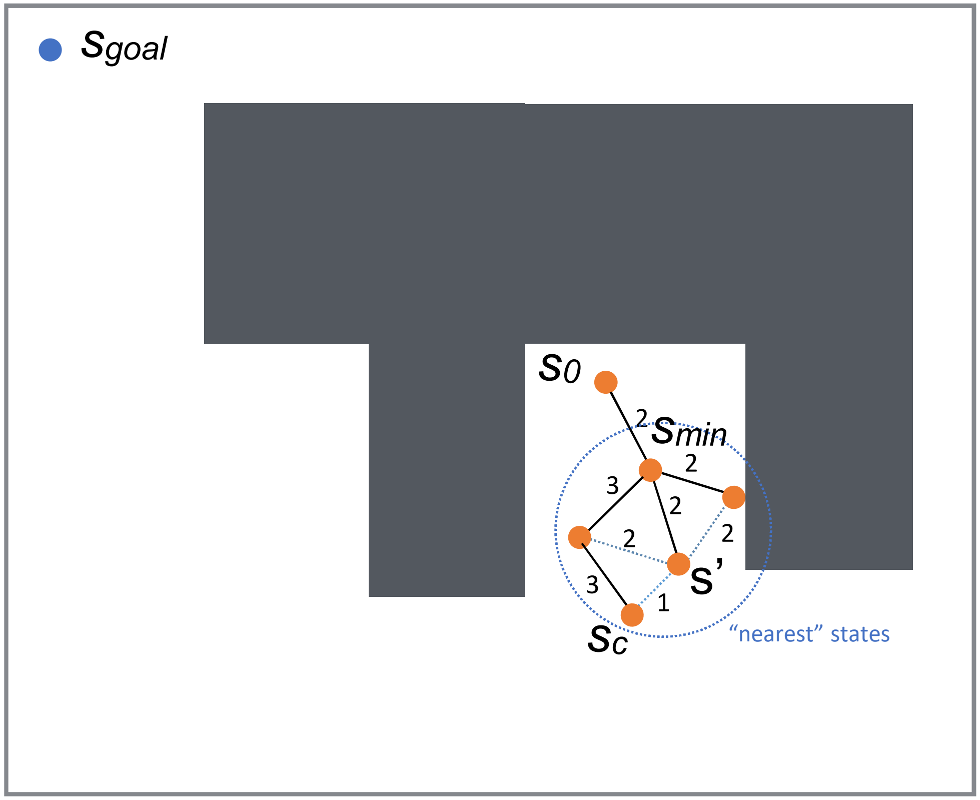
- Sample states $s \in S = R^{15}$ until s is collision-free (often goal directed)
- Find closest state $s_c \in T$
- Extend s_c toward s resulting in state s'
- Find all $s_{near} \subseteq T$ within a distance d to s'
- **Find $s_{min} \in s_{near}$, that has the lowest *path cost* to s_0 -> s_{min} -> s'**
- **Add edge s_{min} -> s' to T**
- Check path cost through s' to all states in $s \in s_{near}$, if any are lower than existing path cost to s, then "rewire" tree to include edge s' -> s
- Repeat until maximum iterations reached and T contains a path from s_0 to s_{goal}



RRT* by example

RRT* (input: s_0, s_{goal}, initial state tree T)

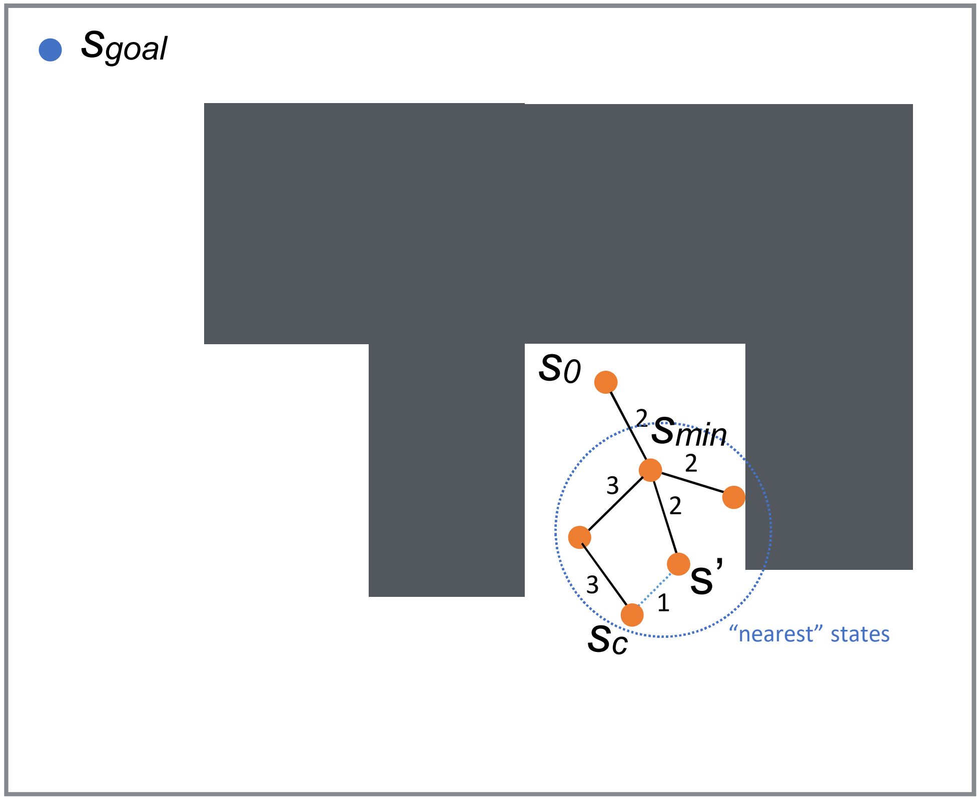
- Sample states $s \in S = R^{15}$ until s is collision-free (often goal directed)
- Find closest state $s_c \in T$
- Extend s_c toward s resulting in state s'
- Find all $s_{near} \subseteq T$ within a distance d to s'
- **Find $s_{min} \in s_{near}$, that has the lowest *path cost* to s_0 -> s_{min} -> s'**
- **Add edge s_{min} -> s' to T**
- Check path cost through s' to all states in $s \in s_{near}$, if any are lower than existing path cost to s, then "rewire" tree to include edge s' -> s
- Repeat until maximum iterations reached and T contains a path from s_0 to s_{goal}



RRT* by example

RRT* (input: s_0, s_{goal}, initial state tree T)

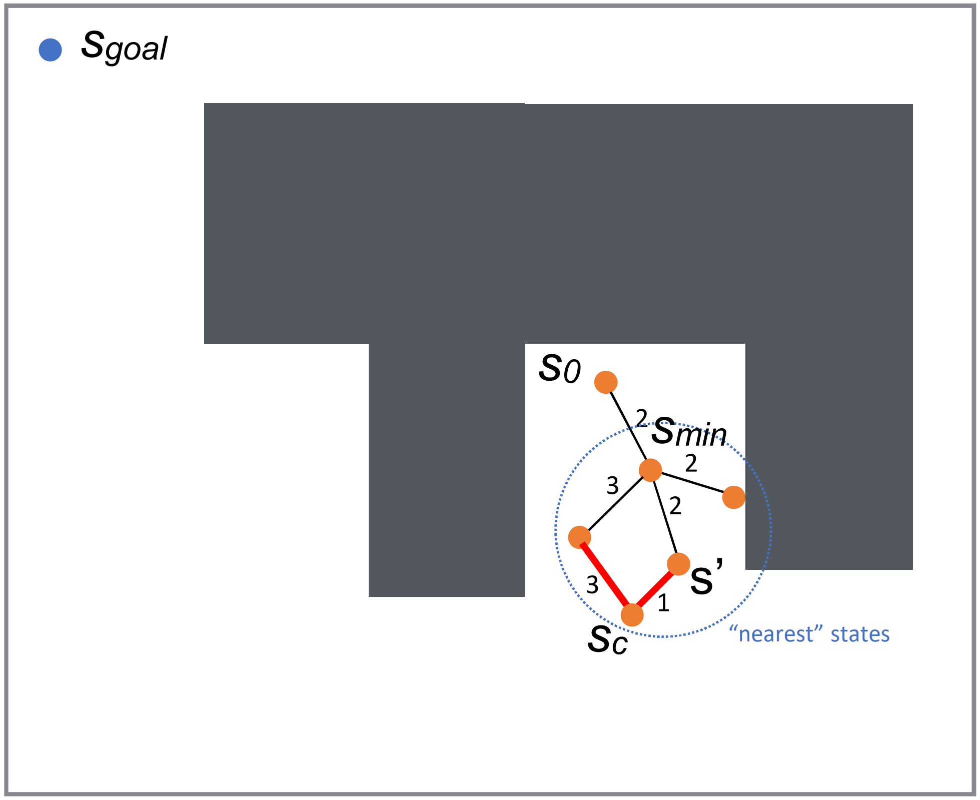
- Sample states $s \in S = R^{15}$ until s is collision-free (often goal directed)
- Find closest state $s_c \in T$
- Extend s_c toward s resulting in state s'
- Find all $s_{near} \subseteq T$ within a distance d to s'
- **Find $s_{min} \in s_{near}$, that has the lowest *path cost* to s_0 -> s_{min} -> s'**
- **Add edge s_{min} -> s' to T**
- Check path cost through s' to all states in $s \in s_{near}$, if any are lower than existing path cost to s, then "rewire" tree to include edge s' -> s
- Repeat until maximum iterations reached and T contains a path from s_0 to s_{goal}



RRT* by example

RRT* (input: s_0, s_{goal}, initial state tree T)

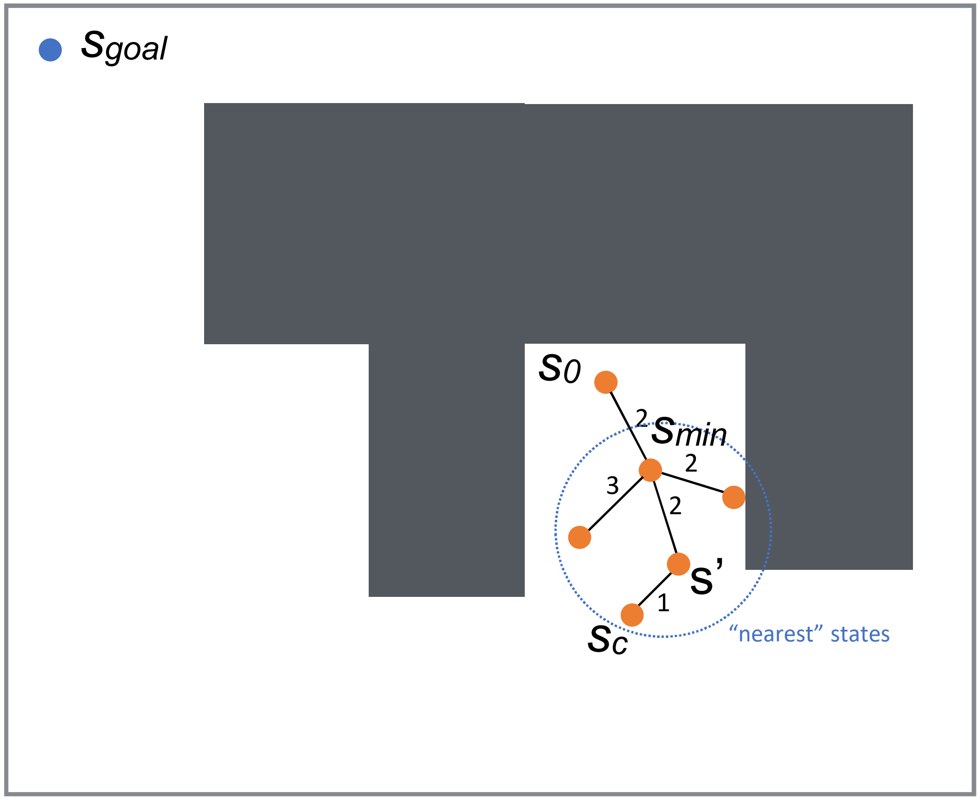
- Sample states $s \in S = R^{15}$ until s is collision-free (often goal directed)
- Find closest state $s_c \in T$
- Extend s_c toward s resulting in state s'
- Find all $s_{near} \subseteq T$ within a distance d to s'
- Find $s_{min} \in s_{near}$, that has the lowest *path cost* to s_0 -> s_{min} -> s'
- Add edge s_{min} -> s' to T
- **Check path cost through s' to all states in $s \in s_{near}$, if any are lower than existing path cost to s, then "rewire" tree to include edge s' -> s**
- Repeat until maximum iterations reached and T contains a path from s_0 to s_{goal}



RRT* by example

RRT* (input: s_0, s_{goal}, initial state tree T)

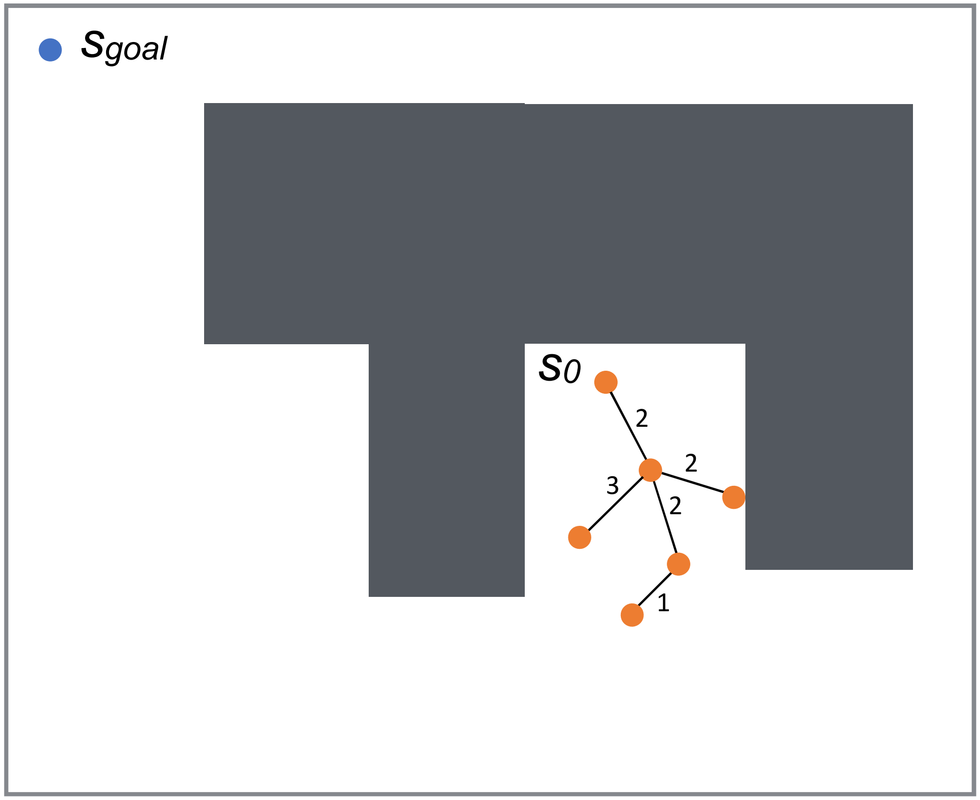
- Sample states $s \in S = R^{15}$ until s is collision-free (often goal directed)
- Find closest state $s_c \in T$
- Extend s_c toward s resulting in state s'
- Find all $s_{near} \subseteq T$ within a distance d to s'
- Find $s_{min} \in s_{near}$, that has the lowest *path cost* to s_0 -> s_{min} -> s'
- Add edge s_{min} -> s' to T
- **Check path cost through s' to all states in $s \in s_{near}$, if any are lower than existing path cost to s, then "rewire" tree to include edge s' -> s**
- Repeat until maximum iterations reached and T contains a path from s_0 to s_{goal}



RRT* by example

RRT* (input: s_0, s_{goal}, initial state tree T)

- Sample states $s \in S = R^{15}$ until s is collision-free (often goal directed)
- Find closest state $s_c \in T$
- Extend s_c toward s resulting in state s'
- Find all $s_{near} \subseteq T$ within a distance d to s'
- Find $s_{min} \in s_{near}$, that has the lowest *path cost* to s_0 -> s_{min} -> s'
- Add edge s_{min} -> s' to T
- Check path cost through s' to all states in $s \in s_{near}$, if any are lower than existing path cost to s, then "rewire" tree to include edge s' -> s
- **Repeat until maximum iterations reached and T contains a path from s_0 to s_{goal}**

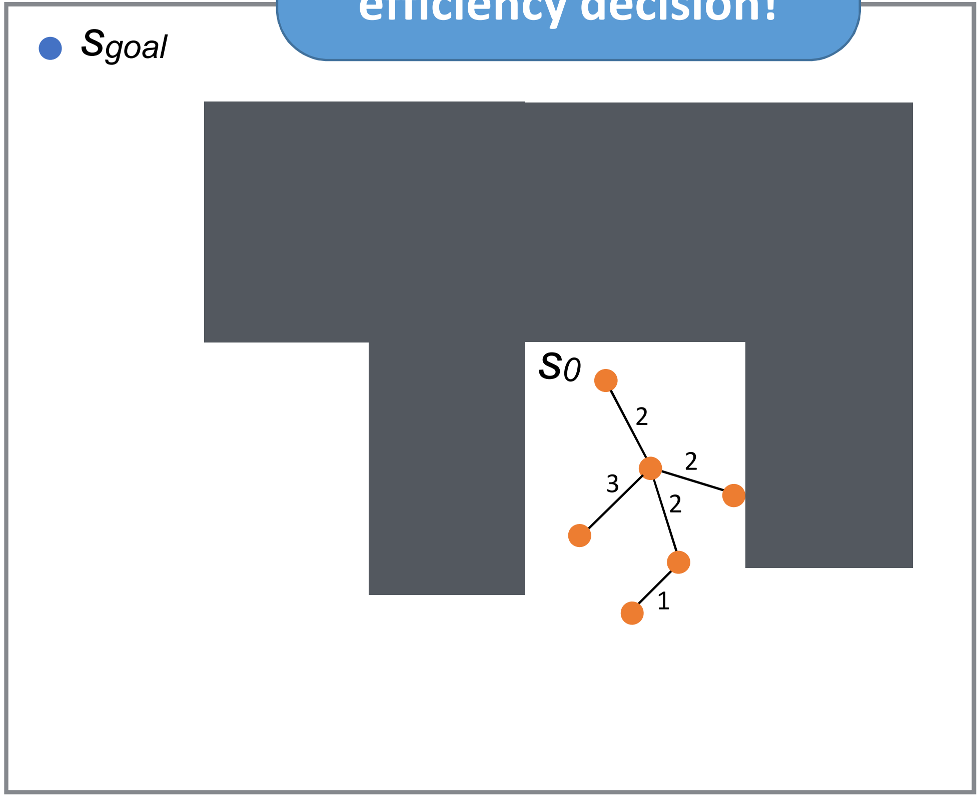


RRT* by example

Nearest radius size is another sample vs. computational efficiency decision!

RRT* (input: s_0, s_{goal}, initial state tree T)

- Sample states $s \in S = R^{15}$ until s is collision-free (often goal directed)
- Find closest state $s_c \in T$
- Extend s_c toward s resulting in state s'
- Find all $s_{near} \subseteq T$ within a distance d to s'
- Find $s_{min} \in s_{near}$, that has the lowest *path cost* to s_0 -> s_{min} -> s'
- Add edge s_{min} -> s' to T
- Check path cost through s' to all states in $s \in s_{near}$, if any are lower than existing path cost to s, then "rewire" tree to include edge s' -> s
- **Repeat until maximum iterations reached and T contains a path from s_0 to s_{goal}**



RRT* Algorithm

[Source: Karaman & Fazzoli]

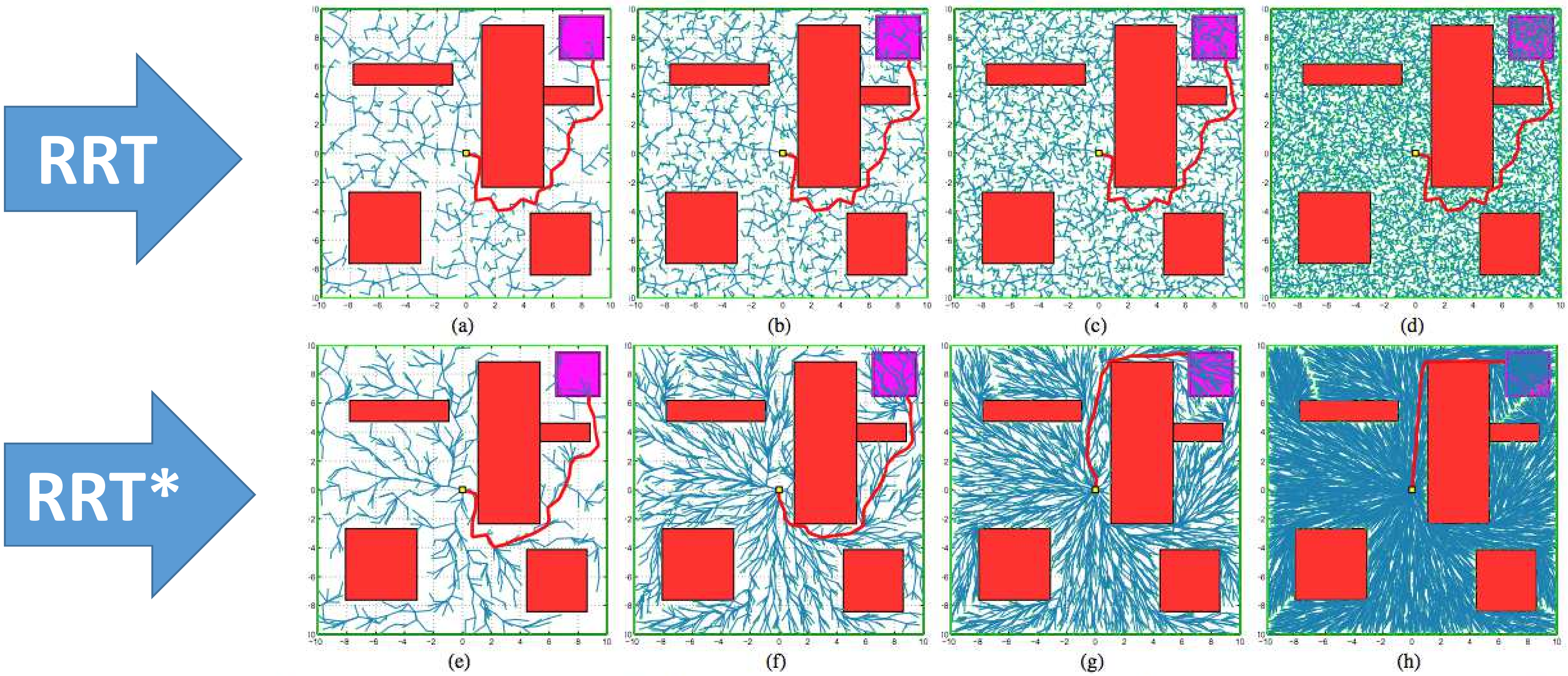


Fig. 1. A Comparison of the RRT* and RRT algorithms on a simulation example. The tree maintained by the RRT algorithm is shown in (a)-(d) in different stages, whereas that maintained by the RRT* algorithm is shown in (e)-(h). The tree snapshots (a), (e) are at 1000 iterations , (b), (f) at 2500 iterations, (c), (g) at 5000 iterations, and (d), (h) at 15,000 iterations. The goal regions are shown in magenta. The best paths that reach the target are highlighted with red.

Properties of RRT*

- **Complete**? Yes (still)!

Properties of RRT*

- **Complete**? Yes (still)!
- **Optimal**? Yes! But can still take a long time to converge to optimum!

Properties of RRT*

- **Complete**? Yes (still)!
- **Optimal**? Yes! But can still take a long time to converge to optimum!
- Like RRT, **dozens of variants** of RRT* (e.g., bias samples to best path area)

Properties of RRT*

- **Complete**? Yes (still)!
- **Optimal**? Yes! But can still take a long time to converge to optimum!
- Like RRT, **dozens of variants** of RRT* (e.g., bias samples to best path area)
- Is there an analogous **PRM*** algorithm?
  - PRMs are already asymptotically optimal as #nodes -> infinity
  - There is a variant called PRM* that works just like PRM, but reduces the “nearest points” ball as the number of samples grows

Properties of RRT*

- **Complete**? Yes (still)!
- **Optimal**? Yes! But can still take a long time to converge to optimum!
- Like RRT, **dozens of variants** of RRT* (e.g., bias samples to best path area)
- Is there an analogous **PRM*** algorithm?
  - PRMs are already asymptotically optimal as #nodes -> infinity
  - There is a variant called PRM* that works just like PRM, but reduces the “nearest points” ball as the number of samples grows
- Can we combine PRMs (or graph planning generally) with RRT*?
  - There is an algorithm call **Fast Marching Trees (FMT*)** which tries to do the “best of both world”

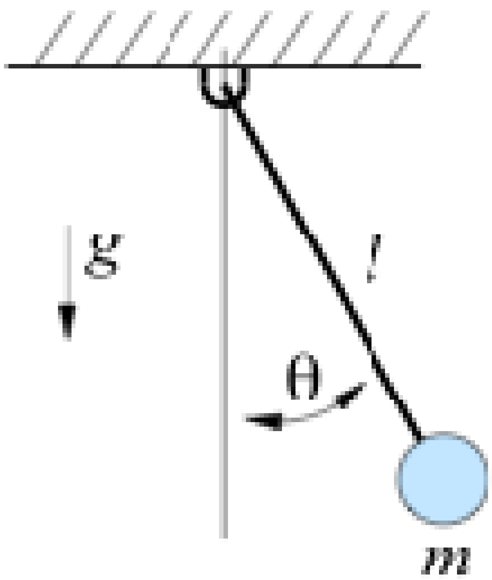
Ok so why can’t robots use these awesome kinematic planning algorithms all the time and be better at life?!?

Ok so why can’t robots use these awesome kinematic planning algorithms all the time and be better at life?!?

Dynamics (aka Physics)

Ok so why can’t robots use these awesome kinematic planning algorithms all the time and be better at life?!?

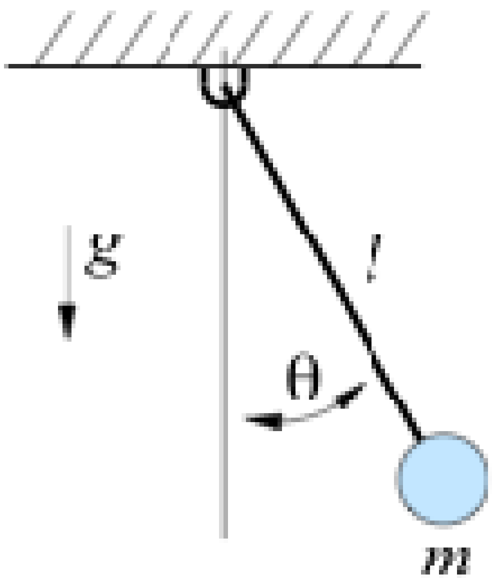
The Simplest “Robot”



- States: $s = \{\theta, \dot{\theta}\}$ aka angle and angular velocity
- Actions: $a = \tau$ aka torque at joint
- Transitions: $s' = f(s, a)$ aka physics

Ok so why can't robots use these awesome kinematic planning algorithms all the time and be better at life?!?

The Simplest "Robot"

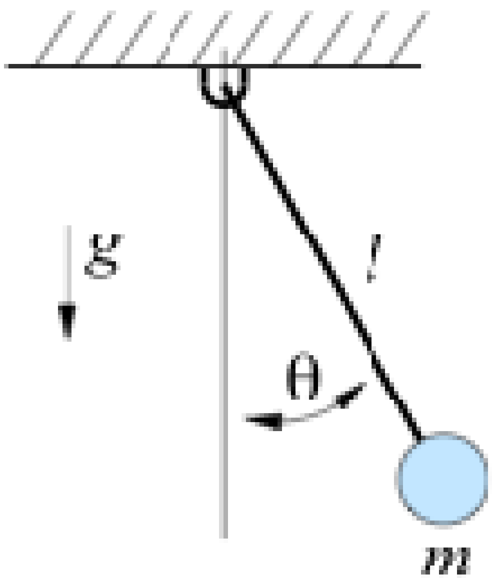


Q: Why do we need to track position and velocity?

- States: $s = \{\theta, \dot{\theta}\}$ aka angle and angular velocity
- Actions: $a = \tau$ aka torque at joint
- Transitions: $s' = f(s, a)$ aka physics

Ok so why can't robots use these awesome kinematic planning algorithms all the time and be better at life?!?

The Simplest "Robot"



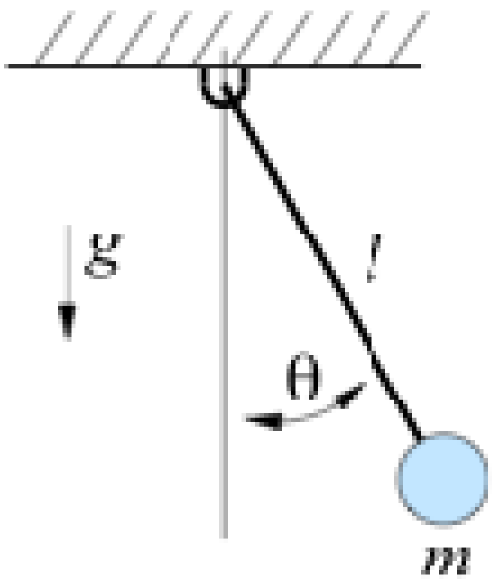
- States: $s = \{\theta, \dot{\theta}\}$ aka angle and angular velocity
- Actions: $a = \tau$ aka torque at joint
- Transitions: $s' = f(s, a)$ aka physics

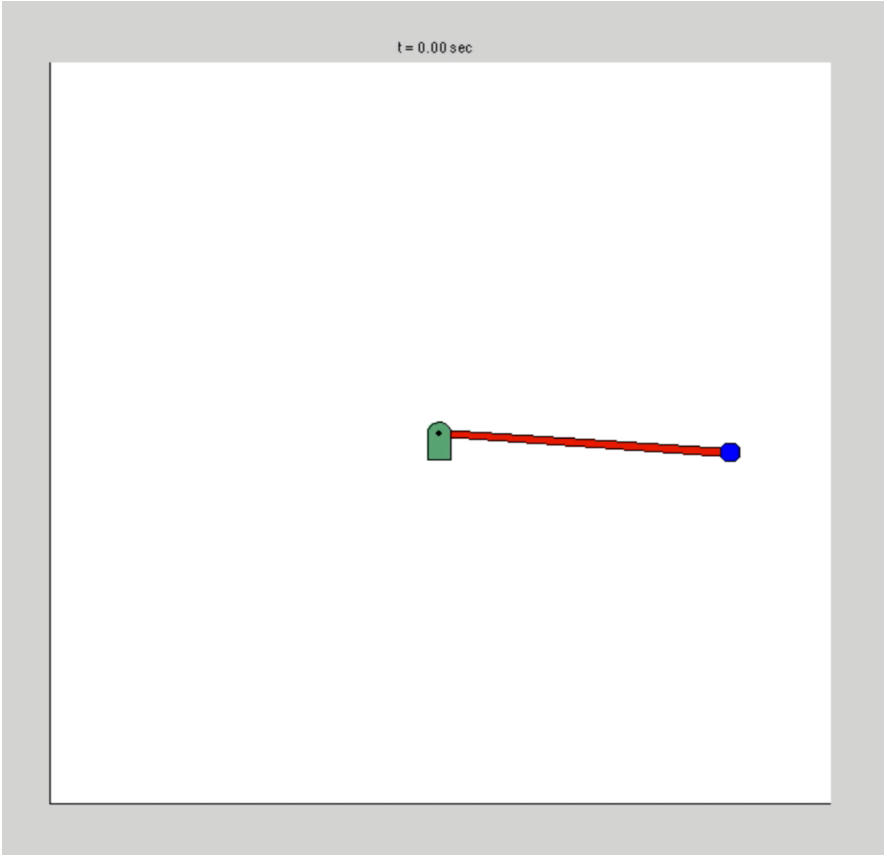
Euler Integrator

$$\boldsymbol{m} = \boldsymbol{l} = \mathbf{1}$$
$$\dot{\boldsymbol{s}} = \{\dot{\theta}, \tau + \boldsymbol{g} \sin\theta - \boldsymbol{\alpha}\dot{\boldsymbol{\theta}}\}$$
$$s' = s + \boldsymbol{dt} * \dot{\boldsymbol{s}}$$

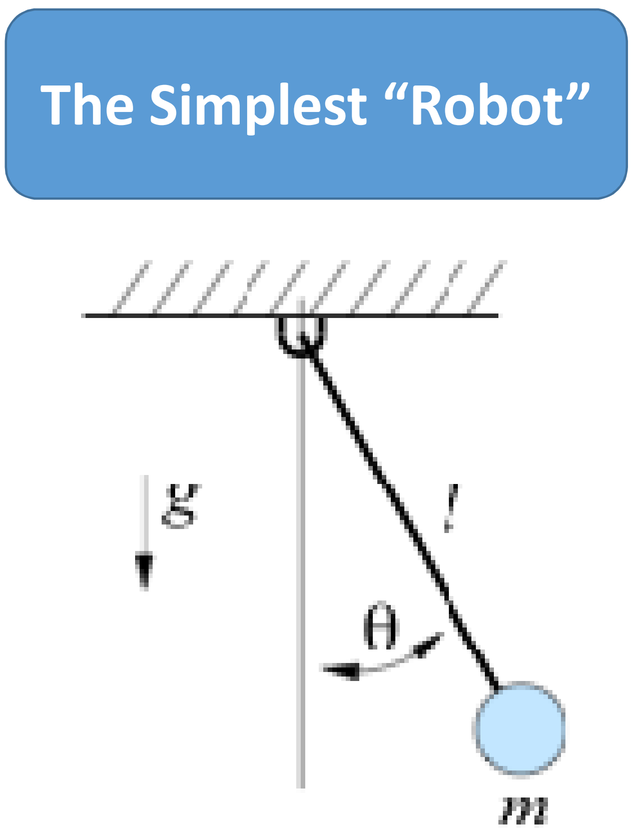
Ok so why can't robots use these awesome kinematic planning algorithms all the time and be better at life?!?

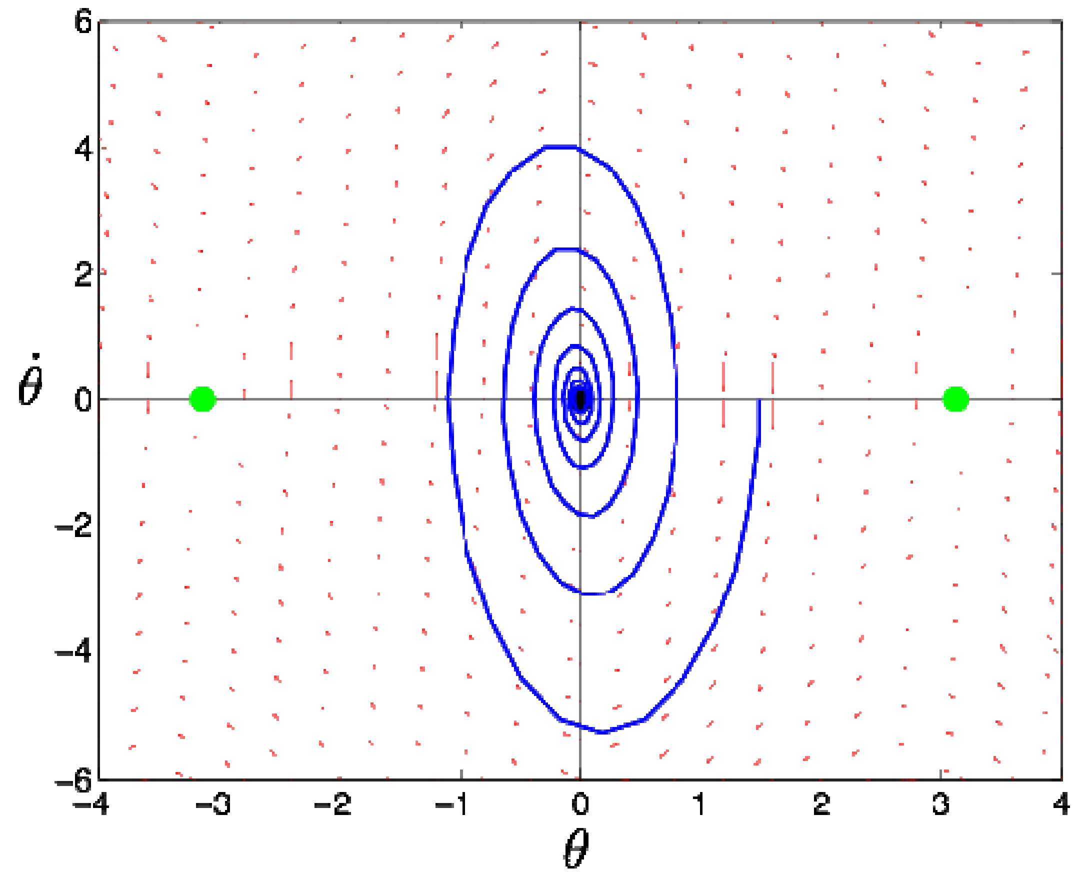
The Simplest "Robot"





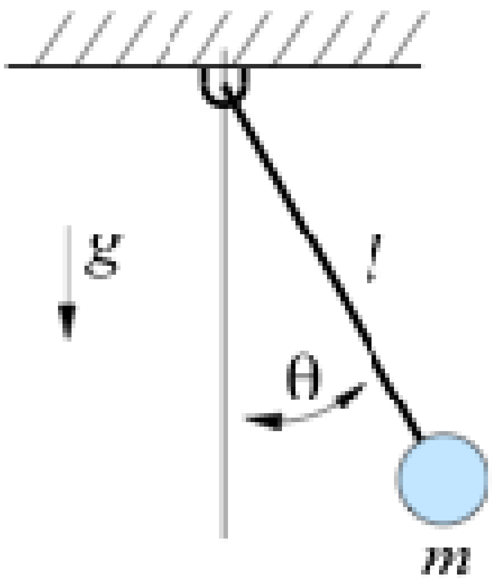
Ok so why can't robots use these awesome kinematic planning algorithms all the time and be better at life?!?

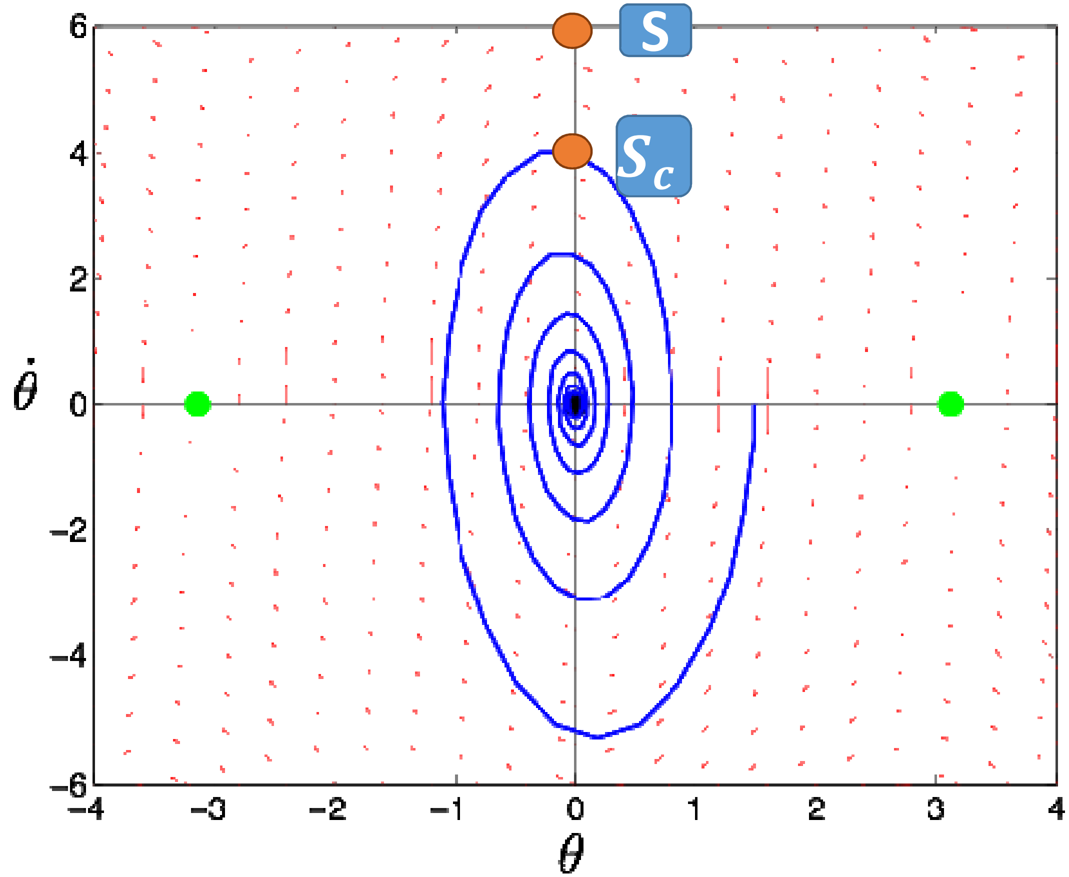




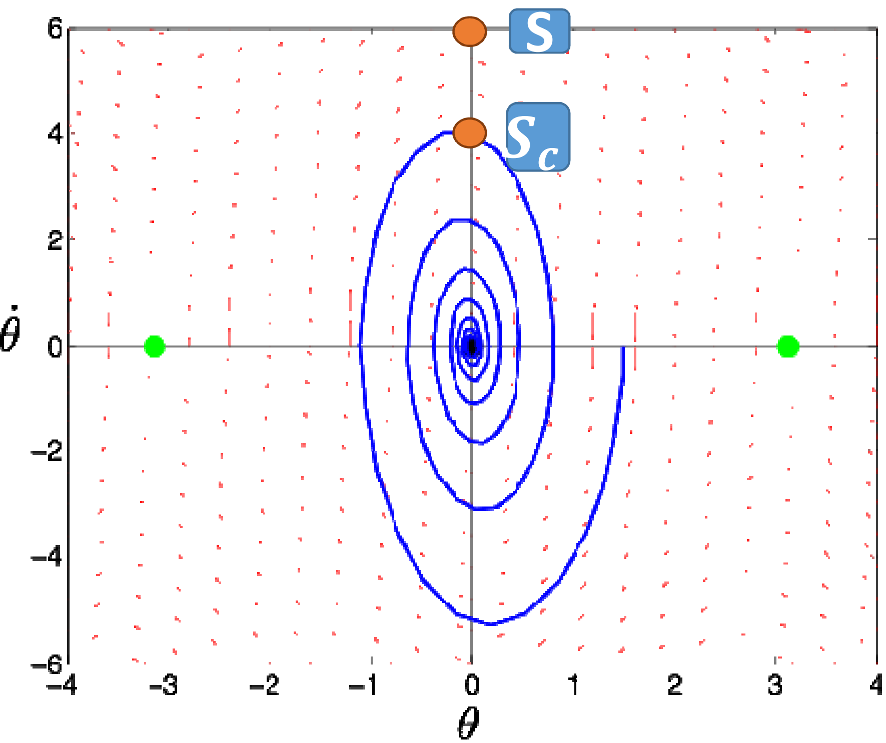
Ok so why can't robots use these awesome kinematic planning algorithms all the time and be better at life?!?

The Simplest "Robot"





Ok so why can't robots use these awesome kinematic planning algorithms all the time and be better at life?!?

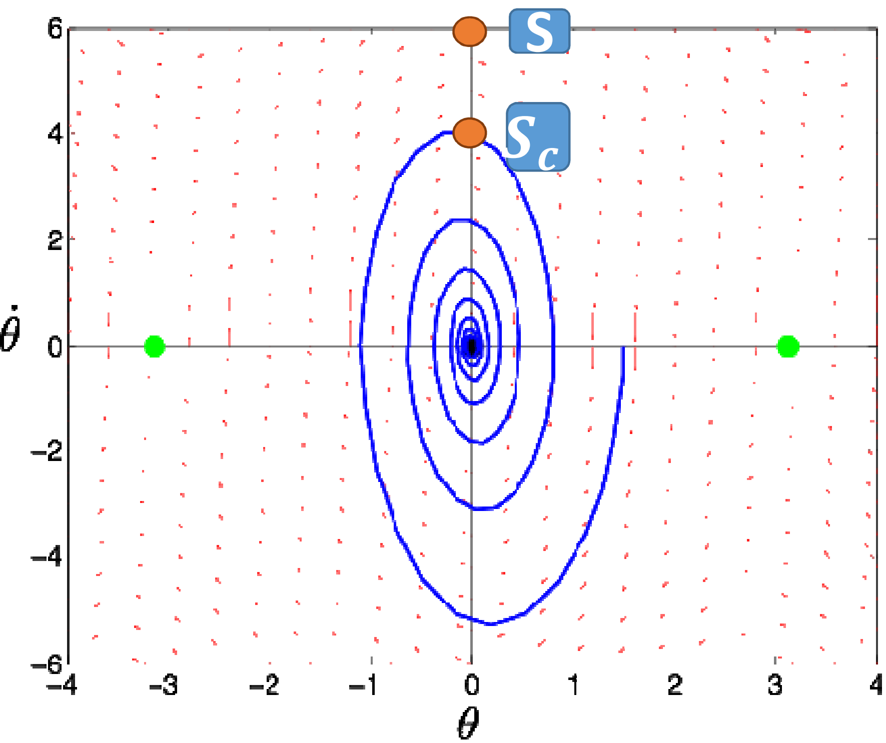


Challenges for Dynamic RRTs

The "connect" operation is complex!

- We need to solve a **boundary value problem** (find a path from sc to s' such that follows the dynamics)
- Basically a "mini" planning problems

Ok so why can’t robots use these awesome kinematic planning algorithms all the time and be better at life?!?



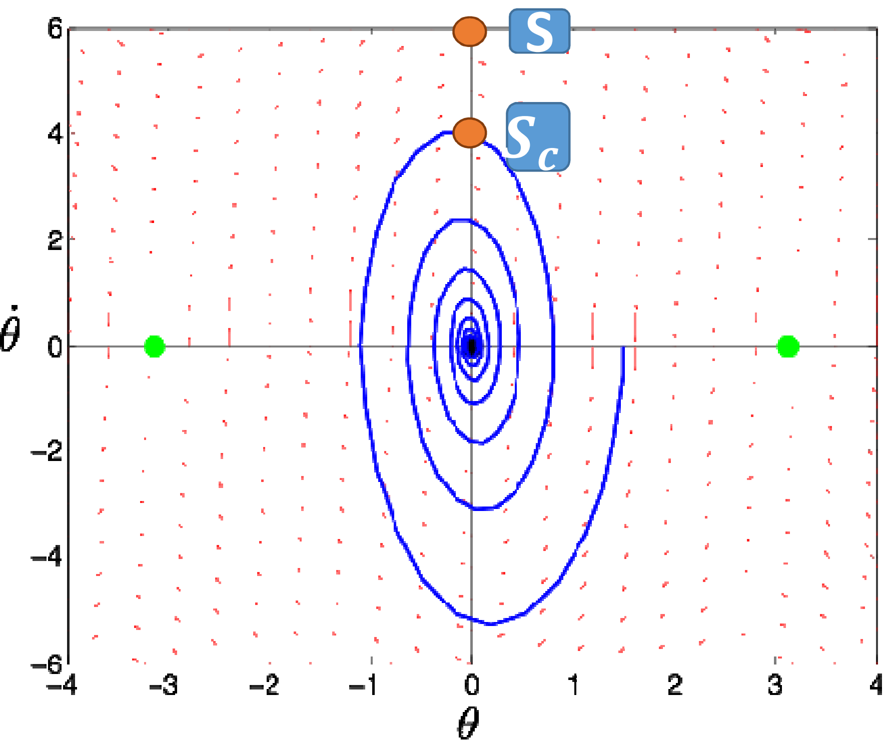
Challenges for Dynamic RRTs

The “connect” operation is complex!

- We need to solve a **boundary value problem** (find a path from sc to s’ such that follows the dynamics)
- P ... ems

Q: Why don’t we just try a discretization of possible actions instead of solving a boundary value problem?

Ok so why can't robots use these awesome kinematic planning algorithms all the time and be better at life?!?



Challenges for Dynamic RRTs

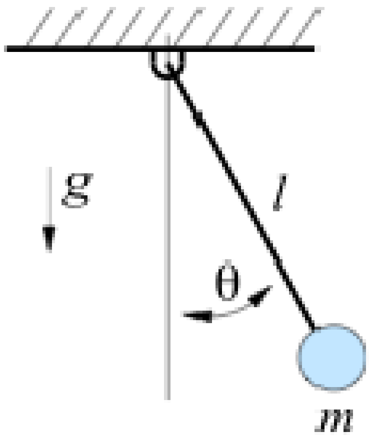
The "connect" operation is complex!

- We need to solve a **boundary value problem** (find a path from sc to s' such that follows the dynamics)
- Basically a "mini" planning problems

Remember from last time with our humanoid robot: $|A| = 10^{20}$

Curse of dimensionality!

Ok so why can't robots use these awesome kinematic planning algorithms all the time and be better at life?!?

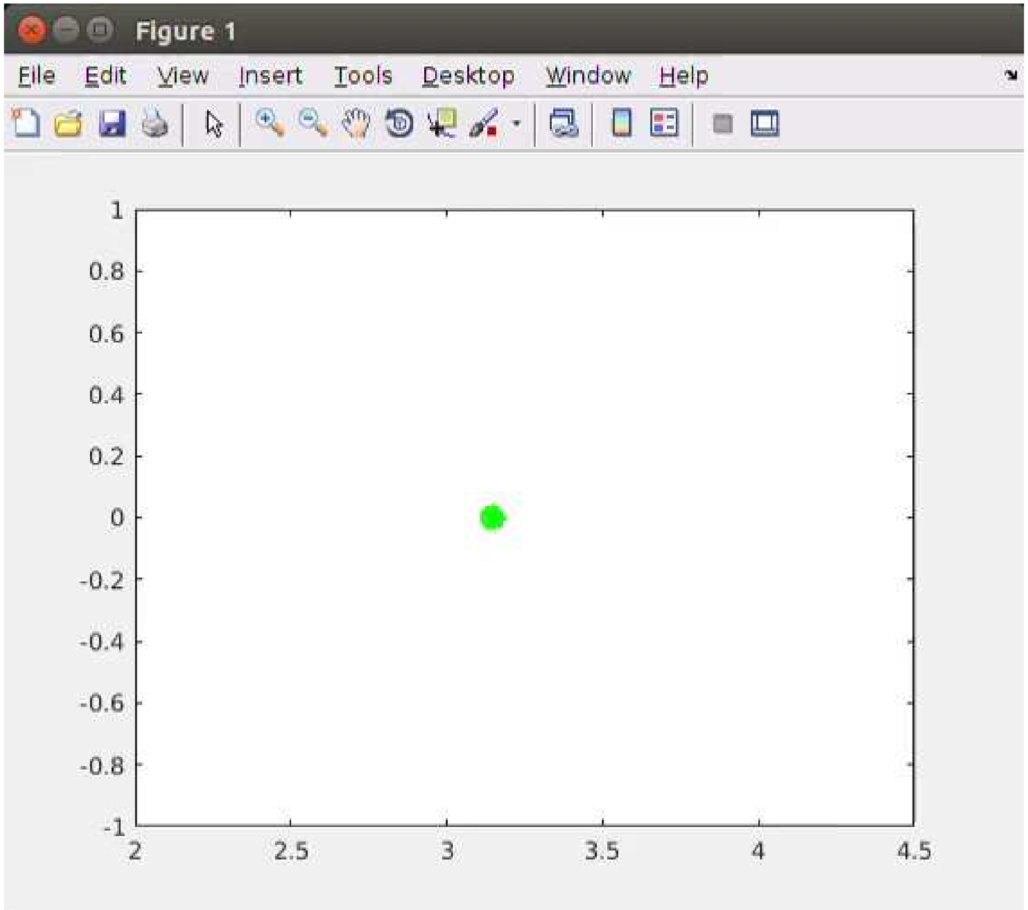


- States: $s = \{\theta, \dot{\theta}\}$ aka angle and angular velocity
- Actions: $a = \tau$ aka torque at joint
- Transitions: $s' = f(s, a)$ aka physics

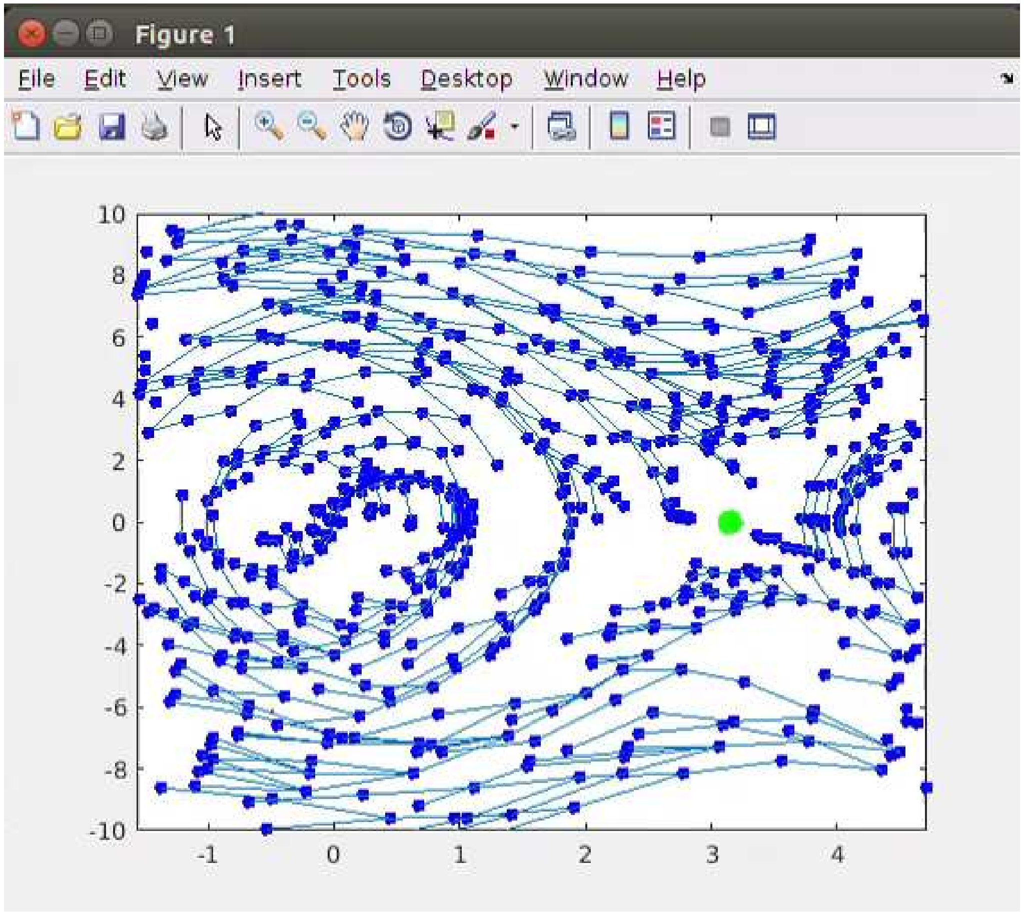
Let's try it anyway for the pendulum since $|\boldsymbol{A}| = \boldsymbol{d}$

Task: **start** from the **stable downward equilibrium (0,0)** and **swing up** to the **unstable upward equilibrium (π,0)**

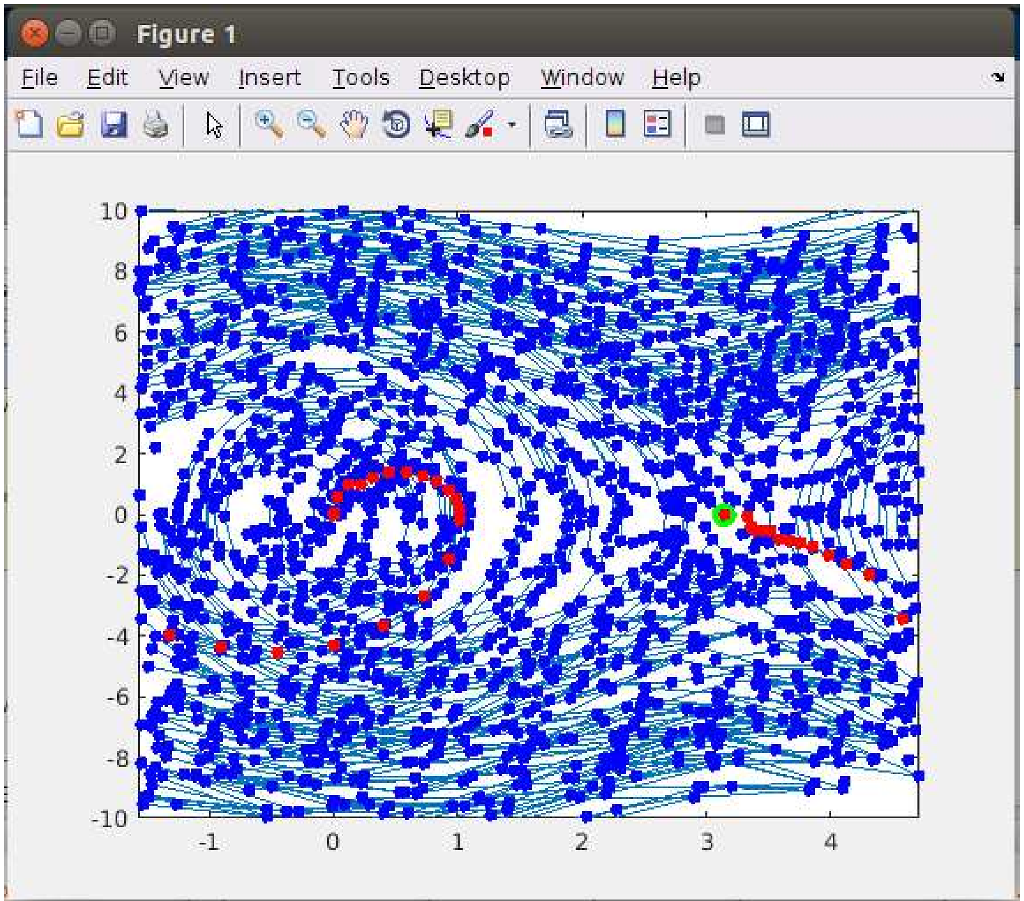
Ok so why can't robots use these awesome kinematic planning algorithms all the time and be better at life?!?



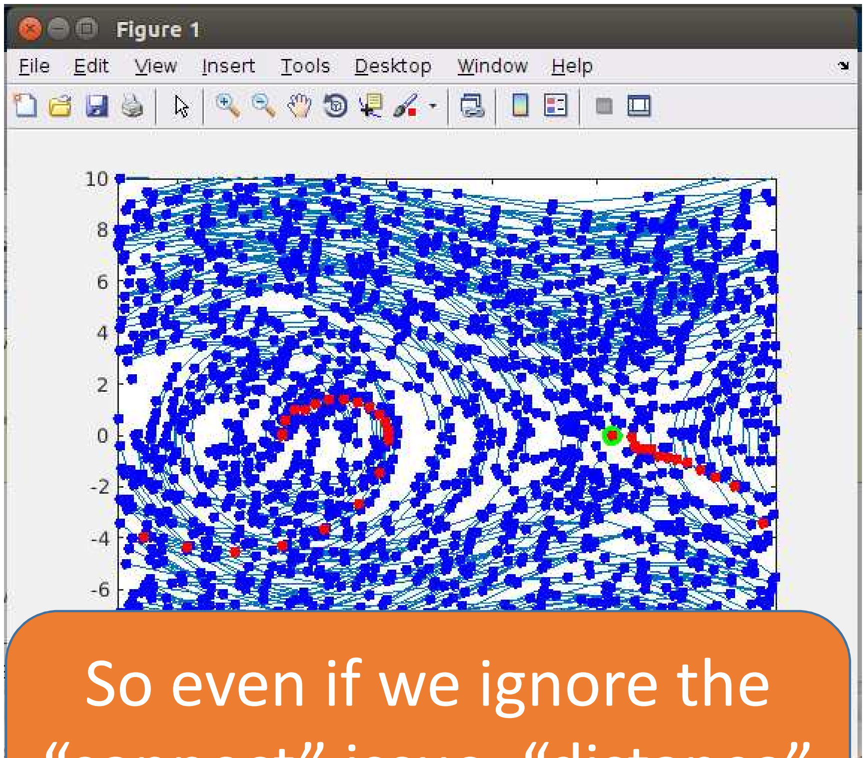
Ok so why can't robots use these awesome kinematic planning algorithms all the time and be better at life?!?



Ok so why can't robots use these awesome kinematic planning algorithms all the time and be better at life?!?



Ok so why can't robots use these awesome kinematic planning algorithms all the time and be better at life?!?



So even if we ignore the "connect" issue, "distance" is still a problem

Challenges for Dynamic RRTs

The "connect" operation is complex!

- We need to solve a **boundary value problem** (find a path from sc to s' such that follows the dynamics)
- Basically a "mini" planning problems

What is the "closest state in the tree"

- The "**distance**" between states of dynamical systems is **not well-defined** (Definitely asymmetric!)

So what do we do?

Can we build robots in such a way that we can ignore dynamics?

- E.g., really strong motors, never move too quickly, etc.

So what do we do?

Can we build robots in such a way that we can ignore dynamics?

- E.g., really strong motors, never move too quickly, etc.
- Short answer is no…

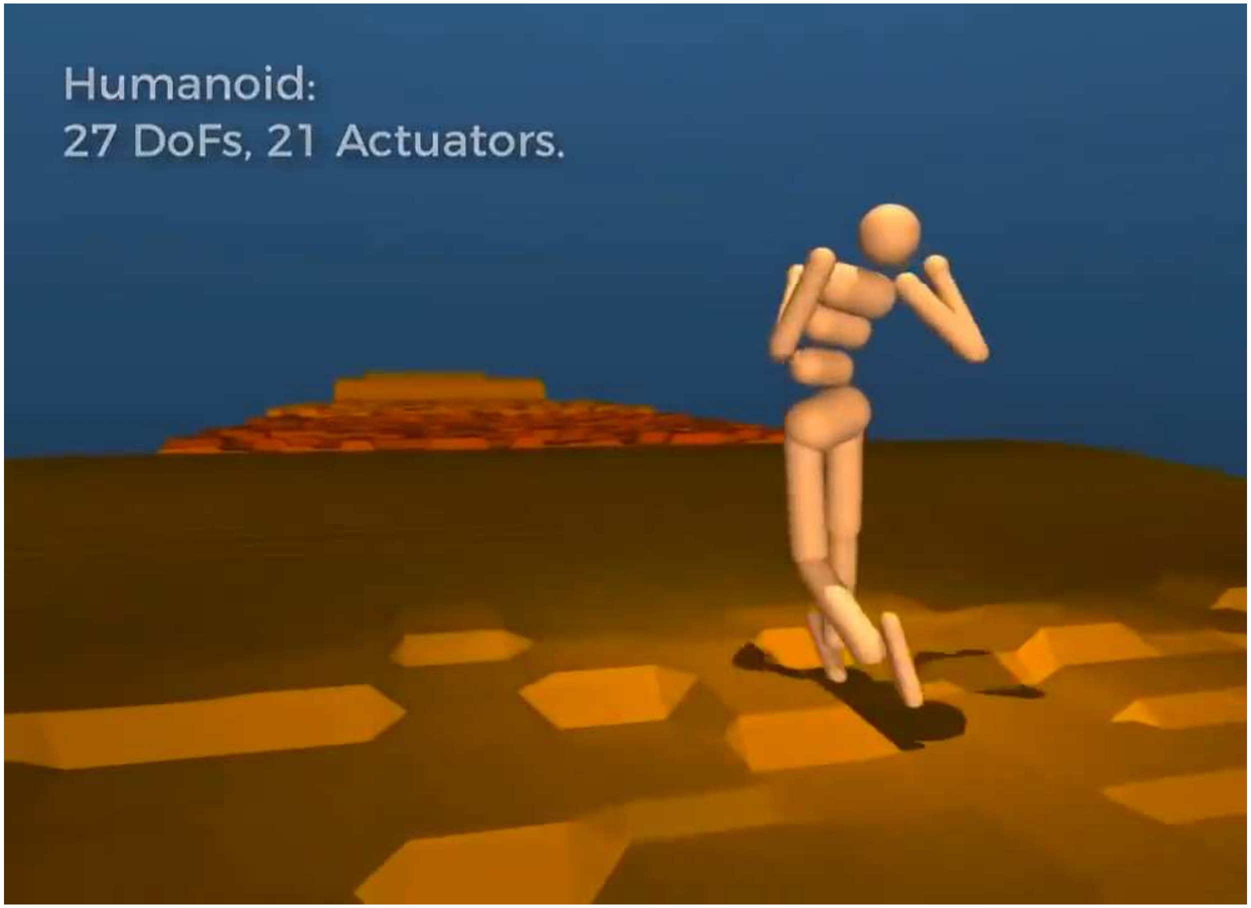
So what do we do?

Can we build robots in such a way that we can ignore dynamics?

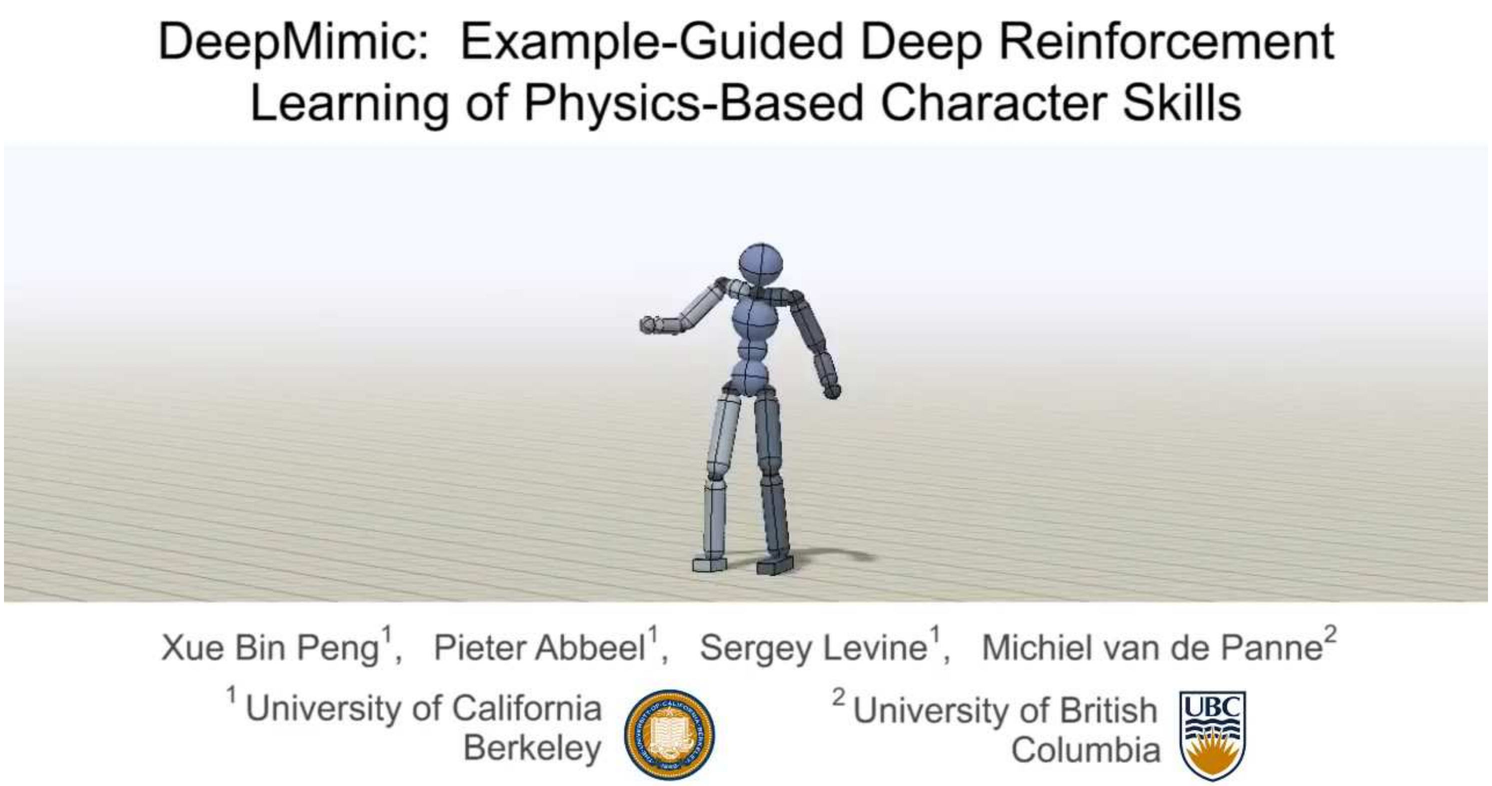
- E.g., really strong motors, never move too quickly, etc.
- Short answer is no…

Can we use RL to learn distance metrics or optimal policies?

So what do we do?



So what do we do?



So what do we do?

Skill	T_{cycle} (s)	$N_{samples}$ (10^6)	NR
Backflip	1.75	72	0.729
Balance Beam	0.73	96	0.783
Baseball Pitch	2.47	57	0.785
Cartwheel	2.72	51	0.804
Crawl	2.93	68	0.932
Dance A	1.62	67	0.863
Dance B	2.53	79	0.822
Frontflip	1.65	81	0.485
Getup Face-Down	3.28	49	0.885
Getup Face-Up	4.02	66	0.838
Headspin	1.92	112	0.640
Jog	0.80	51	0.951

This still doesn't scale well! >100,000,000 seconds is >1000 days

So what do we do?

Can we build robots in such a way that we can ignore dynamics?

- E.g., really strong motors, never move too quickly, etc.
- Short answer is no…

Can we use RL to learn distance metrics or optimal policies?

- This is an open research question and while their have been some very successful examples, they are often correlated with massive training times

Can we just use some key frames?

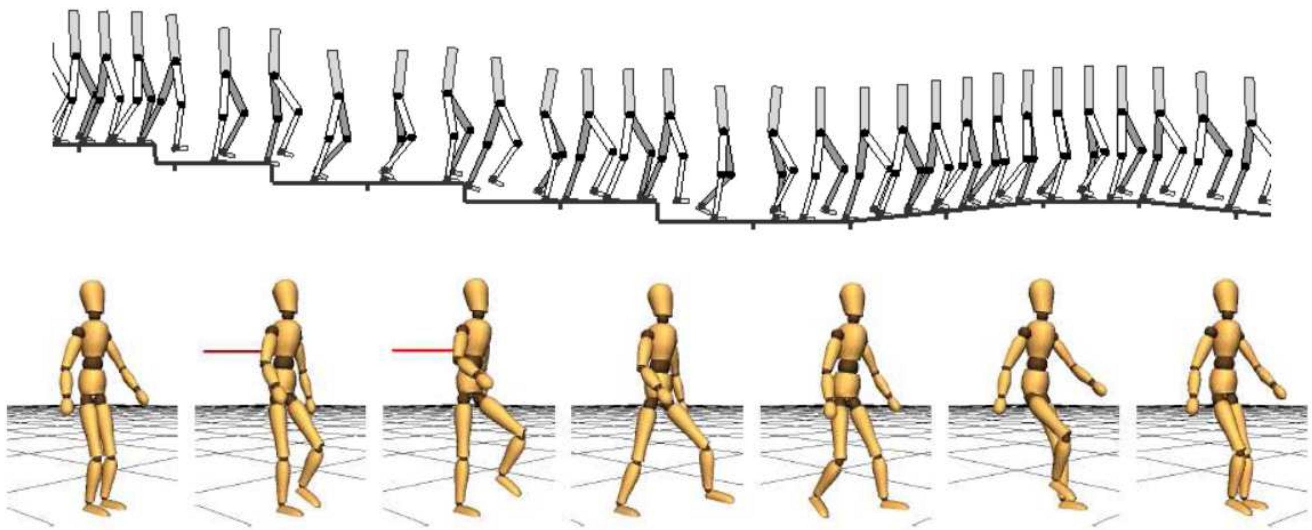
So what do we do?

SIMBICON: Simple Biped Locomotion Control

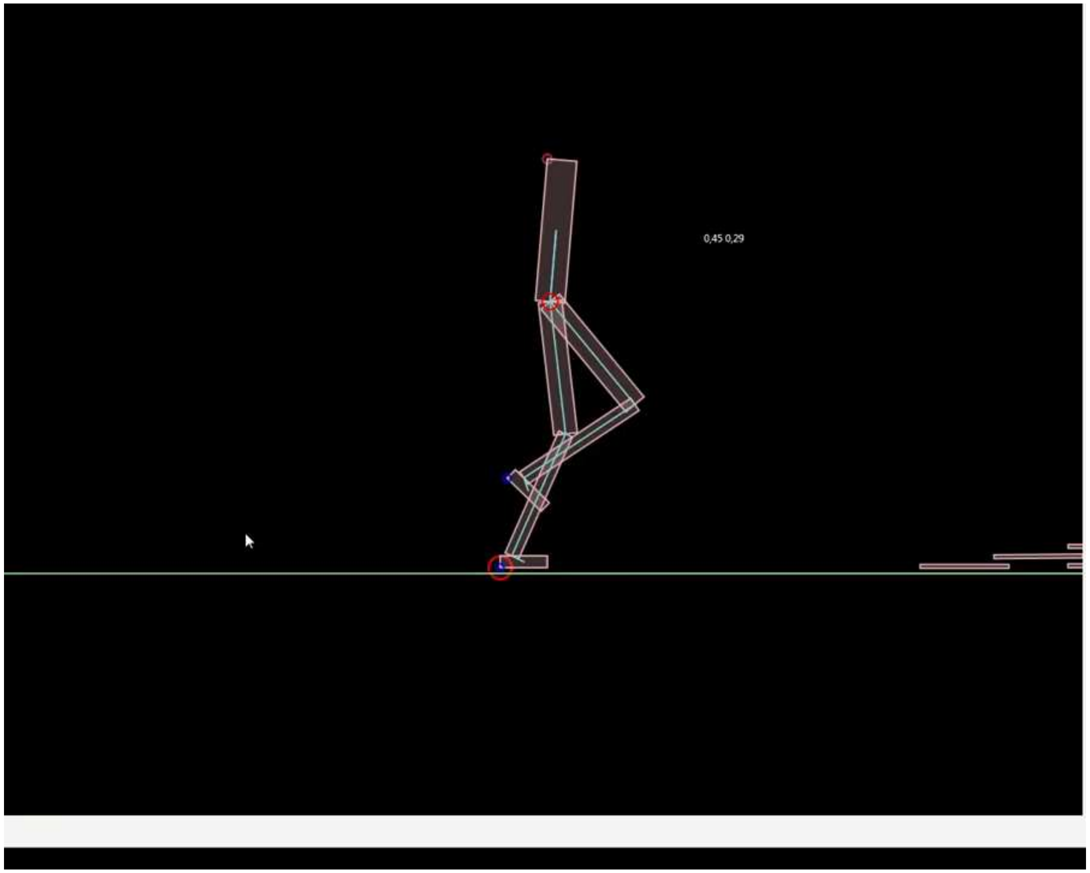
ACM Transaction on Graphics (Proceedings of SIGGRAPH 2007)

KangKang Yin   Kevin Loken   Michiel van de Panne

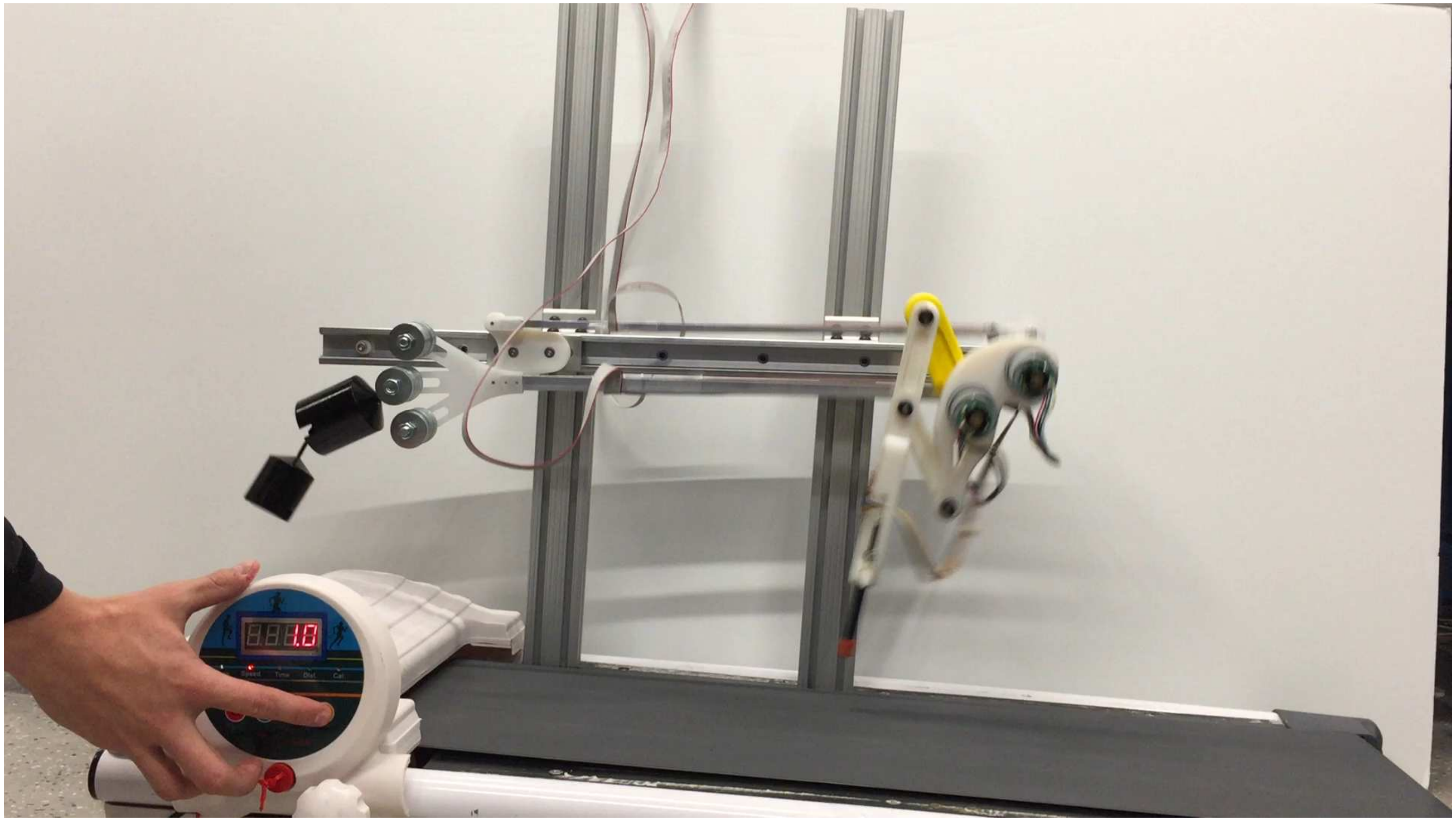
University of British Columbia



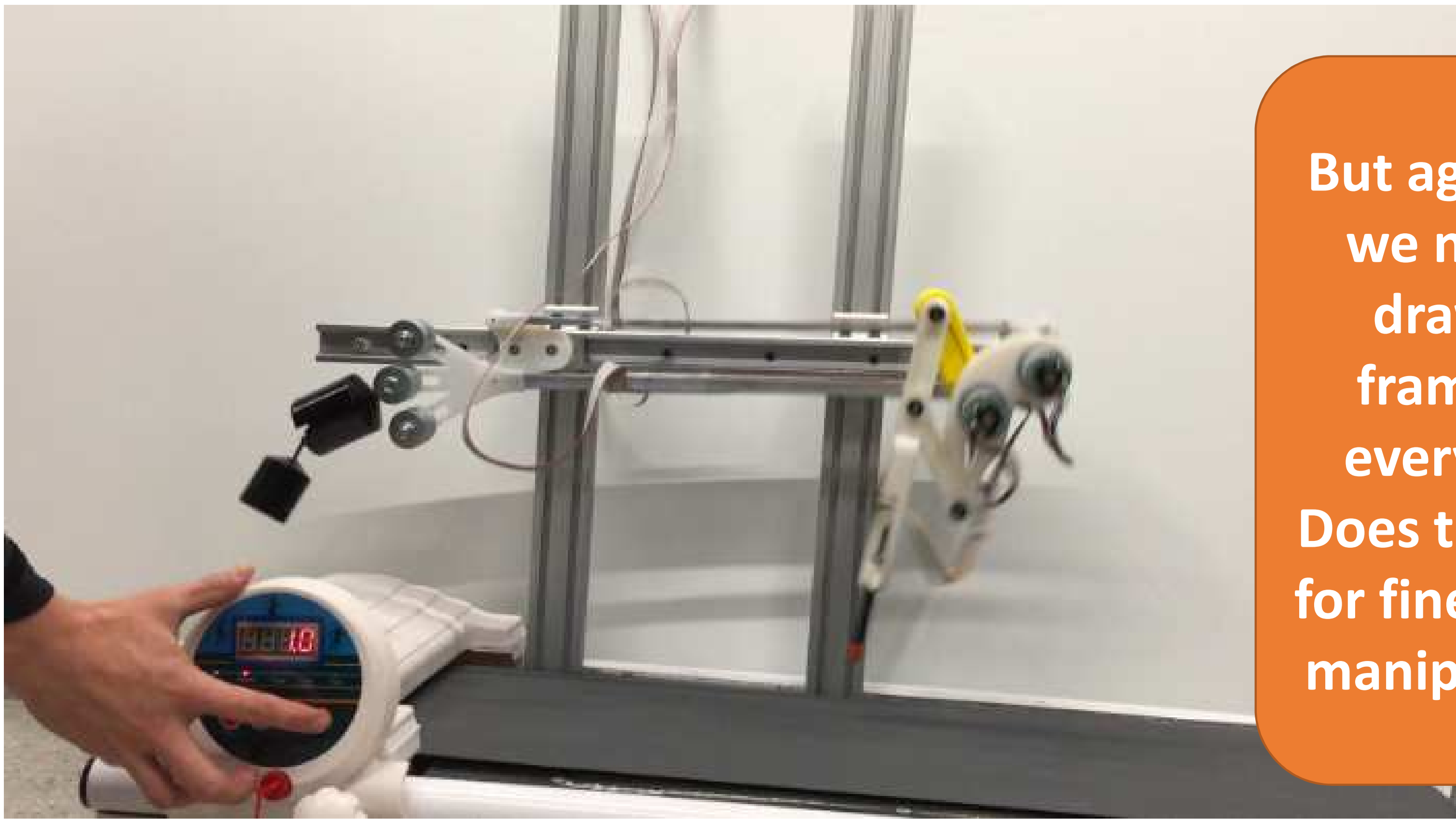
So what do we do?



So what do we do?



So what do we do?



But again now we need to draw key frames for everything. Does that scale for fine grained manipulation?

So what do we do?

Can we build robots in such a way that we can ignore dynamics?

- E.g., really strong motors, never move too quickly, etc.
- Short ans

Can we use R ... optimal policies?

- This is an ... le their have been some very ... often correlated with massive tra

Can we just use some key frames?

So what else can we do?!?

So what do we do?

Lots of math!

So what do we do?

Lots of math!

So what do we do?

Its actually not that bad and the math isn't actually that scary I promise!

So what do we do?

Its actually not that bad and the math isn't actually that scary I promise!

Trajectory Optimization* (starred as in not tested in detail – not as in optimal trajectory optimization)

Why do we keep bringing up optimization stuff and putting * next to it?

Many problems in AI (and ML) can be written as mathematical programs

- In doing so, you can often find interesting properties of the problem (convexity, integerness, etc.) or useful relaxations

Why do we keep bringing up optimization stuff and putting * next to it?

Many problems in AI (and ML) can be written as mathematical programs

- In doing so, you can often find interesting properties of the problem (convexity, integerness, etc.) or useful relaxations

There's a wide variety of tools available for solving convex, non-convex, and even non-smooth optimization problems

Why do we keep bringing up optimization stuff and putting * next to it?

Many problems in AI (and ML) can be written as mathematical programs

- In doing so, you can often find interesting properties of the problem (convexity, integerness, etc.) or useful relaxations

There's a wide variety of tools available for solving convex, non-convex, and even non-smooth optimization problems

Often a good "first thing to try" for new problems

- Sometimes the best option too!

Why do we keep bringing up optimization stuff and putting * next to it?

Many problems in AI (and ML) can be written as mathematical programs

- In doing so, you can often find interesting properties of the problem (convexity, integerness, etc.) or useful relaxations

There's a wide variety of tools available for solving convex, non-convex, and even non-smooth optimization problems

Often a good "first thing to try" for new problems

- Sometimes the best option too!

AI (and ML) are increasingly using optimization as a tool

Why do we keep bringing up optimization stuff and putting * next to it?

Many problems in AI (and ML) can be written as mathematical programs

- In doing so, you can often find interesting properties of the problem (convexity, integerness, etc.) or useful relaxations

There's a wide variety of tools available for solving convex, non-convex, and even non-smooth optimization problems

Often a good "first thing to try" for new problems

- Sometimes the best option too!

AI (and ML) are increasingly using optimization as a tool

Courses @ Harvard: AM 121/221, CS 284

Trajectory Optimization*

Can we write the planning problem down as an optimization problem?

Minimize a cost in each state (e.g., energy used)

Obey physics

Get to the goal

Trajectory Optimization*

Can we write the planning problem down as an optimization problem?

$$\underset{s_0, a_0, \ldots, s_N, a_N}{\text{minimize}} \sum_{k=0}^{N} c(s_k, a_k)$$

$$\text{subject to} \quad s_{k+1} = f(s_k, a_k)$$

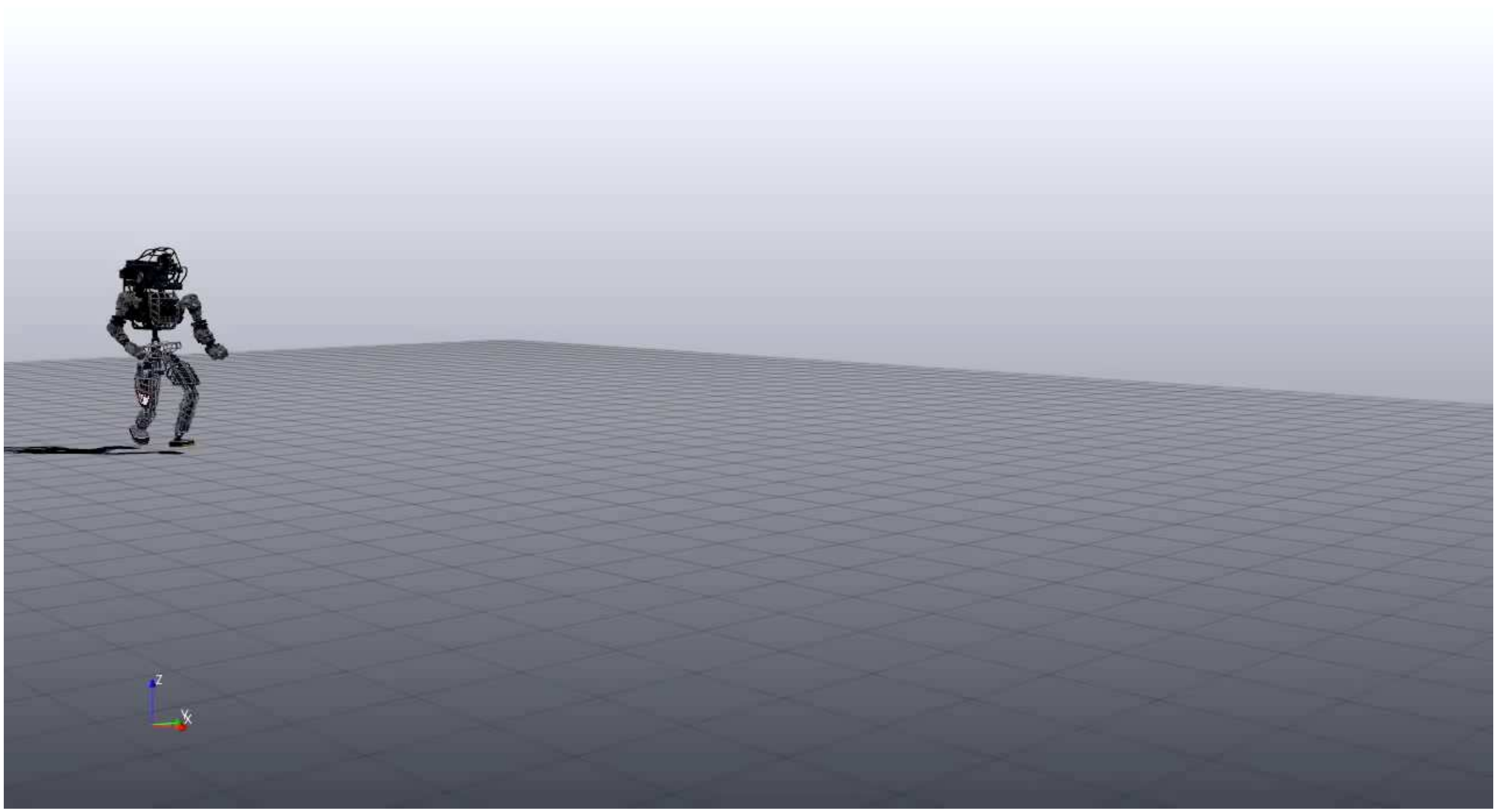
$$s_N = s_{\text{goal}}$$

Minimize a cost in each state (e.g., energy used)

Obey physics

Get to the goal

Atlas 1.0 Trajectory Optimization*



Trajectory Optimization*

Aka Value/Policy Iteration!

But wait can't we just use those Bellman updates to solve this?

- We can start at the goal state and then work backwards computing the lowest cost actions to get to all states all the way back to the start state

$$\underset{s_0,a_0,\ldots,s_N,a_N}{\text{minimize}} \sum_{k=0}^{N} c(s_k, a_k)$$

$$\text{subject to} \quad s_{k+1} = f(s_k, a_k)$$

$$s_N = s_{\text{goal}}$$

$$V_0(s_N) = c(s_N, a_N)$$

$$V_{k+1}(s) = \min_a c(s,a) + V_k\big(f(s,a)\big)$$

Trajectory Optimization*

Aka Value/Policy Iteration!

But wait can't we just use those Bellman updates to solve this?

- We can start at the goal state and then work backwards computing the lowest cost actions to get to all states all the way back to the start state

$$\underset{s_0, a_0, \ldots, s_N, a_N}{\text{minimize}} \sum_{k=0}^{N} c(s_k, a_k)$$

$$\text{subject to} \quad s_{k+1} = f(s_k, a_k)$$

$$s_N = s_{\text{goal}}$$

$$V_0(s_N) = c(s_N, a_N)$$

$$V_{k+1}(s) = \min_a c(s, a) + V_k\big(f(s, a)\big)$$

Q: Will this work?

Trajectory Optimization*

Aka Value/Policy Iteration!

But wait can't we just use those Bellman updates to solve this?

- We can start at the goal state and then work backwards computing the lowest cost actions to get to all states all the way back to the start state

$$\underset{s_0,a_0,\ldots,s_N,a_N}{\text{minimize}} \sum_{k=0}^{N} c(s_k, a_k)$$

$$\text{subject to} \quad s_{k+1} = f(s_k, a_k)$$

$$s_N = s_{\text{goal}}$$

$$V_0(s_N) = c(s_N, a_N)$$

$$V_{k+1}(s) = \min_a c(s,a) + V_k(f(s,a))$$

$|S| = |A| = 10^{20}$

Curse of dimensionality again!

Trajectory Optimization*

What if instead of finding a globally optimal path we search for a locally optimal path (off of some initial condition)?

Trajectory Optimization*

What if instead of finding a globally optimal path we search for a locally optimal path (off of some initial condition)?

- This works well in practice (think local search)

Trajectory Optimization*

What if instead of finding a globally optimal path we search for a locally optimal path (off of some initial condition)?

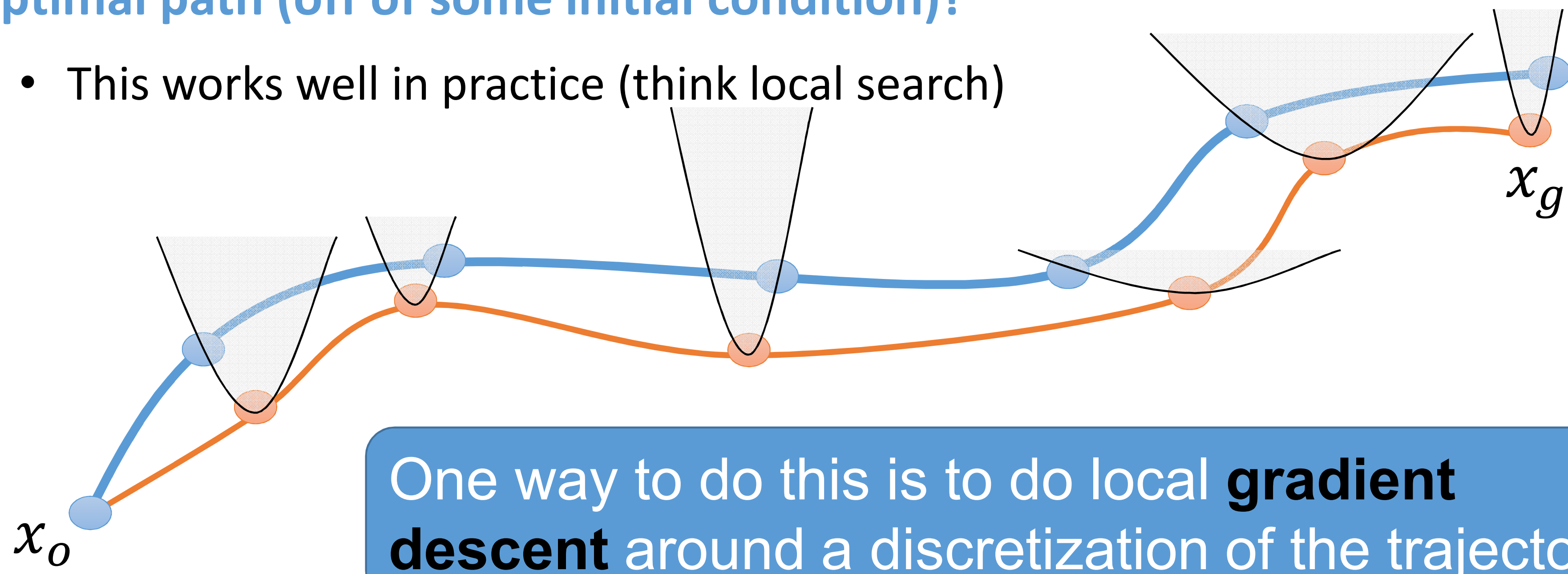
- This works well in practice (think local search)

By making slight perturbations to the current trajectory (blue) we can get to the goal (orange)

Trajectory Optimization*

What if instead of finding a globally optimal path we search for a locally optimal path (off of some initial condition)?

- This works well in practice (think local search)



One way to do this is to do local **gradient descent** around a discretization of the trajectory

Trajectory Optimization*

What if instead of finding a globally optimal path we search for a locally optimal path (off of some initial condition)?

- This works well in practice (think local search)

There are also a whole host of algorithms one can use to solve these problems including:

- DDP, SQP, Interior-Point Methods, Trust-Region Methods, etc.

Trajectory Optimization*

What if instead of finding a globally optimal path we search for a locally optimal path (off of some initial condition)?

- This works well in practice (think local search)

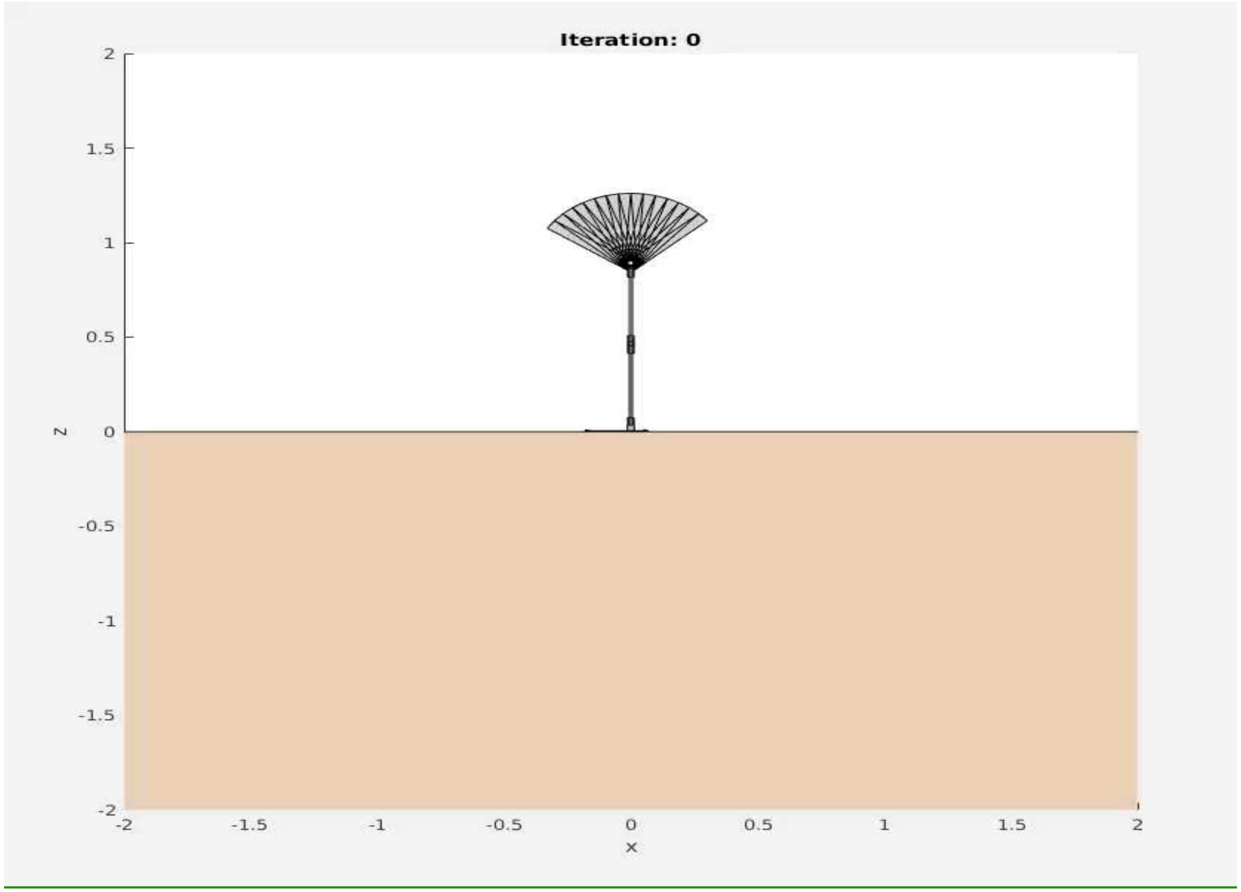
There are also a whole host of algorithms one can use to solve these problems including:

- DDP, SQP, Interior-Point Methods, Trust-Region Methods, etc.

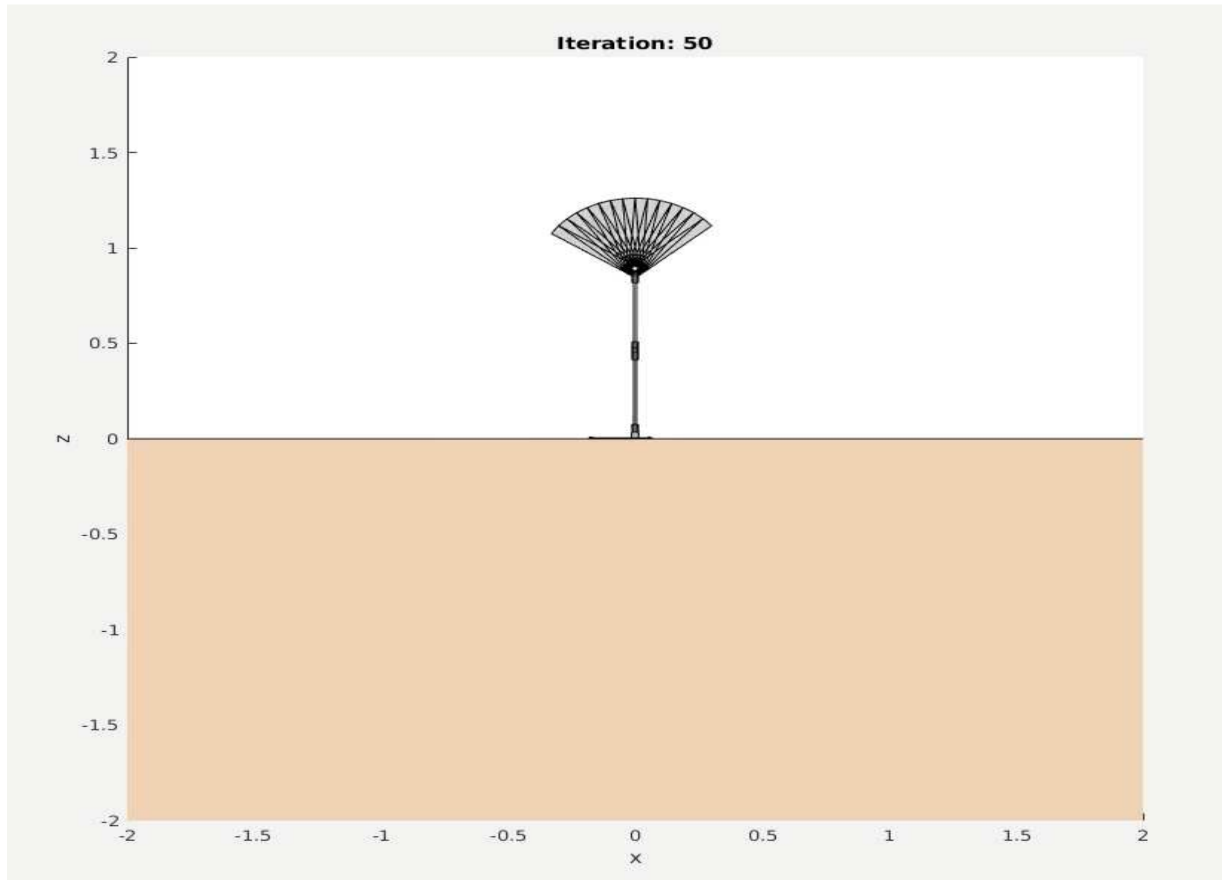
And you can use off-the-shelf solvers to solve these problems. Popular solvers include:

- SNOPT, IPOPT, NLOPT, fmincon (MATLAB), etc.

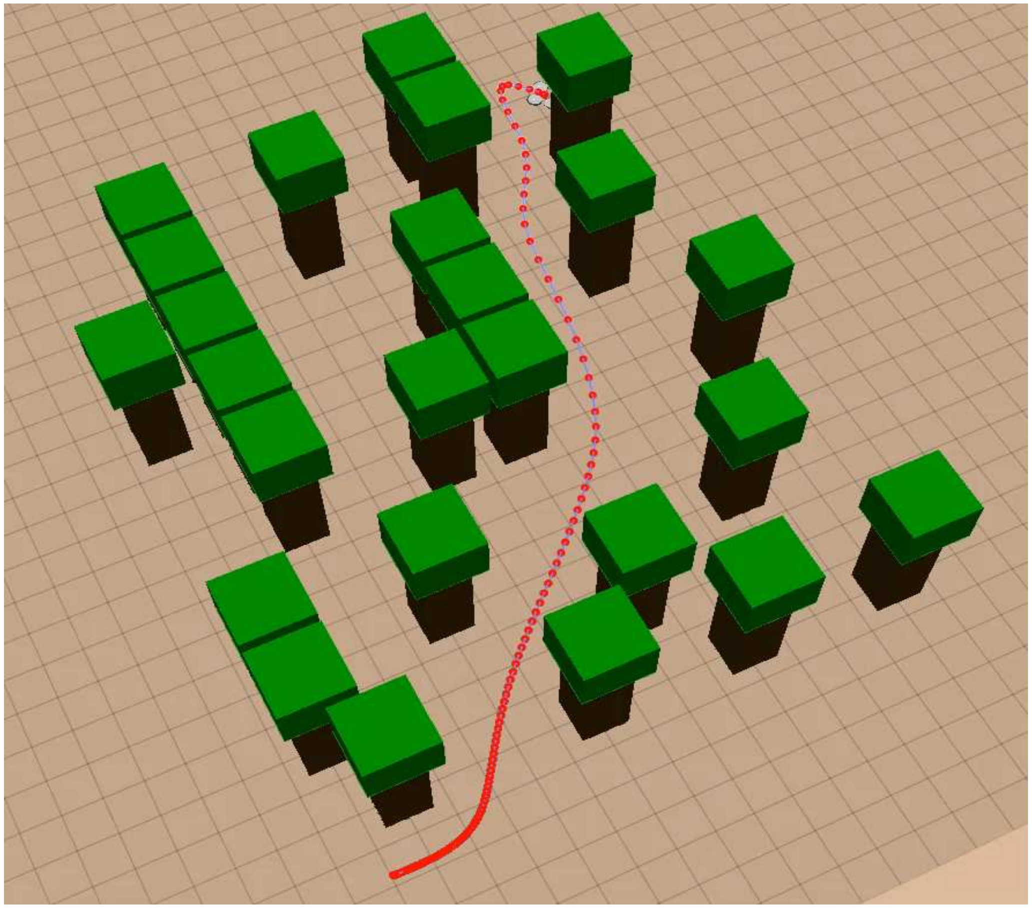
Spring Flamingo Trajectory Optimization*



Spring Flamingo Trajectory Optimization*



Quadrotor in Forest Trajectory Optimization*



Trajectory Optimization in practice*

How can I use trajectory optimization in practice?

Trajectory Optimization in practice*

How can I use trajectory optimization in practice?

1. Figure out your robot’s dynamics

Trajectory Optimization in practice*

How can I use trajectory optimization in practice?

1. Figure out your robot’s dynamics
2. Invent a cost function

Trajectory Optimization in practice*

How can I use trajectory optimization in practice?

1. Figure out your robot's dynamics
2. Invent a cost function
3. Add constraints for obstacles, etc.

Trajectory Optimization in practice*

How can I use trajectory optimization in practice?

1. Figure out your robot's dynamics
2. Invent a cost function
3. Add constraints for obstacles, etc.
4. Send problem to your favorite solver

Trajectory Optimization in practice*

How can I use trajectory optimization in practice?

1. Figure out your robot's dynamics
2. Invent a cost function
3. Add constraints for obstacles, etc.
4. Send problem to your favorite solver
5. Iterate on cost/constraint formulation if the result isn't what you expect (often true)

Trajectory Optimization in practice *

How can I use trajectory optimization in practice?

1. Figure out your robot’s dynamics
2. Invent a cost function
3. Add constraints for obstacles, etc.
4. Send problem to your favorite solver
5. Iterate on cost/constraint formulation if the result isn’t what you expect (often true)

The above is very “black box”… can you do better by diving into the details of solvers? Yes! But that’s another course entirely!

Trajectory Optimization*

So trajectory optimization solves everything right?

- Can handle full robot **dynamics**

Trajectory Optimization*

So trajectory optimization solves everything right?

- Can handle full robot **dynamics**
- No need for distance metrics

Trajectory Optimization*

So trajectory optimization solves everything right?

- Can handle full robot **dynamics**
- No need for distance metrics
- Finds a **locally optimal** solution – no weird paths coming out!

Trajectory Optimization*

So trajectory optimization solves everything right?

- Can handle full robot **dynamics**
- No need for distance metrics
- Finds a **locally optimal** solution – no weird paths coming out!

But….

- **Not globally optimal** (will often get stuck in local minima)

Trajectory Optimization*

So trajectory optimization solves everything right?

- Can handle full robot **dynamics**
- No need for distance metrics
- Finds a **locally optimal** solution – no weird paths coming out!

But….

- **Not globally optimal** (will often get stuck in local minima)
- **Not even complete** (problems are often non-convex so it may not even find a feasible solution)
  - This is driven by the fact that NLP solvers are not a “technology” yet (there is still a lot of open research questions)

Trajectory Optimization*

So trajectory optimization solves everything right?

- Can handle full robot **dynamics**
- No need for distance metrics
- Finds a **locally optimal** solution – no weird paths coming out!

No free lunch strikes again!

But….

- **Not globally optimal** (will often get stuck in local minima)
- **Not even complete** (problems are often non-convex so it may not even find a feasible solution)
- **Also generally slow**

Trajectory Optimization*

Take CS 284 to learn more!

So trajectory optimization solves everything right?

- Can handle full robot **dynamics**
- No need for distance metrics
- Finds a **locally optimal** solution – no weird paths coming out!

No free lunch strikes again!

But….

- **Not globally optimal** (will often get stuck in local minima)
- **Not even complete** (problems are often non-convex so it may not even find a feasible solution)
- **Also generally slow**

Also ask me about my research later because these are the kinds of things I am working to solve!

Trajectory Optimization*

Summary

1. **Policies** are not feasible for most robots, so we **plan** instead
2. Robot planning usually involves both **task and configuration spaces**
3. **RRTs and PRMs**: powerful tools based on very simple ideas
   - **Probabilistically complete**
   - **Single-query (RRT) vs. Multi-query (PRM)**
4. For many real problems, **collision checking** can be expensive
5. **RRT***: optimal and complete, but can be tricky to apply to dynamic tasks (i.e. where the physics matters, not just geometry)
6. **Trajectory optimization** (CS 284): a broad class of methods built on top of mathematical programming and "state of the art"