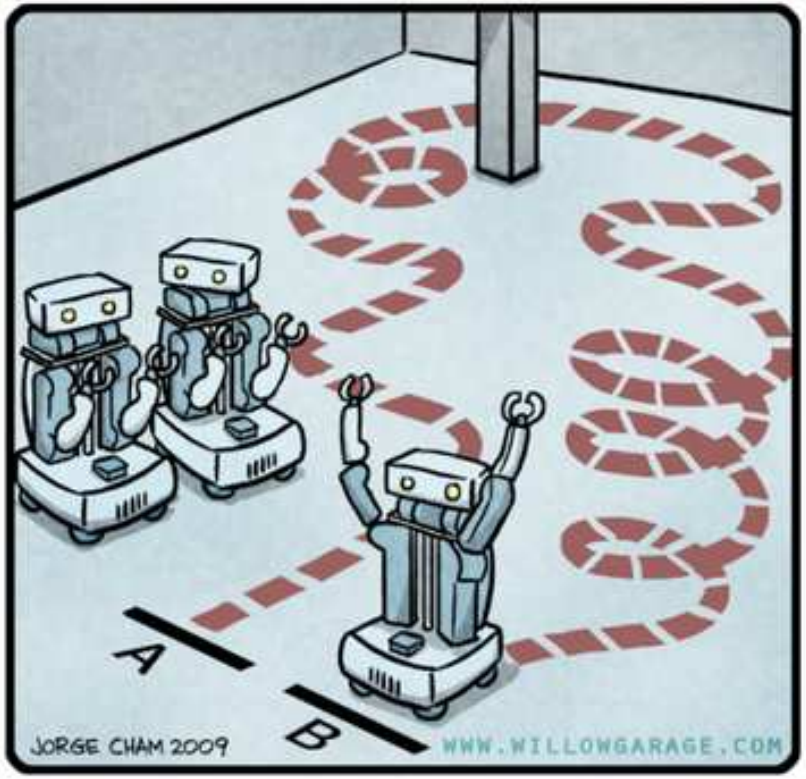
CS 249r: Special Topics in Edge Computing

Intro to Autonomous Systems / Robotics Part 2

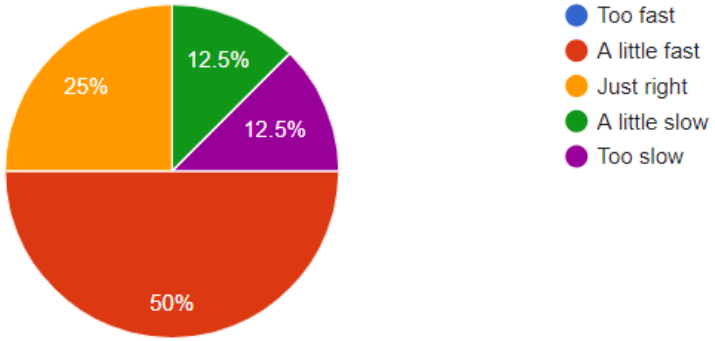


"HIS PATH-PLANNING MAY BE SUB-OPTIMAL, BUT IT'S GOT FLAIR."

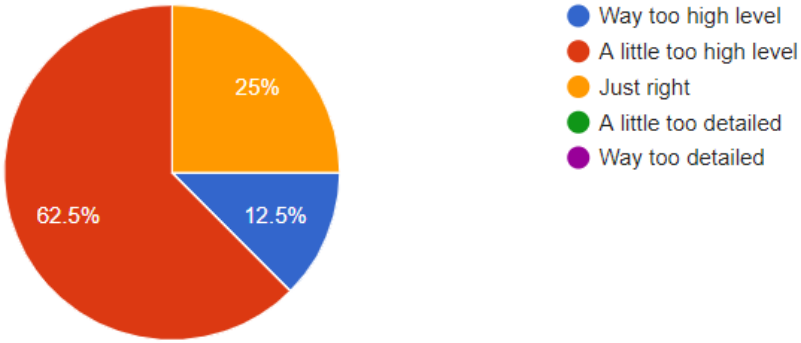
Brian Plancher
Fall 2019

Feedback from last class

How was the pace of class today?



How was the depth of the content covered today?

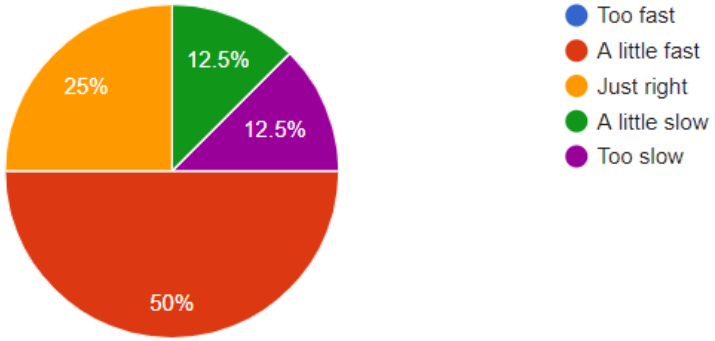


1. Pace is a tad fast
2. Get more technical/depth

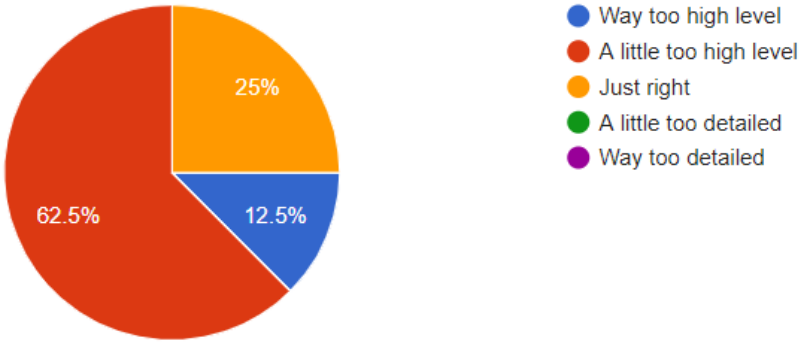
Feedback from last class

Also thanks for the open ended feedback!

How was the pace of class today?



How was the depth of the content covered today?



1. Pace is a tad fast
2. Get more technical/depth

Your homework for next class

We have posted a tentative paper list to Canvas (along with PDFs and links)

Start to think about which papers you want – I will send a link to vote for preferences in a week or so!

If you have an idea for a paper not on the list please run it by us and we may be willing to swap it in!

Pre-Reads for Intro to Domain Specific Architectures

Is dark silicon useful? Harnessing the four horsemen of the coming dar
apocalypse: https://ieeexplore.ieee.org/document/6241647

Turing Lecture: A New Golden Age for Computer Architecture: https://californiaconsultants.org/wp-content/uploads/2018/04/CNSV-1806-Patterson.pdf (watch the lecture its great!)

Your homework for next class

Pre-Reads for Intro to Doma
Specific Architectures

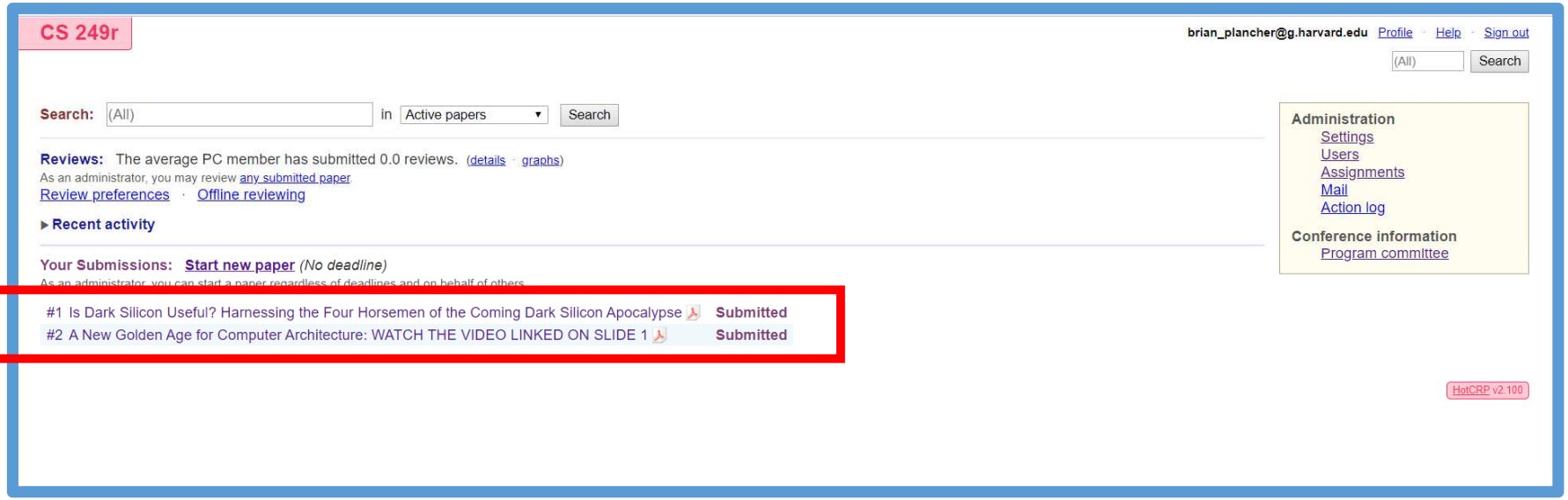
Is dark silicon useful? Harnessing the four horse
apocalypse: https://ieeexplore.ieee.org/docume

Turing Lecture: A New Golden Age for Computer Architecture: https://californiaconsultants.org/wp-content/uploads/2018/04/CNSV-1806-Patterson.pdf (watch the lecture its great!)

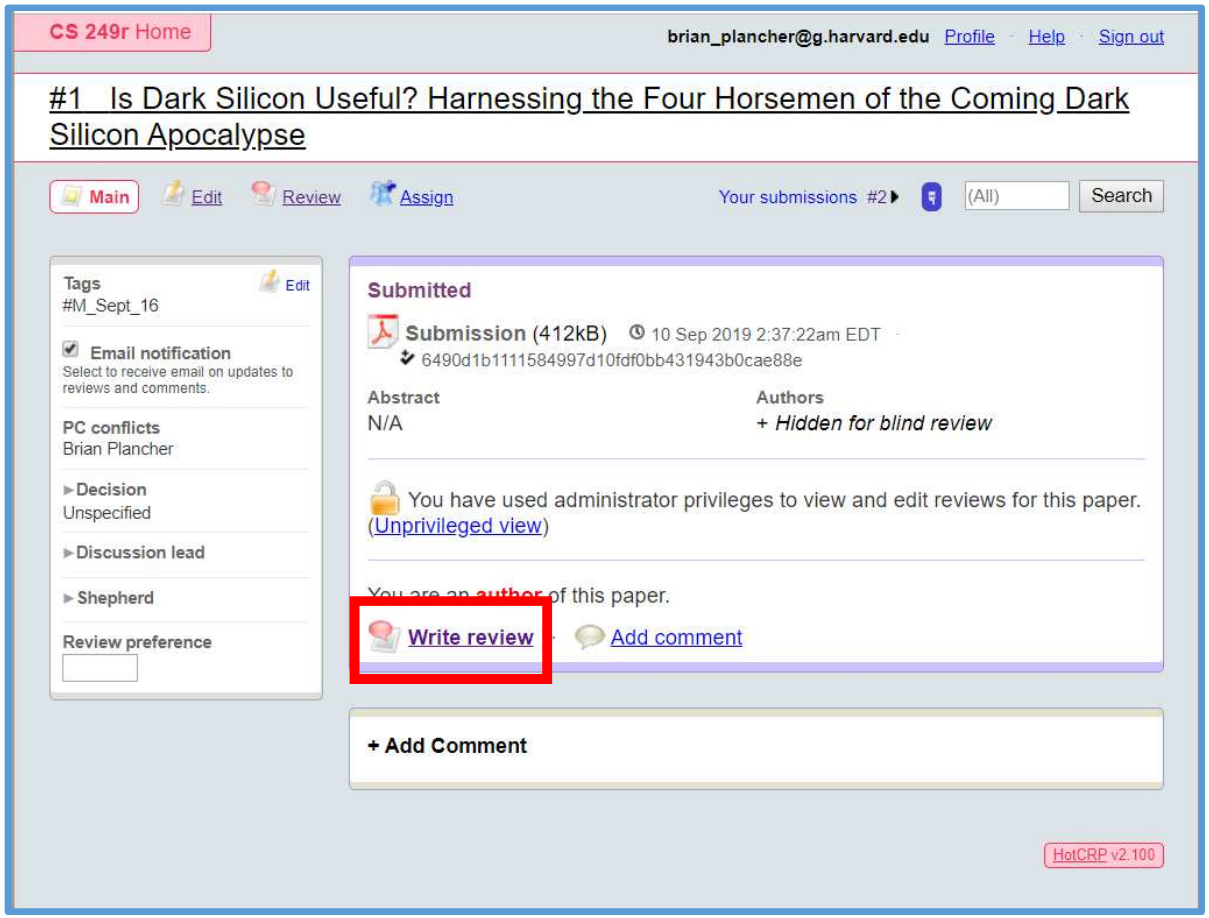
Were going to use **HOTCRP** (linked on Canvas and https://www.eecs.harvard.edu/cs249r/) for these for Monday – you will get an email from **Glenn Holloway** with a Password to access the site. (I am giving him the full roster as of today)

Your homework for next class

Click on a paper to access that paper's page

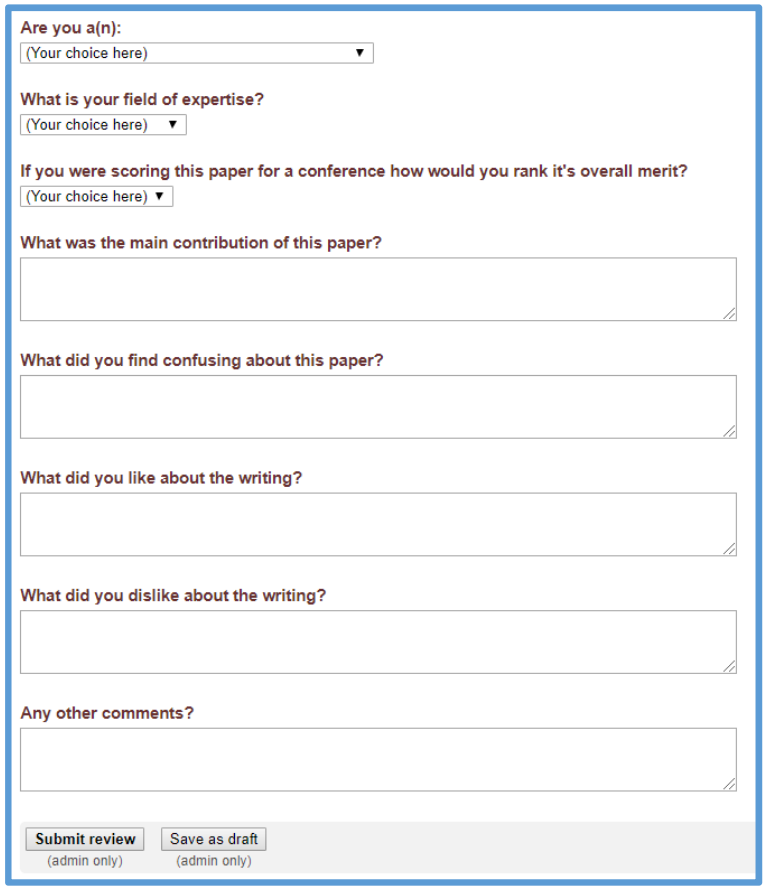


Your homework for next class



Then click “Write review” to open up the form to submit a “review”

Your homework for next class

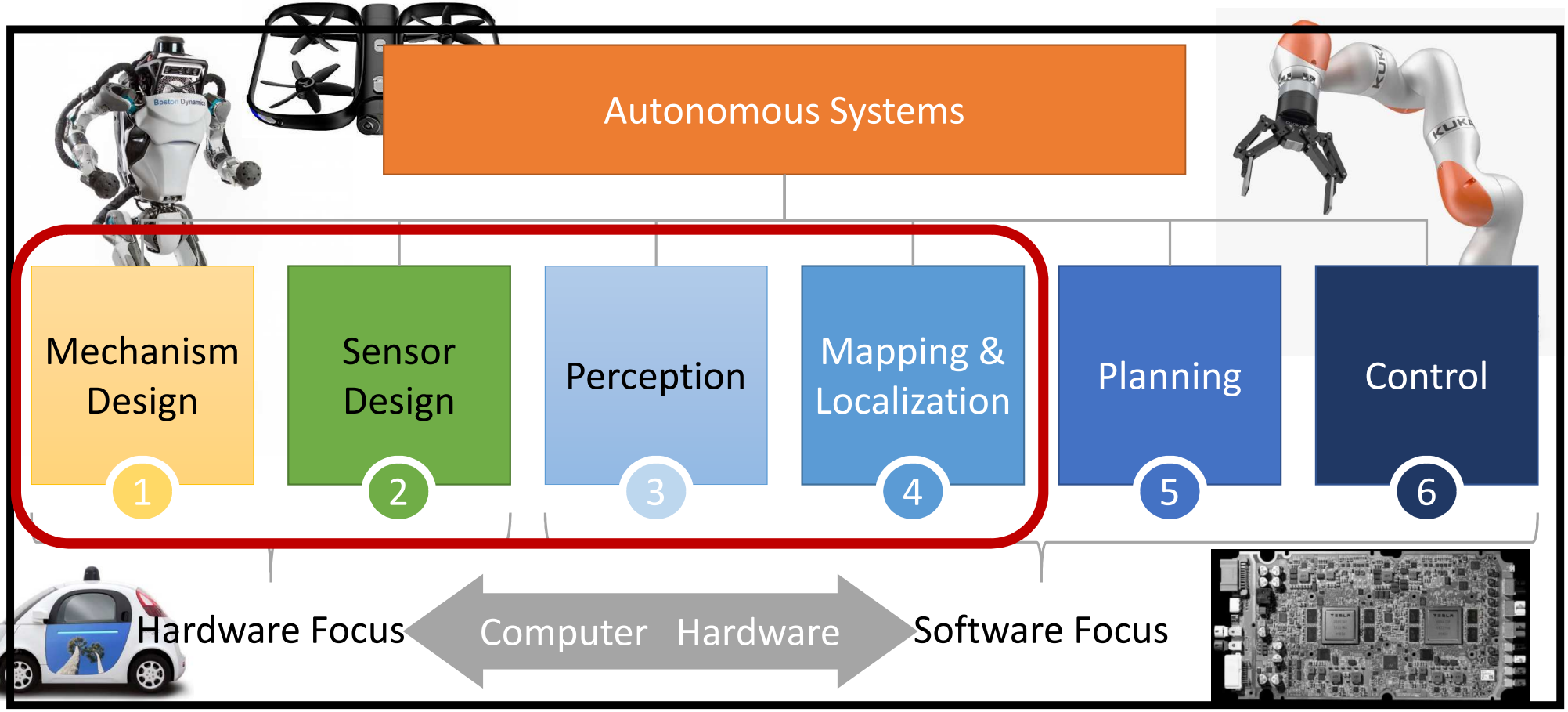


Then just fill it out and submit and you'll be good to go!

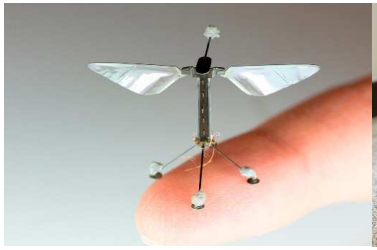
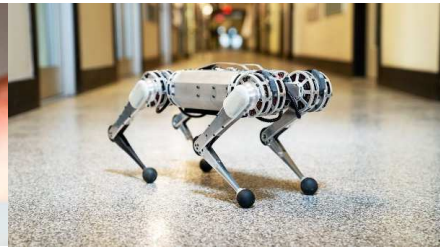
The goal for the next couple of lectures is to develop a high level understanding of:

1. What is an autonomous system
2. Key problems for autonomous systems
3. Some of the most important (classes of) algorithms in robotics
4. The model based vs. model free tradeoff
5. The online vs offline tradeoff
6. The no free lunch theorem and the need for approximations
7. How computer systems / architecture design has and can play a role in improving autonomous systems

Autonomous Systems / Robotics is a BIG space

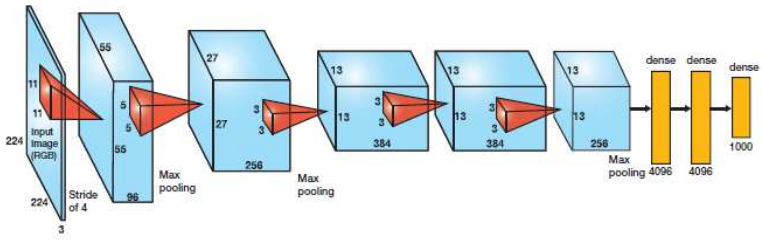


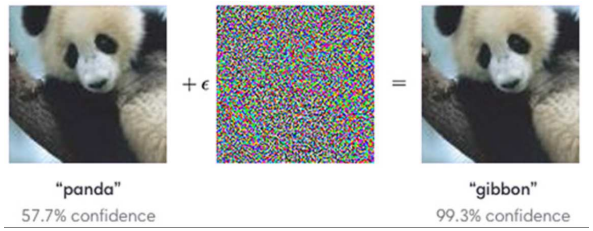
12 Key Takeaways:

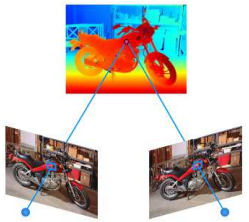




1. When designing algorithms for robots you need to understand the physical capabilities of the robot and you (potentially) need to understand how to model its physical behaviors
2. Different kinds of systems will have different power, weight, and performance budgets for computer hardware

3 Key Takeaways:

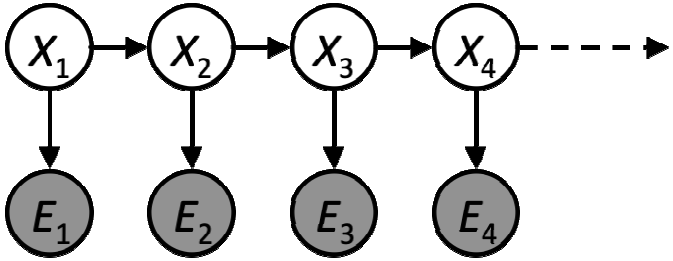


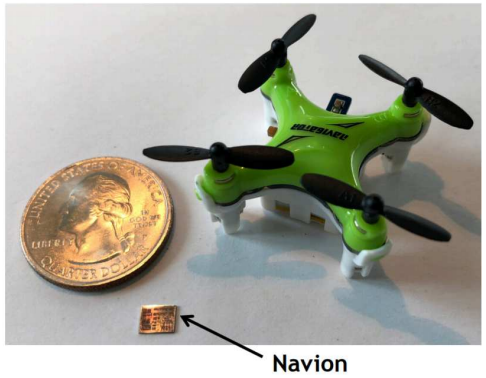


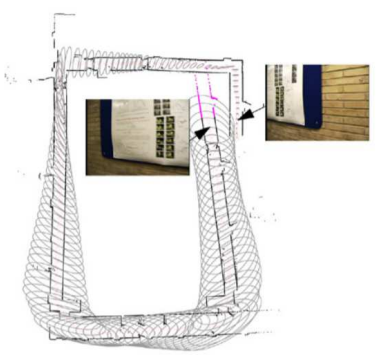


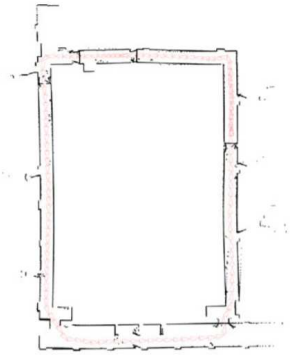
1. As of today it seems like CNNs that automate the design and summary of salient features via convolution are the way to go
   - But/and will need specialized NN running on specialized accelerator chips to get them small enough to fit on small power constrained autonomous systems (e.g., small drones)
   - And we will need to find ways to secure them against attacks!
2. Also, other more targeted problems such as Stereo Depth seem to need accelerators!

4 Key Takeaways:



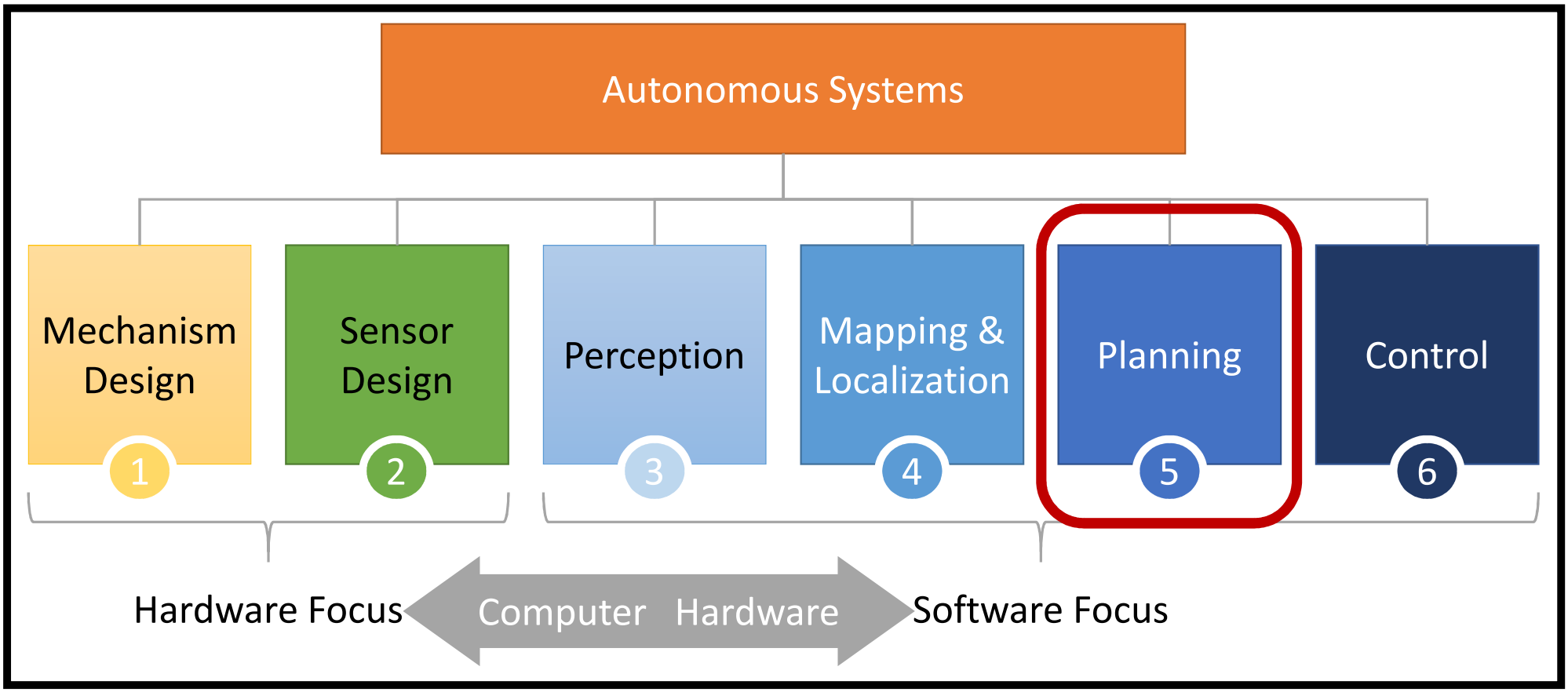




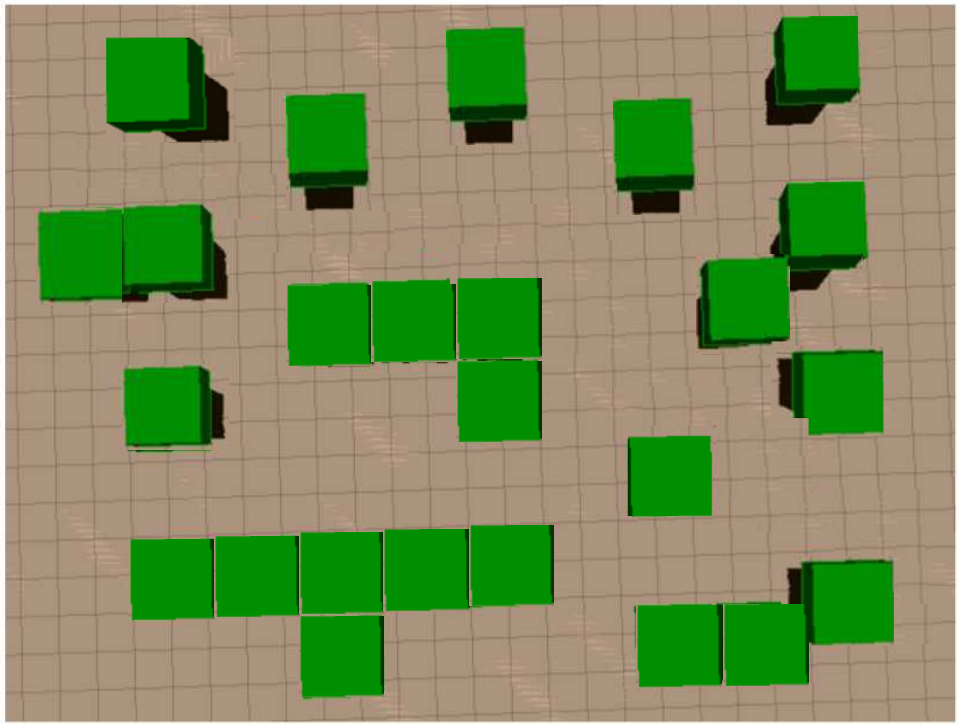


1. The Kalman/Particle Filter uses probability to solve the localization problem but modeling and/or approximations are needed for it to run efficiently online
2. Mapping quickly becomes a memory storage problem
3. Constrained form factors (aka tiny drones) will need novel accelerators to allow for autonomy

Autonomous Systems / Robotics is a BIG space

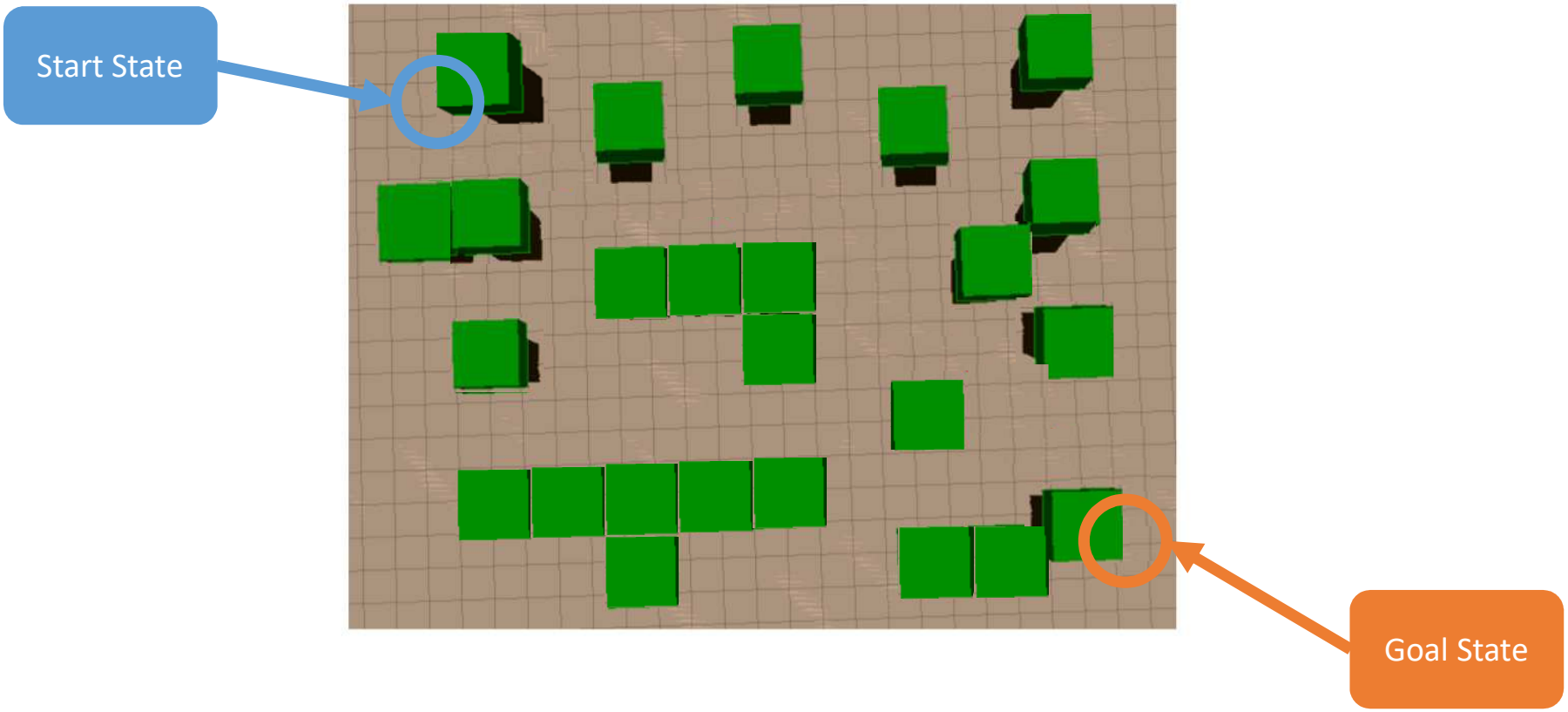


Planning is the process of computing an action plan for a robot based on the previously computed map



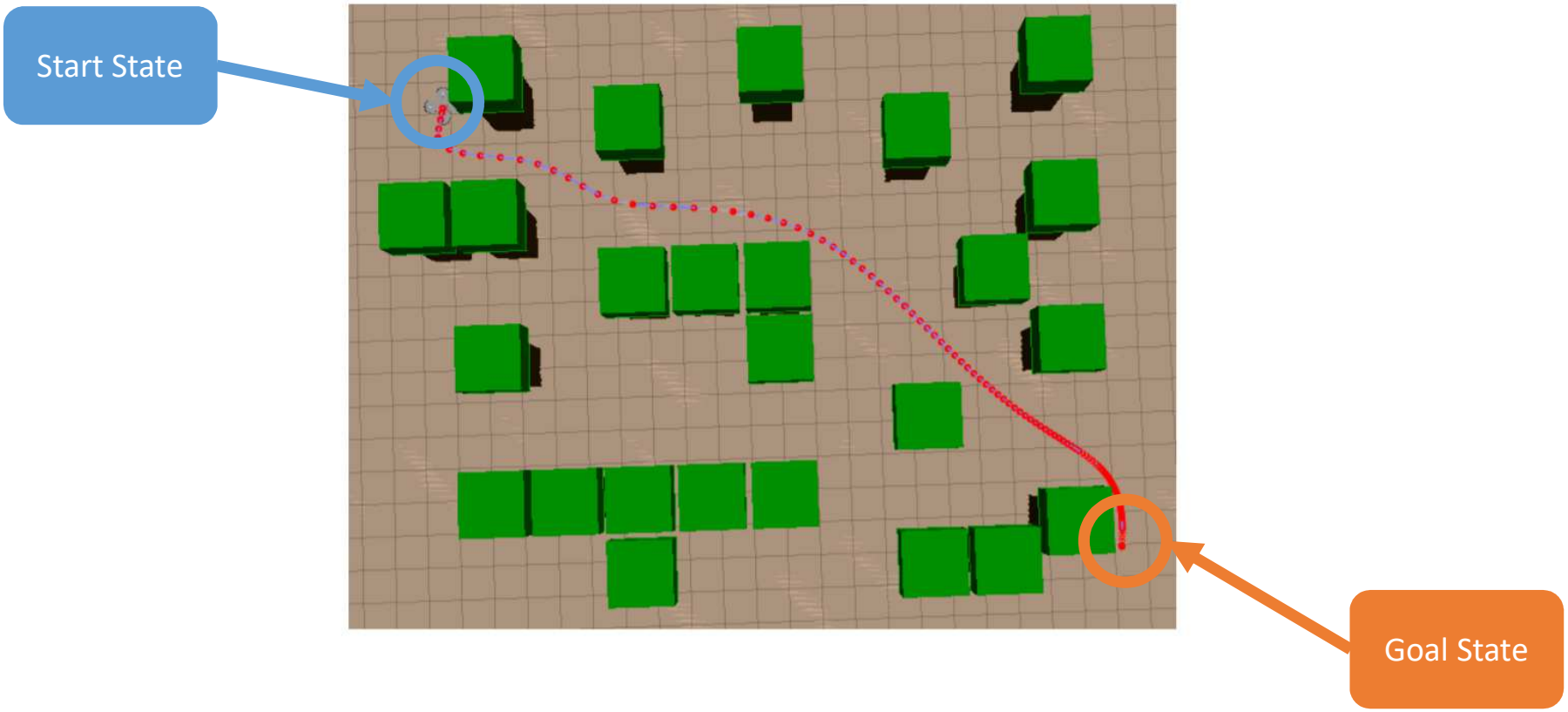
5

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

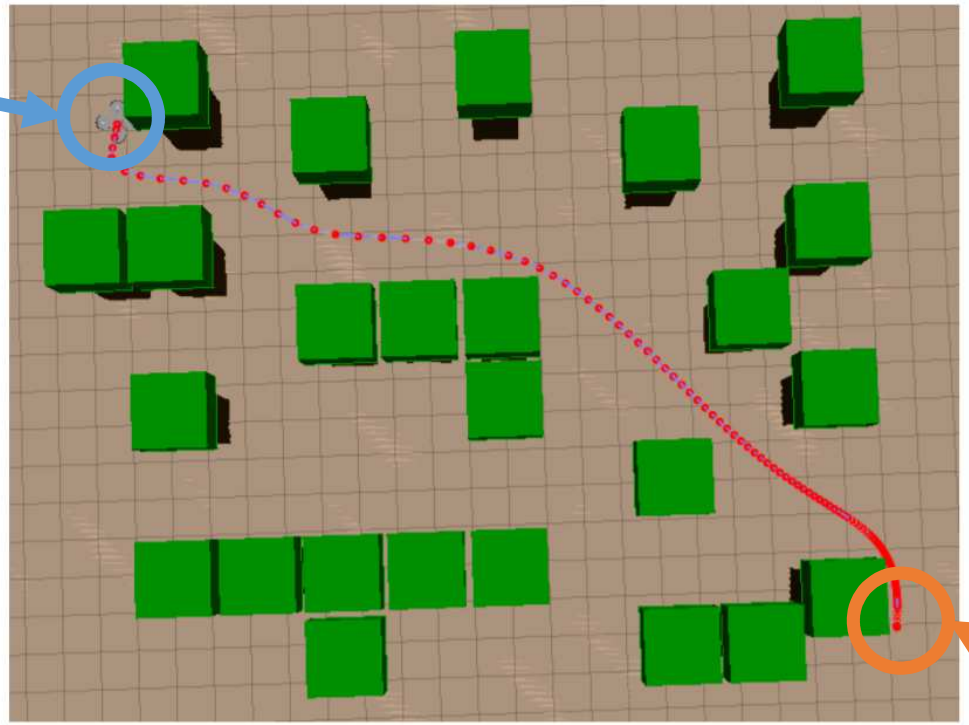


5

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



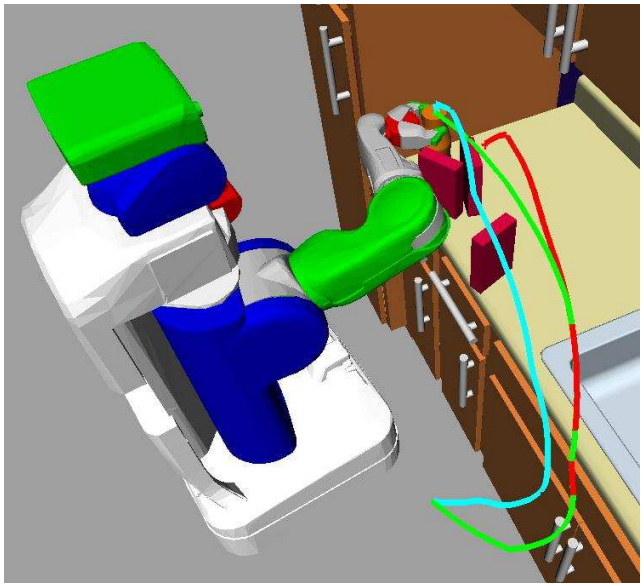
Planning is the process of computing an action plan for a robot based on the previously computed map



Before we can think about how to compute this we need to figure out in what state space are we planning?

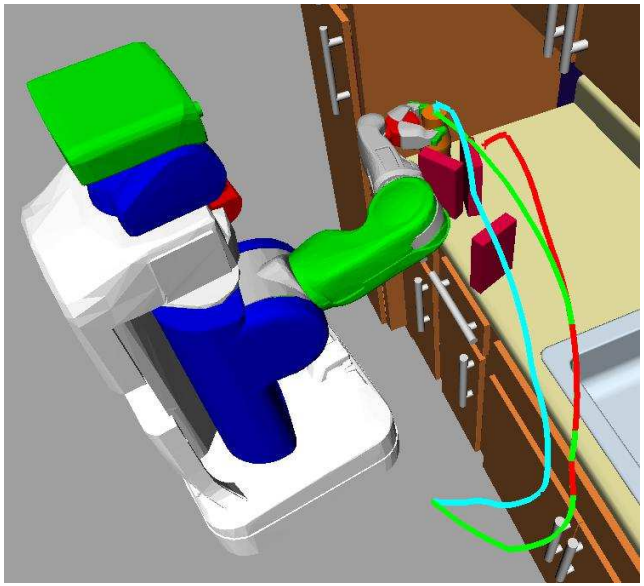
5 Spaces and Transformations (aka where are we planning?)

- Task space: the 6D workspace of the robot
  - E.g., the pose (x,y,z,roll,pitch,yaw) of the robot's hand or an object



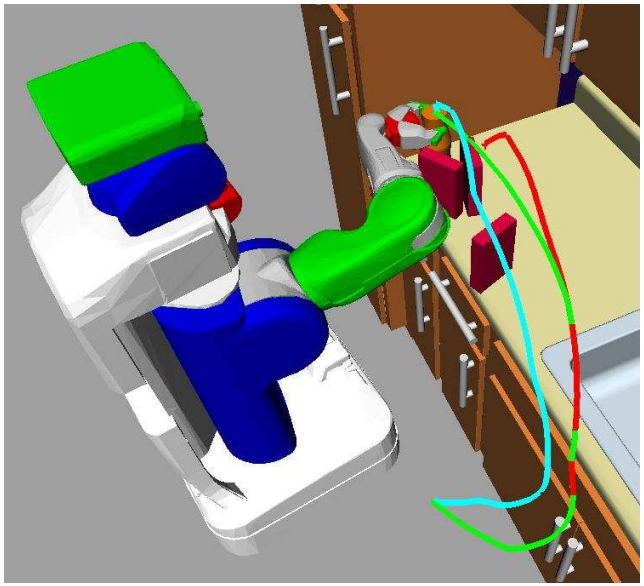
5 Spaces and Transformations (aka where are we planning?)

- Task space: the 6D 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 $q \in \mathbb{R}^n$



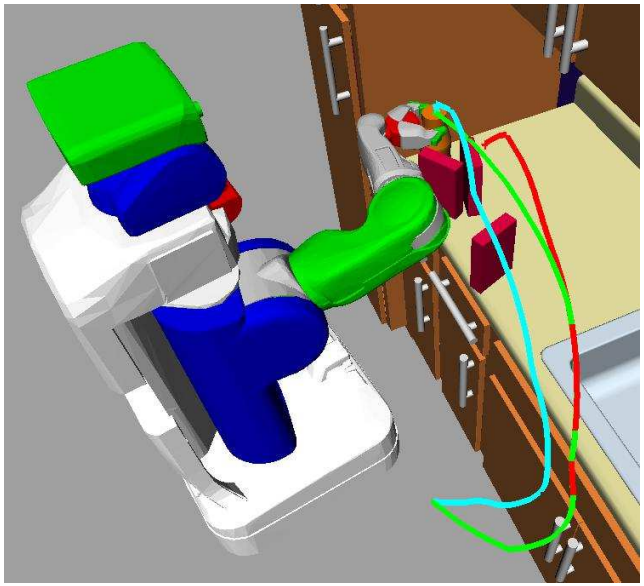
5 Spaces and Transformations (aka where are we planning?)

- Task space: the 6D 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



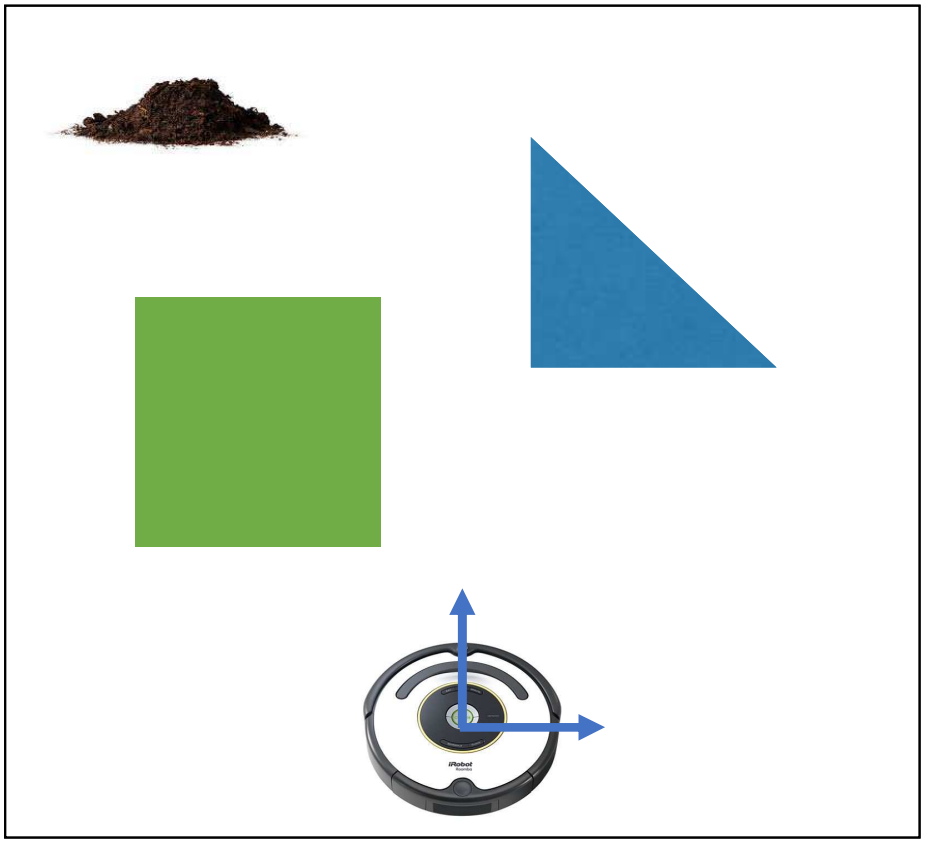
5 Spaces and Transformations (aka where are we planning?)

- Task space: the 6D 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



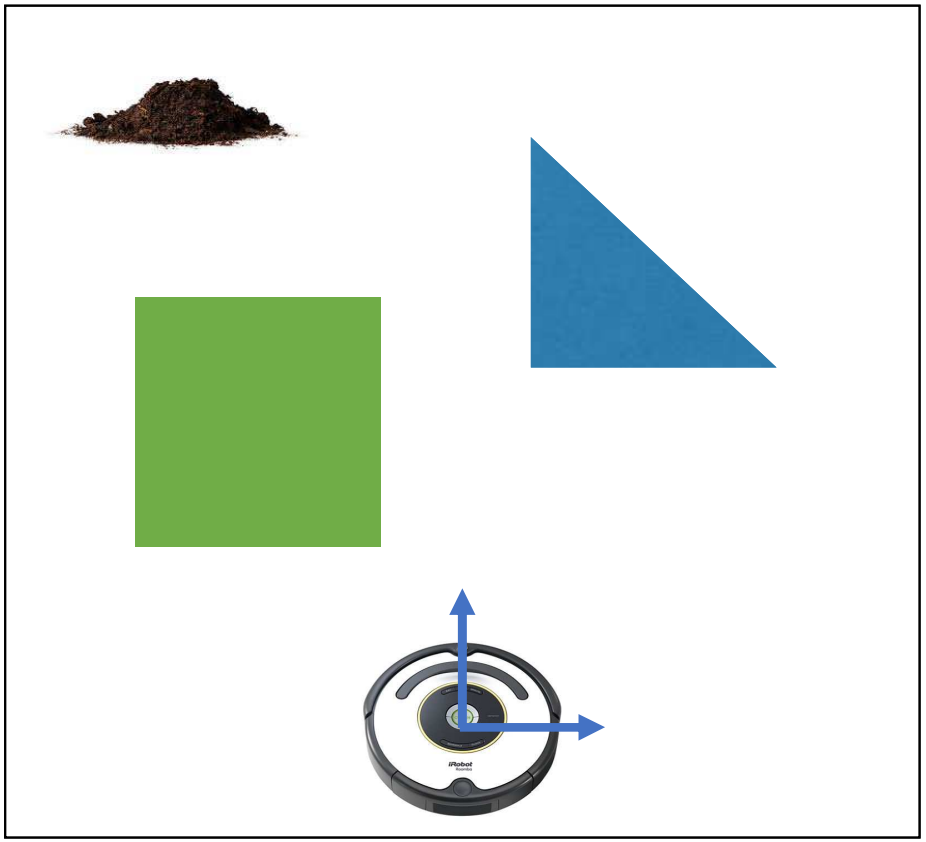
Q: Are forward and inverse kinematics 1 to 1 operations?

5 Configuration Space



Q1: What is the configuration space state for this omnidirectional robot?

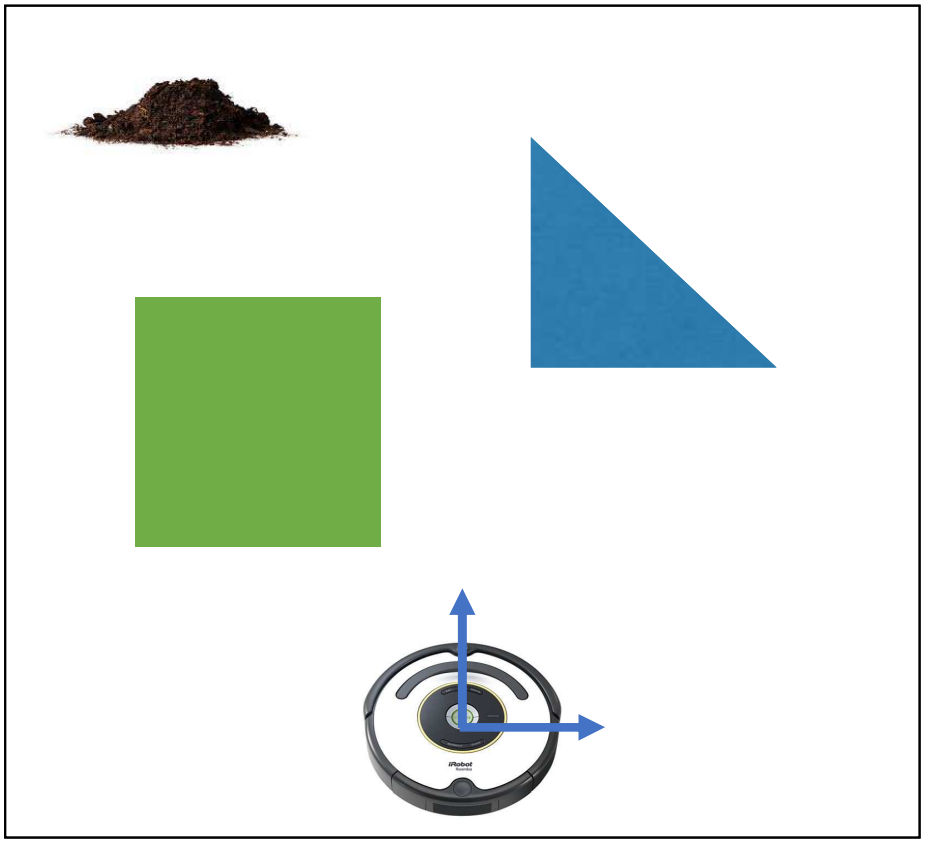
5 Configuration Space



Q1: What is the configuration space state for this omnidirectional robot?

A1: (x,y) position of the center of the robot

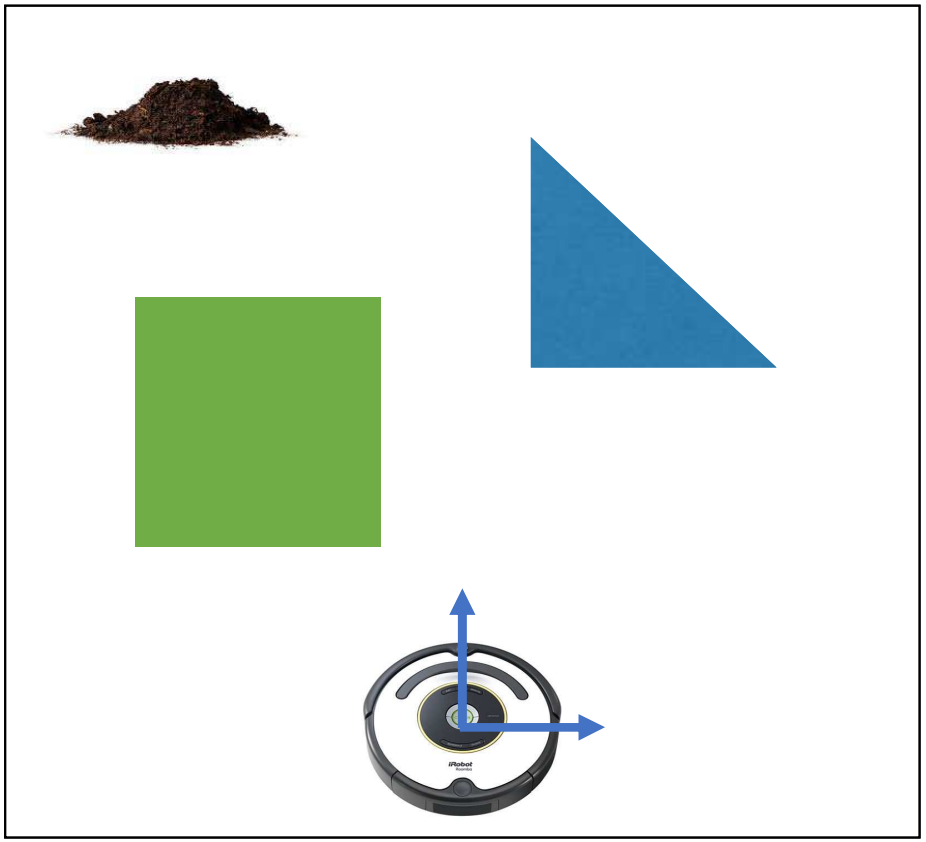
5 Configuration Space

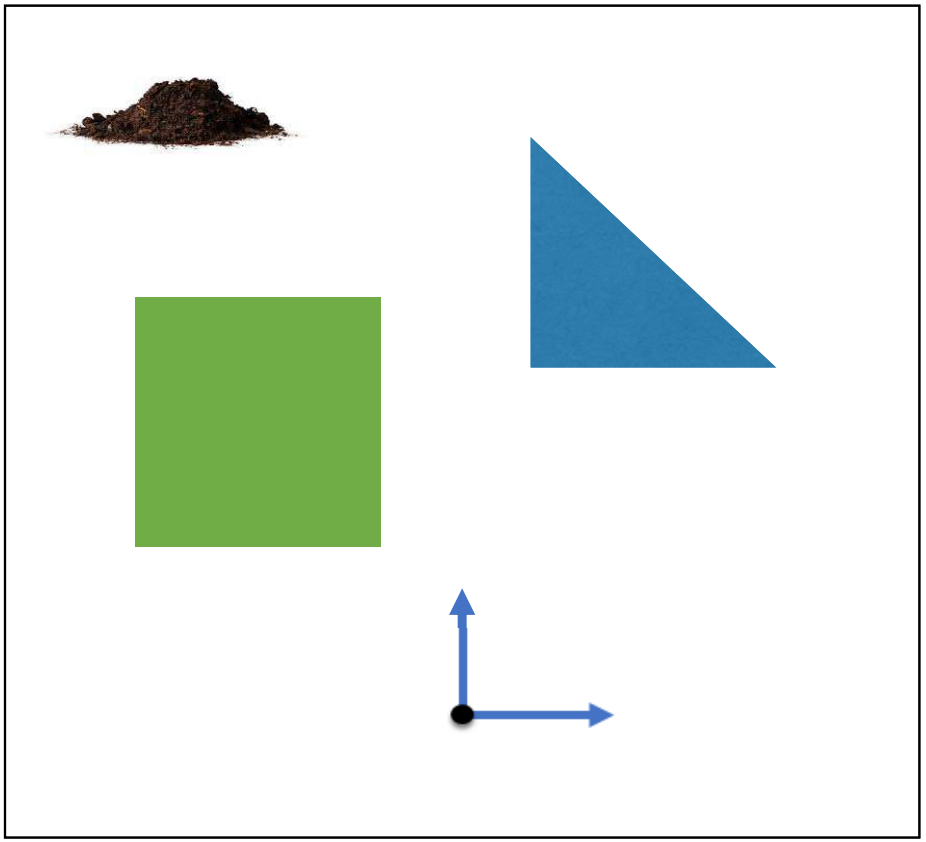


Q2: How can we map this robot's world into configuration space?

5 Configuration Space

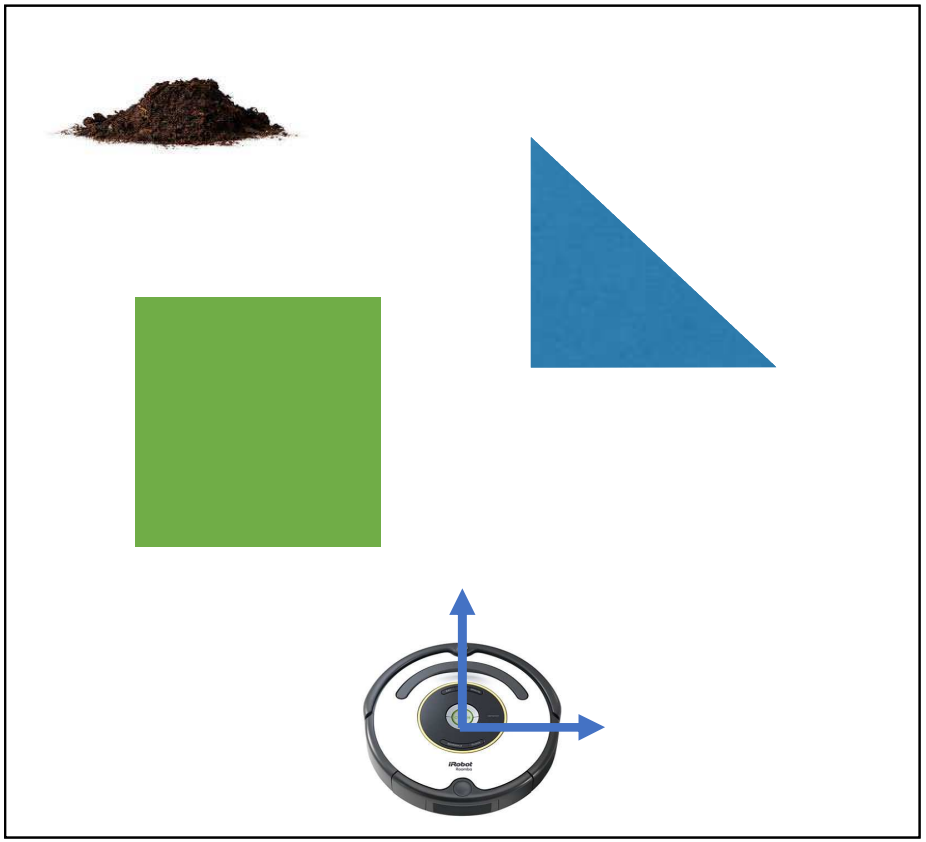
Well we want the robot to become a single (x,y) point

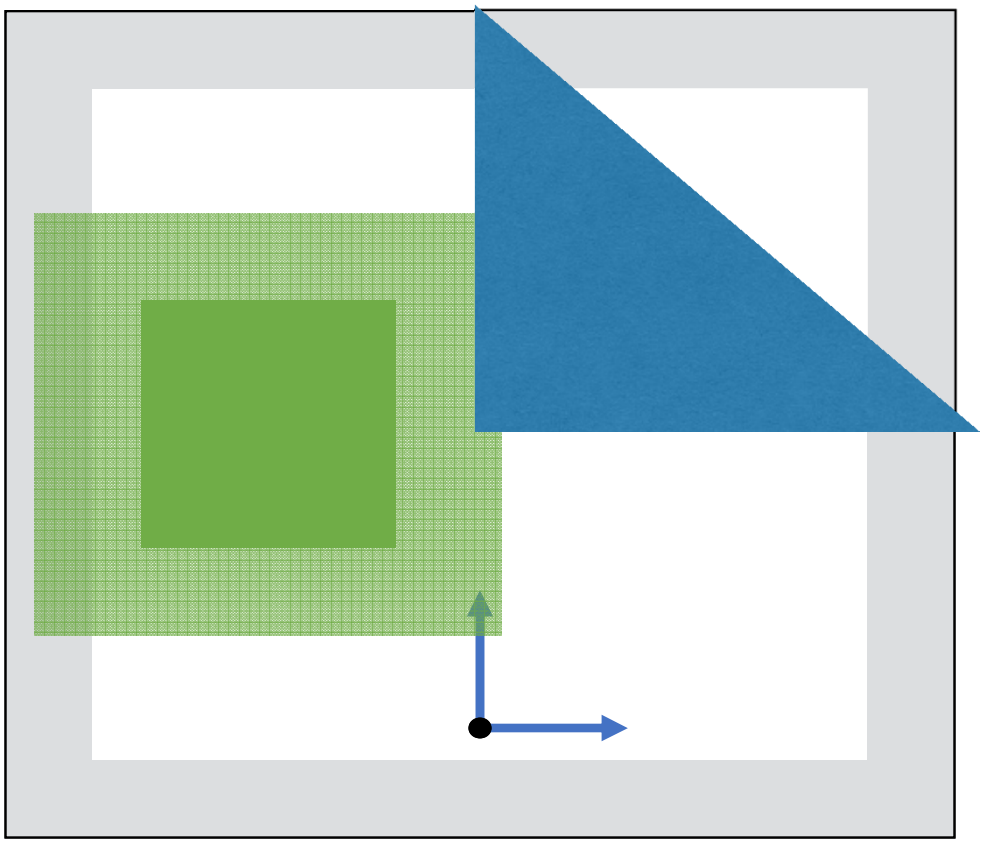




5 Configuration Space

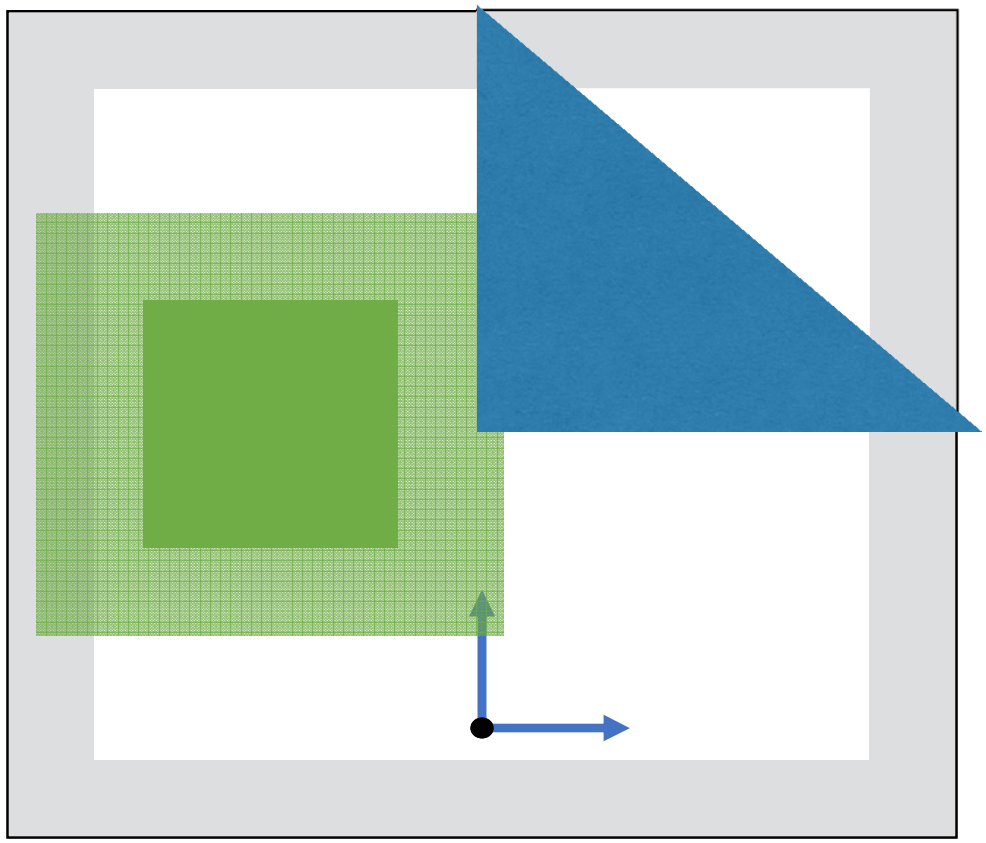
So we need to inflate the obstacles accordinly



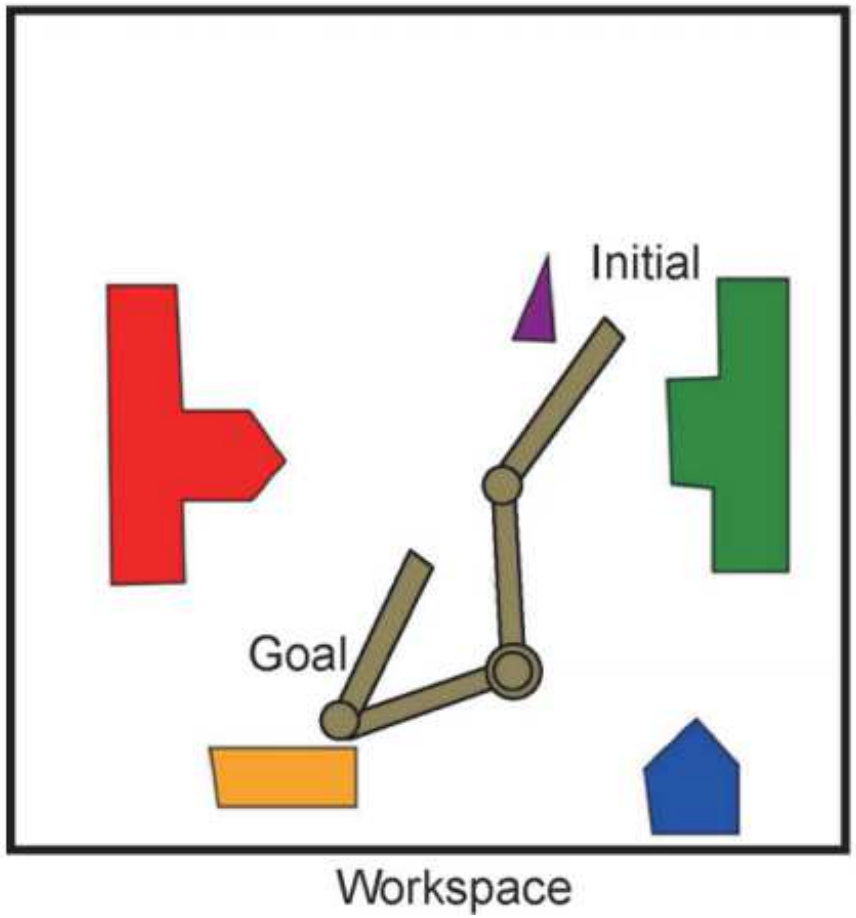


5 Configuration Space

- **Insight:** mapping task space obstacles and goals into configuration space allows us to **plan a path for a single point** instead of worrying about a full robot

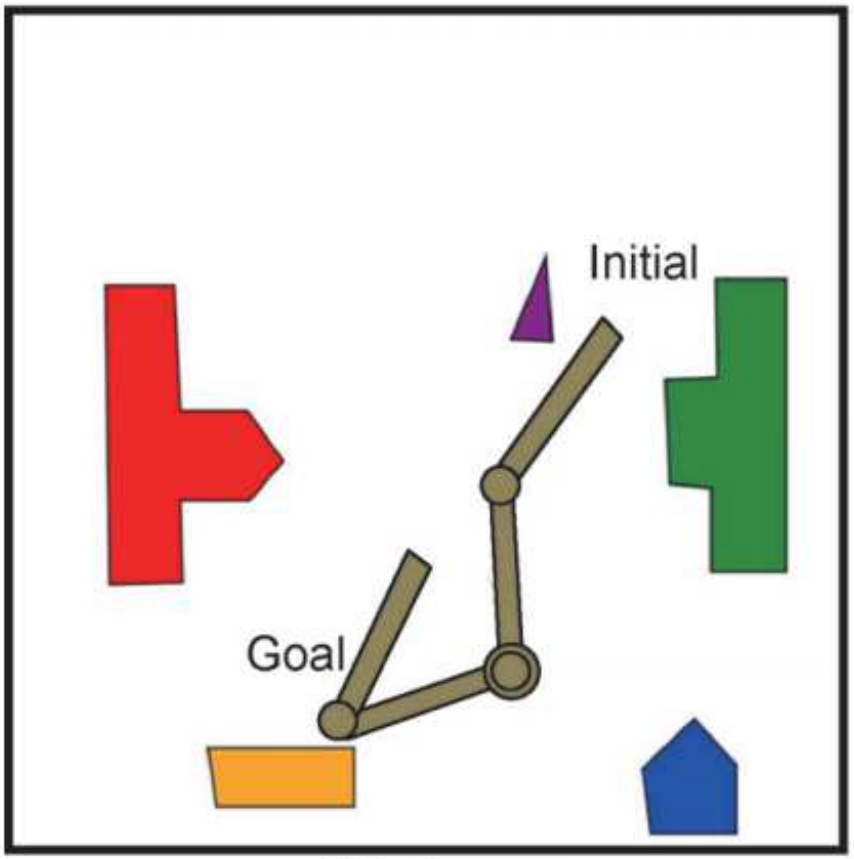


5 Configuration Space

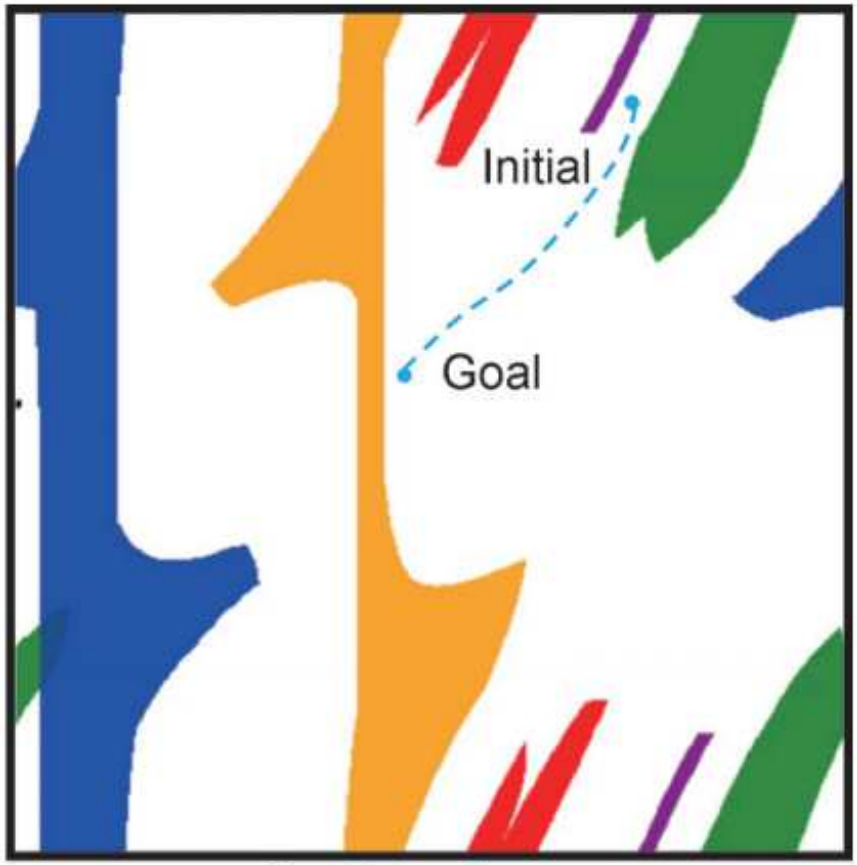


How can we map this robot and its world into configuration space?

5 Configuration Space



Workspace

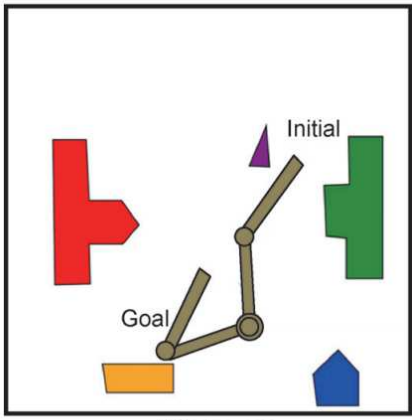


Configuration space

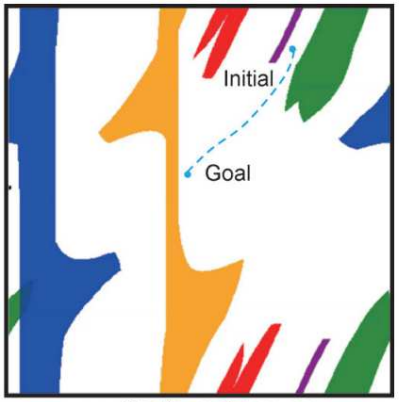
5 How to use configuration space in practice

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

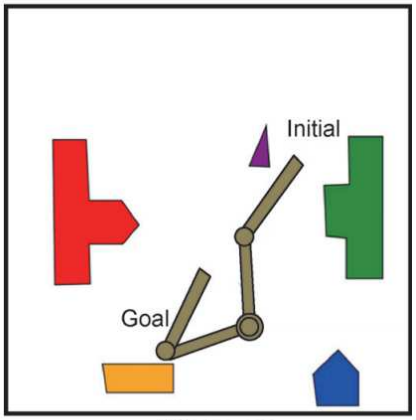
5 How to use configuration space in practice

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

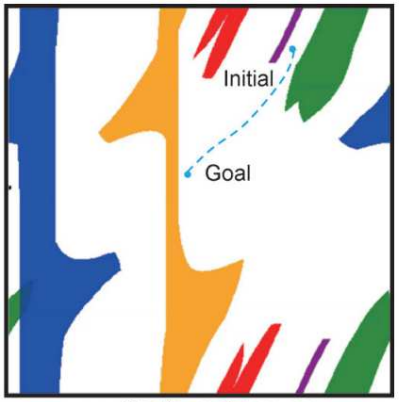
- **Problem:** mapping obstacles into configuration space is hard

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

Treat the collision checker as a black box function evaluator!



Workspace



Configuration space

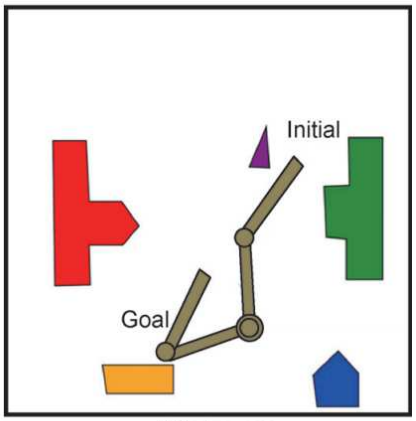
5 How to use configuration space in practice

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

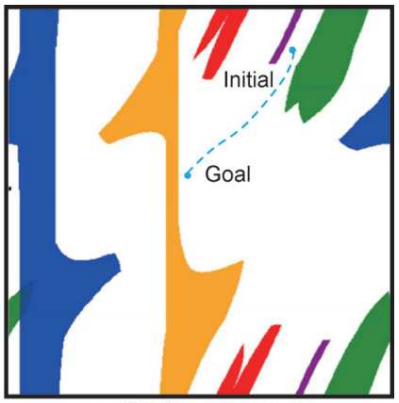
- **Problem:** mapping obstacles into configuration space is hard

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

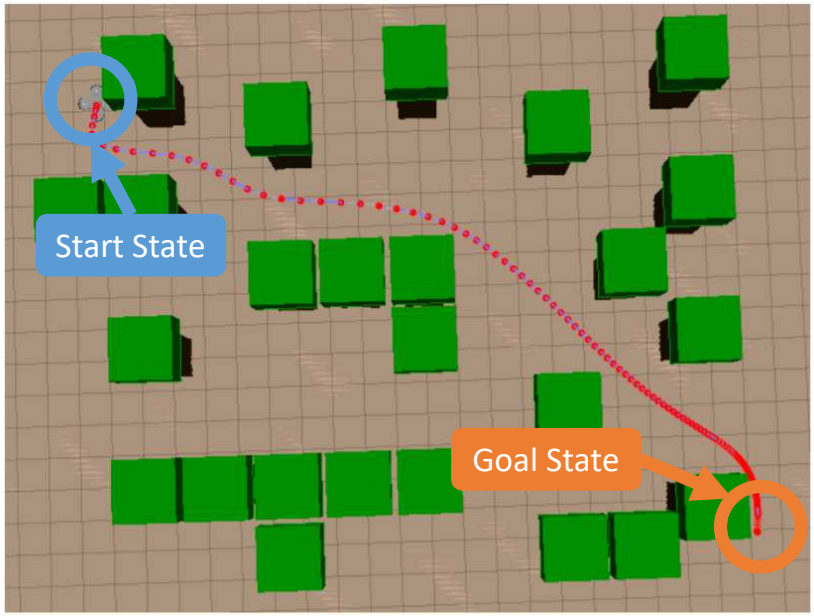


Workspace



Configuration space

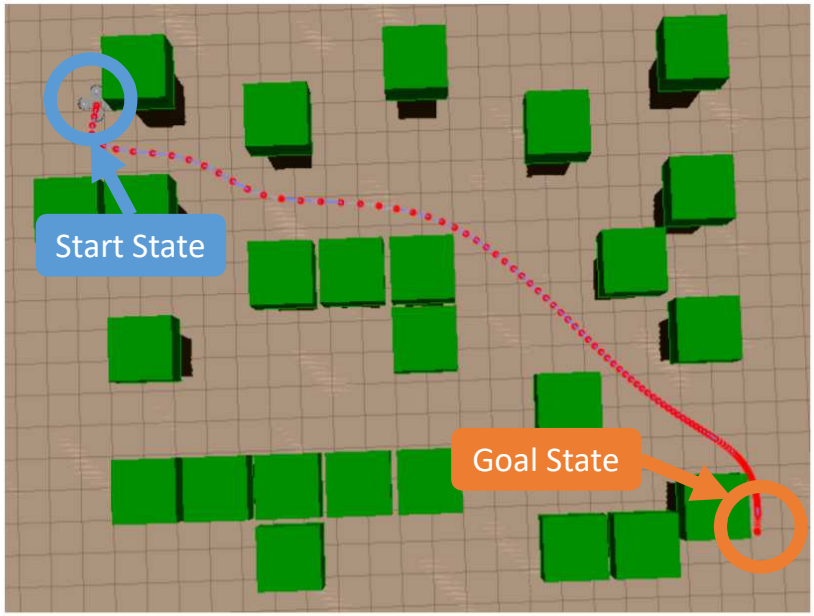
5 Planning in Configuration Space



Goal: Find shortest collision-free path from start to goal

States: configurations $q \in \mathcal{R}^6$ **Actions:** Δq **Transition:** $q' \leftarrow q + \Delta q$

5 Planning in Configuration Space

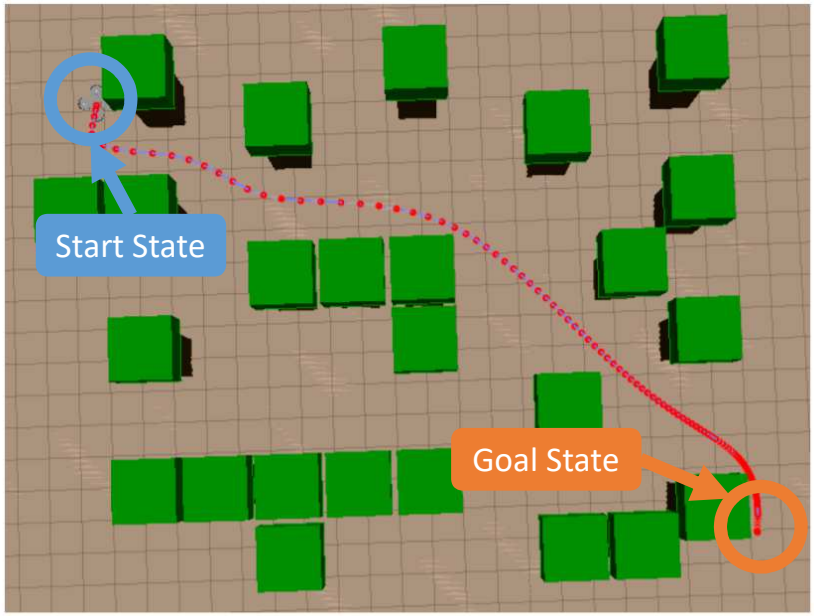


$q \in \mathcal{R}^6$: (x,y,z,θ,ϕ,φ)

Goal: Find shortest collision-free path from start to goal

States: configurations $q \in \mathcal{R}^6$ **Actions:** Δq **Transition:** $q' \leftarrow q + \Delta q$

5 Planning in Configuration Space

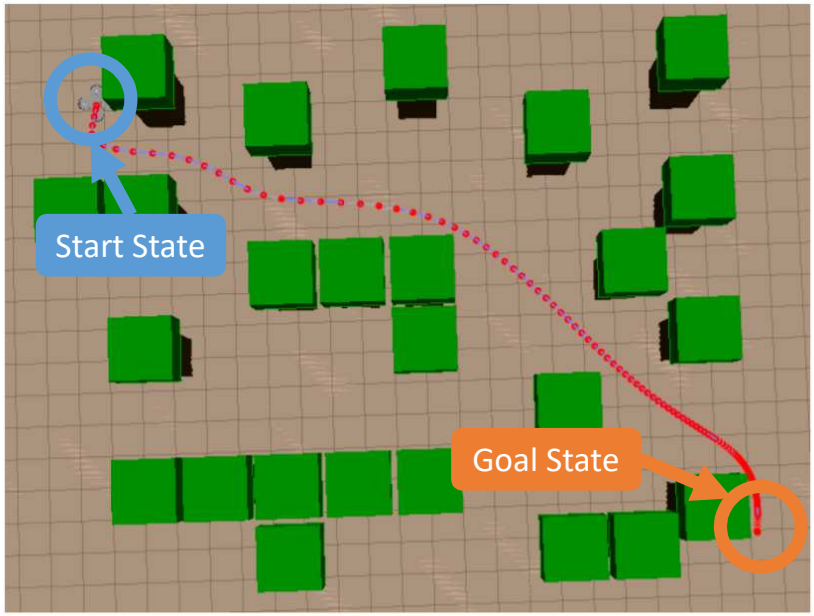


One approach is to ***discretize*** the statespace (grid it) and use graph search (think A* which is known fast)

Goal: Find shortest collision-free path from start to goal

States: configurations $q \in \mathcal{R}^6$ **Actions:** Δq **Transition:** $q' \leftarrow q + \Delta q$

5 Planning in Configuration Space



One approach is to ***discretize*** the statespace (grid it) and use graph search (think A* which is known fast)

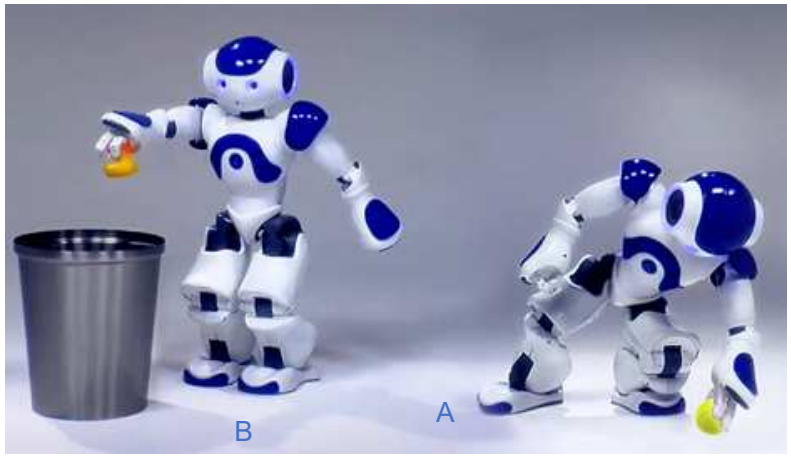
Unfortunately if we use say 100 discrete steps in each direction we get:

$$|S| = 100^6$$

Goal: Find shortest collision-free path from start to goal

States: configurations $q \in \mathcal{R}^6$ **Actions:** Δq **Transition:** $q' \leftarrow q + \Delta q$

5 Planning in Configuration Space



One approach is to ***discretize*** the statespace (grid it) and use graph search (think A* which is known fast)

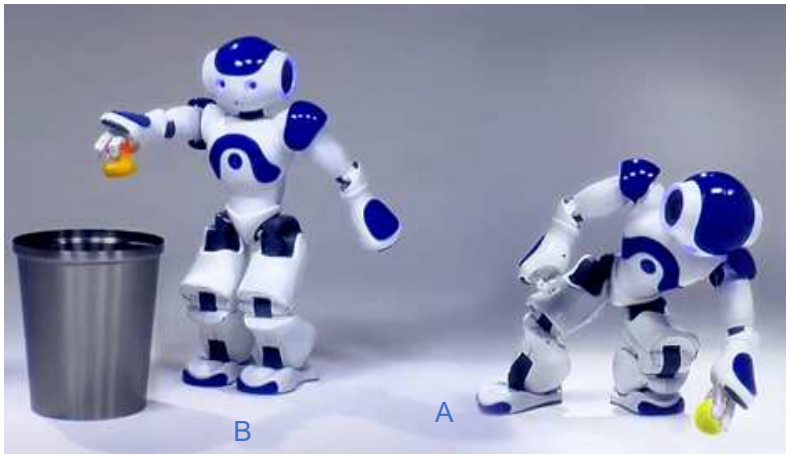
Unfortunately if we use say 100 discrete steps in each direction we get:

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

Goal: Find shortest collision-free path fr

States: configurations $q \in \mathcal{R}^{20}$ **Actions:** Δ

5 Planning in Configuration Space



One approach is to ***discretize*** the statespace (grid it) and use graph search (think A* which is known fast)

Unfortunately if we use say 100 discrete steps in each direction we get:

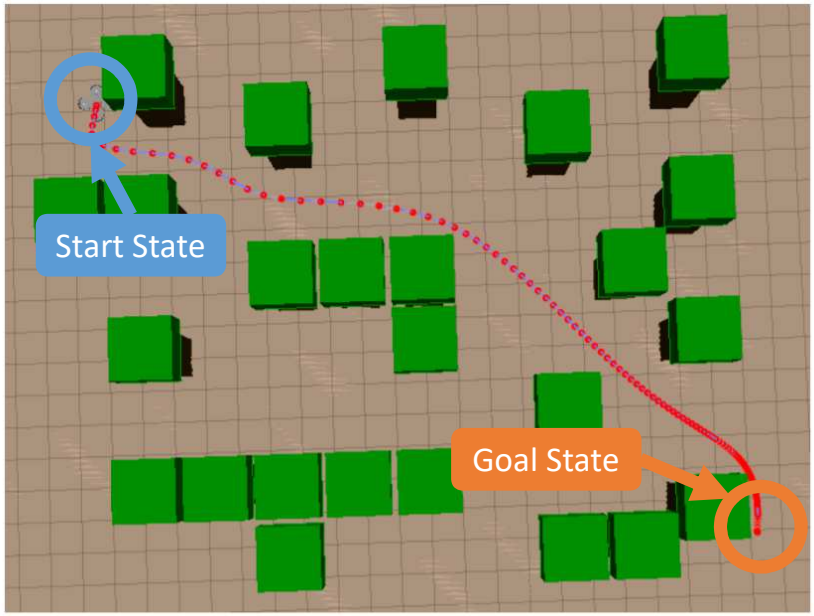
$$|S| = 100^{20}$$

Curse of Dimensionality!

Goal: Find shortest collision-free path from start to goal

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

5 Planning in Configuration Space

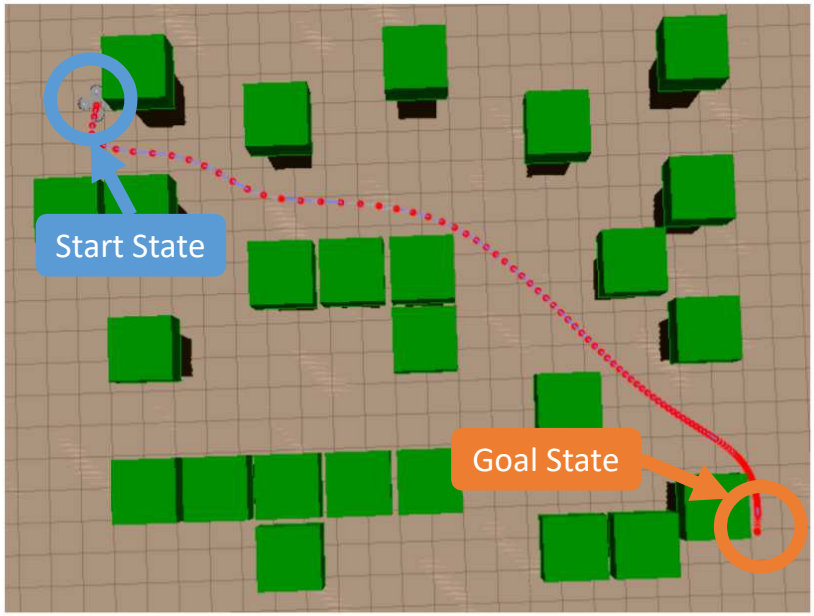


So if we can't explicitly form the graph and search the configuration space what can we do?

Goal: Find shortest collision-free path from start to goal

States: configurations $q \in \mathcal{R}^6$ **Actions:** Δq **Transition:** $q' \leftarrow q + \Delta q$

5 Planning in Configuration Space



What if we **incrementally build up a path** toward the goal?

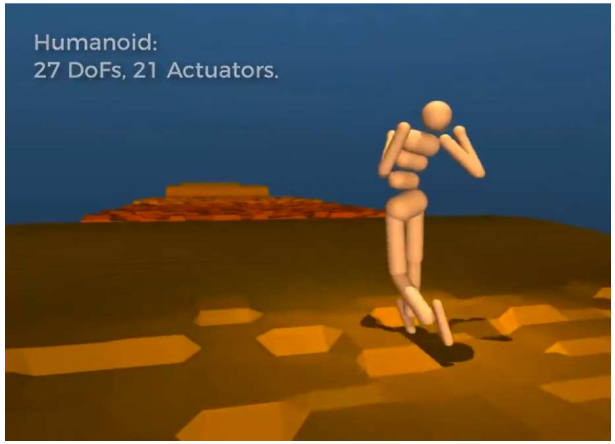
Goal: Find shortest collision-free path from start to goal

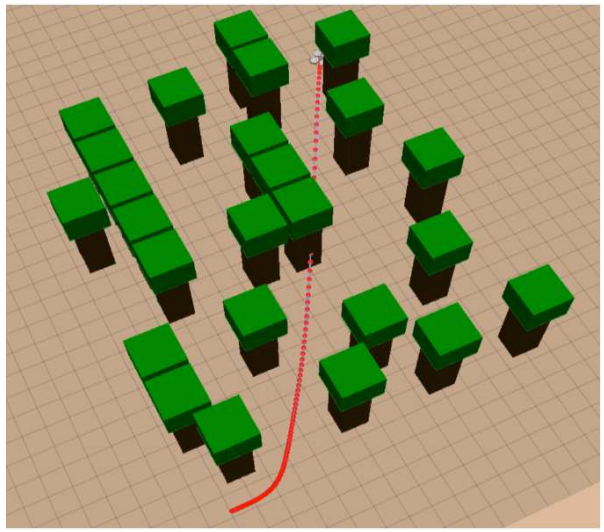
States: configurations $q \in \mathcal{R}^6$ **Actions:** Δq **Transition:** $q' \leftarrow q + \Delta q$

5 Planning in Configuration Space

Random Search

Machine Learning





Local Search

5 Rapidly Exploring Random Trees (RRTs)

One of the most famous robot motion planning algorithms is **Rapidly Exploring Random Trees (RRTs)** [Lavalle & Kuffner]

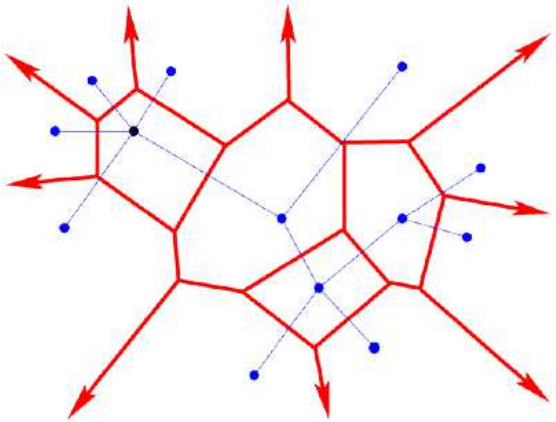
The main idea is to **use randomness** to **rapidly explore** an entire state space to find a path from a given start location to the goal.

5 Randomness encourages exploration

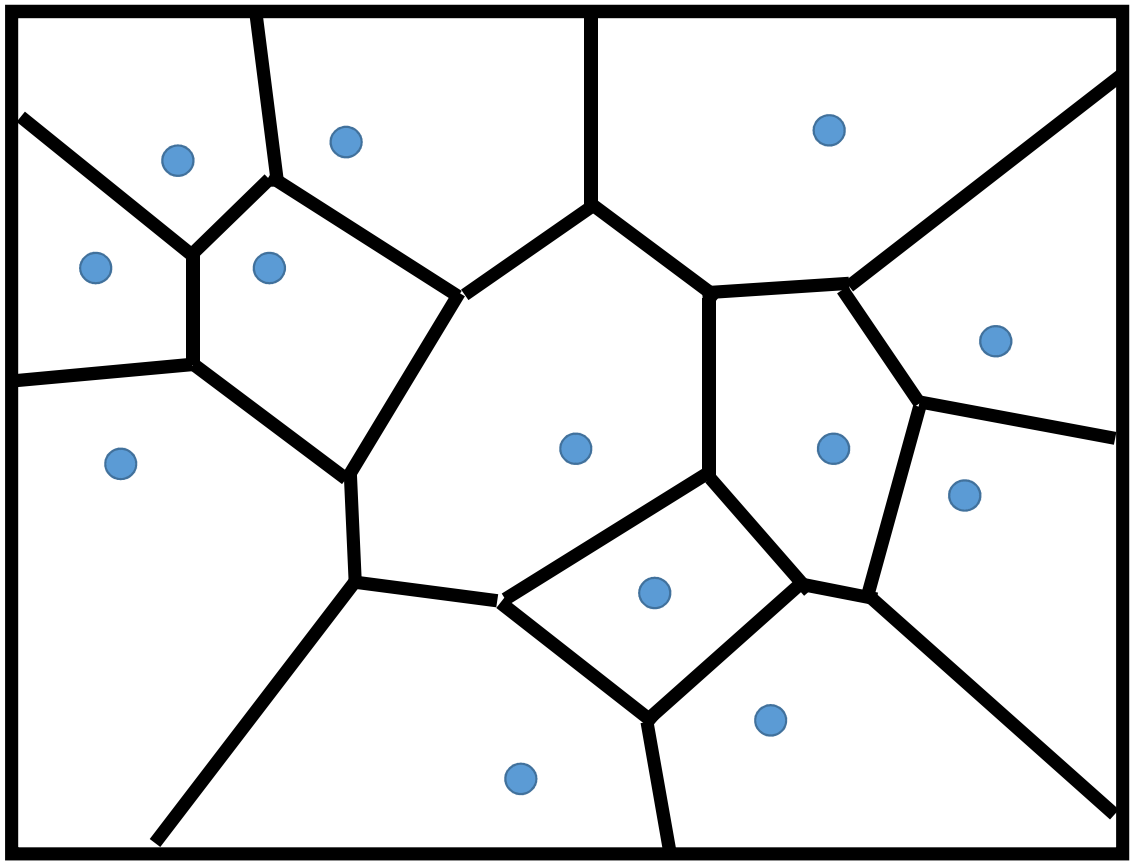
Key idea: uniform random sampling in configuration space is actually a heuristic that encourages exploration!

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

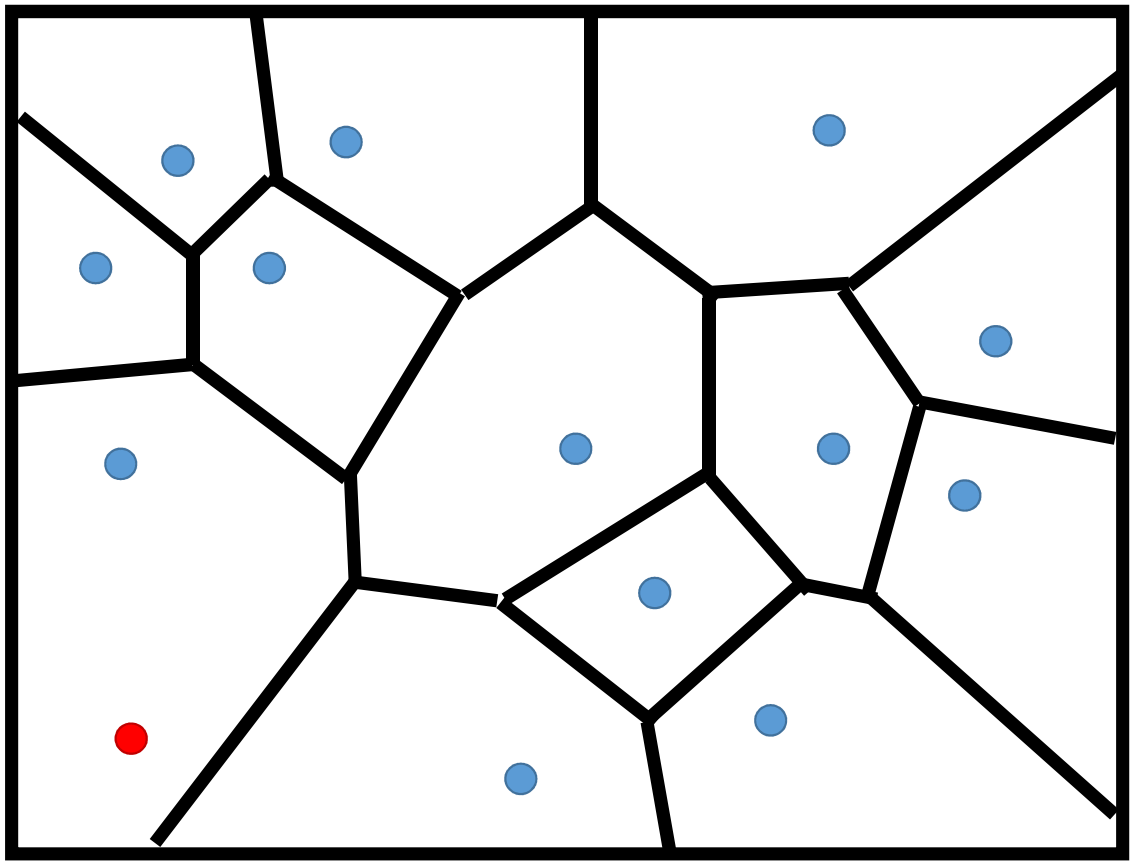
Def: Voronoi region is the set of points in space that are closest to a particular node in the tree:



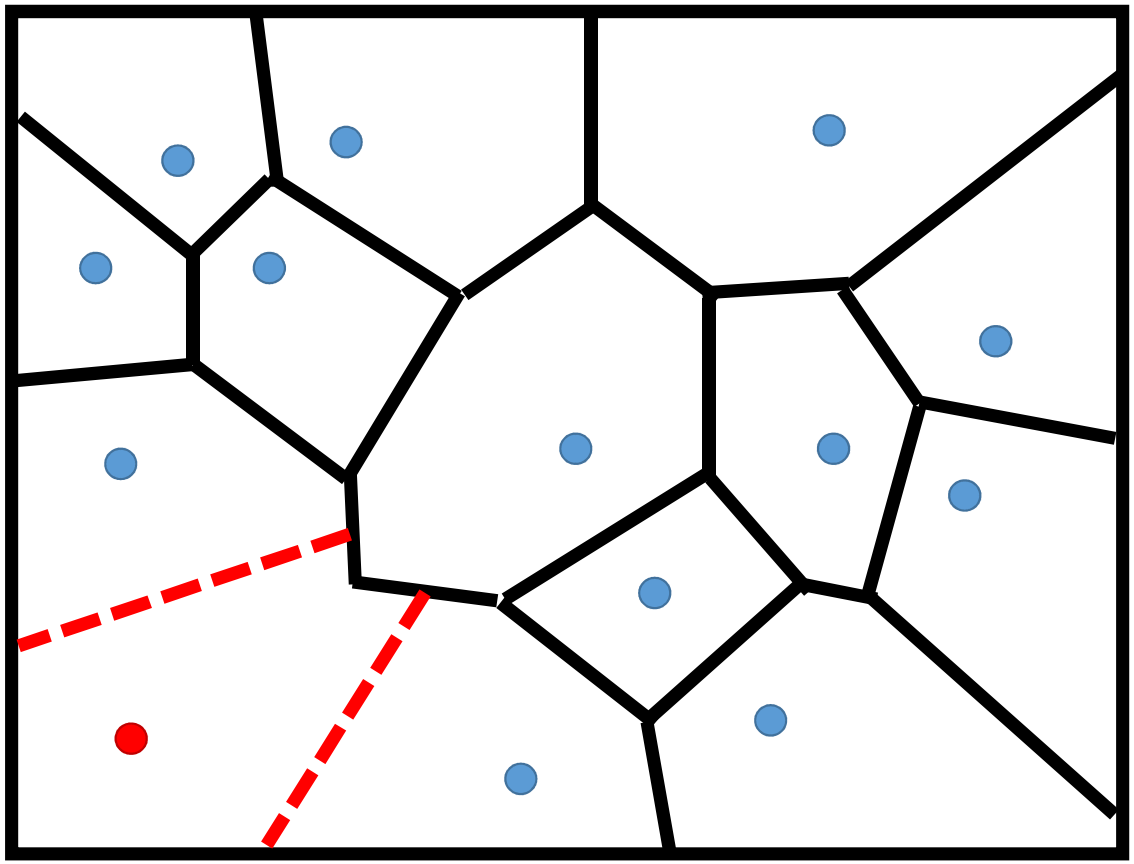
5 Randomness encourages exploration



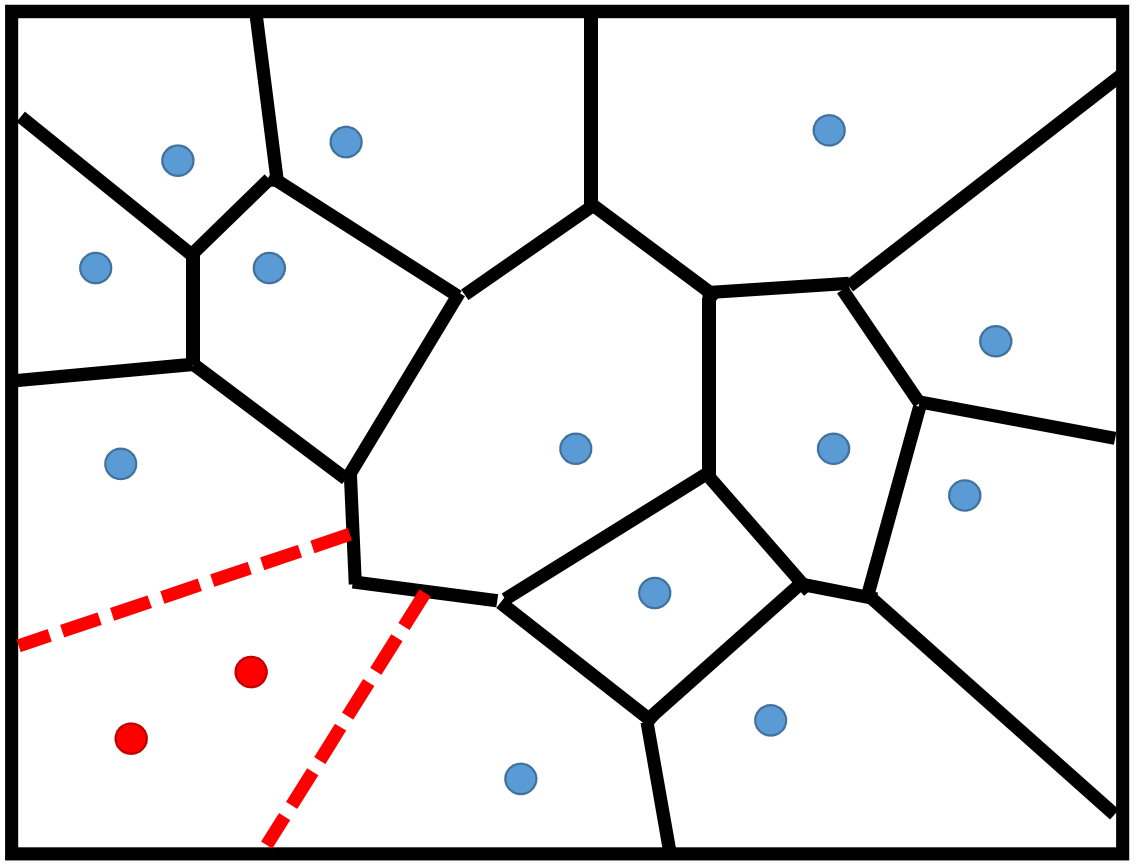
5 Randomness encourages exploration



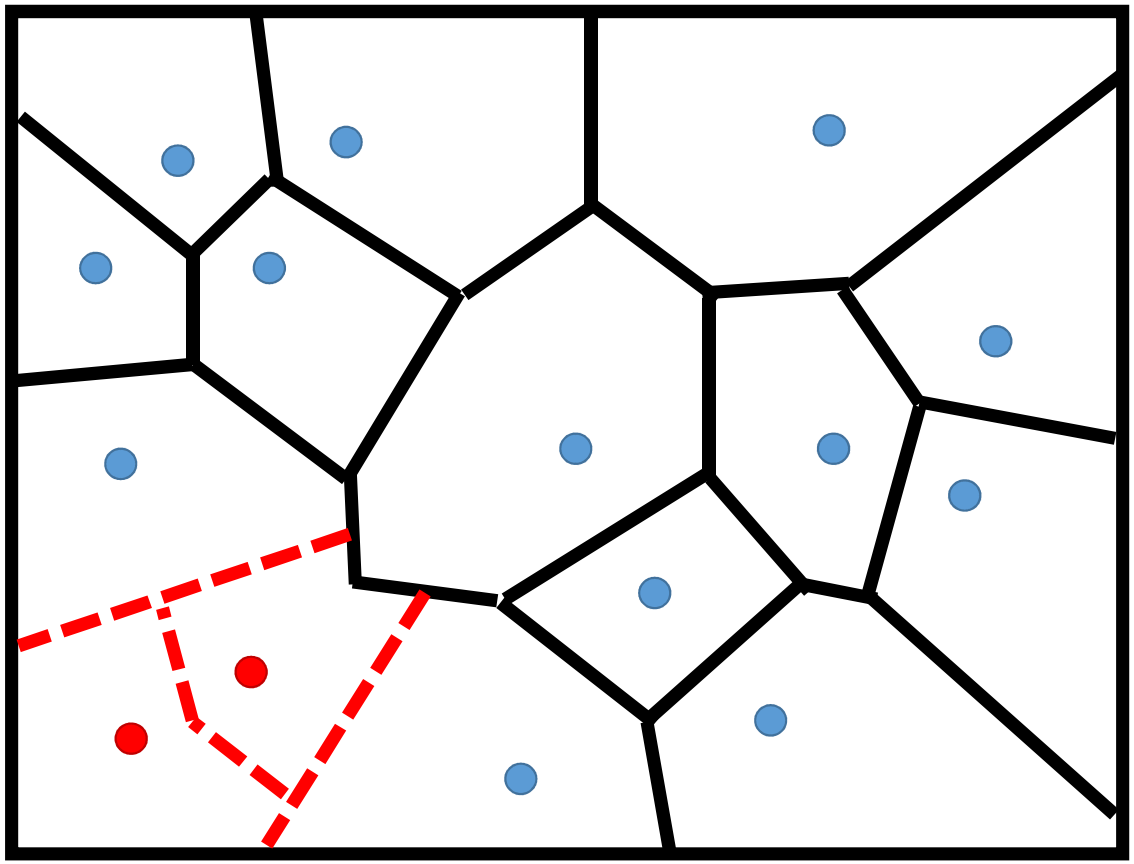
5 Randomness encourages exploration



5 Randomness encourages exploration



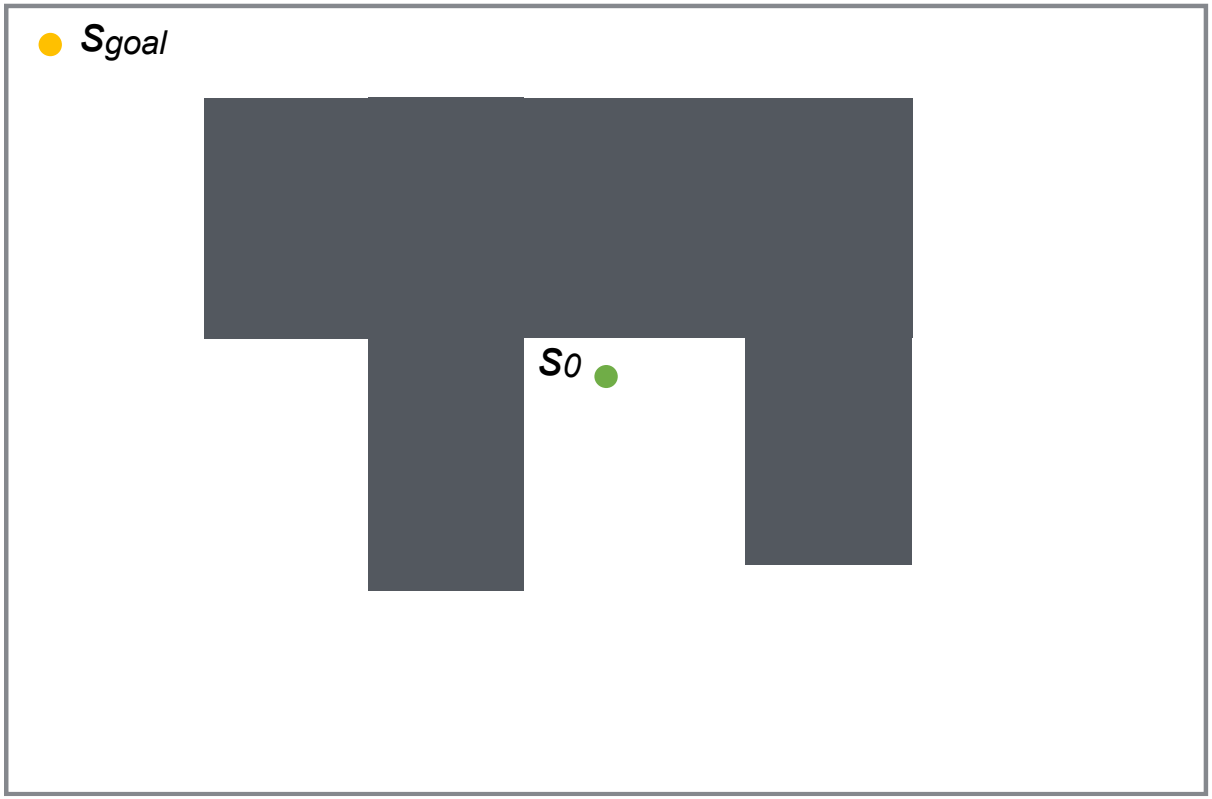
5 Randomness encourages exploration



5 Rapidly Exploring Random Trees (RRTs)

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

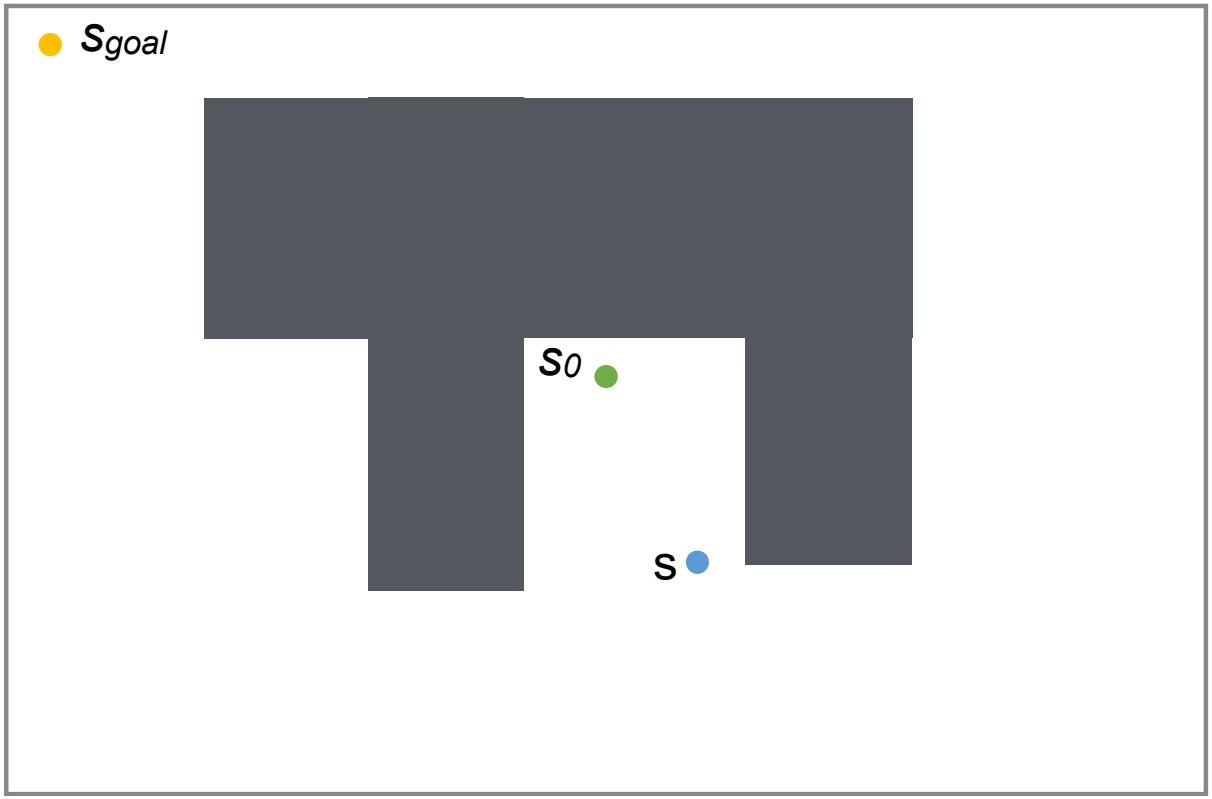
- Sample states $s \in S = R^n$ until s is collision-free
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



5 Rapidly Exploring Random Trees (RRTs)

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

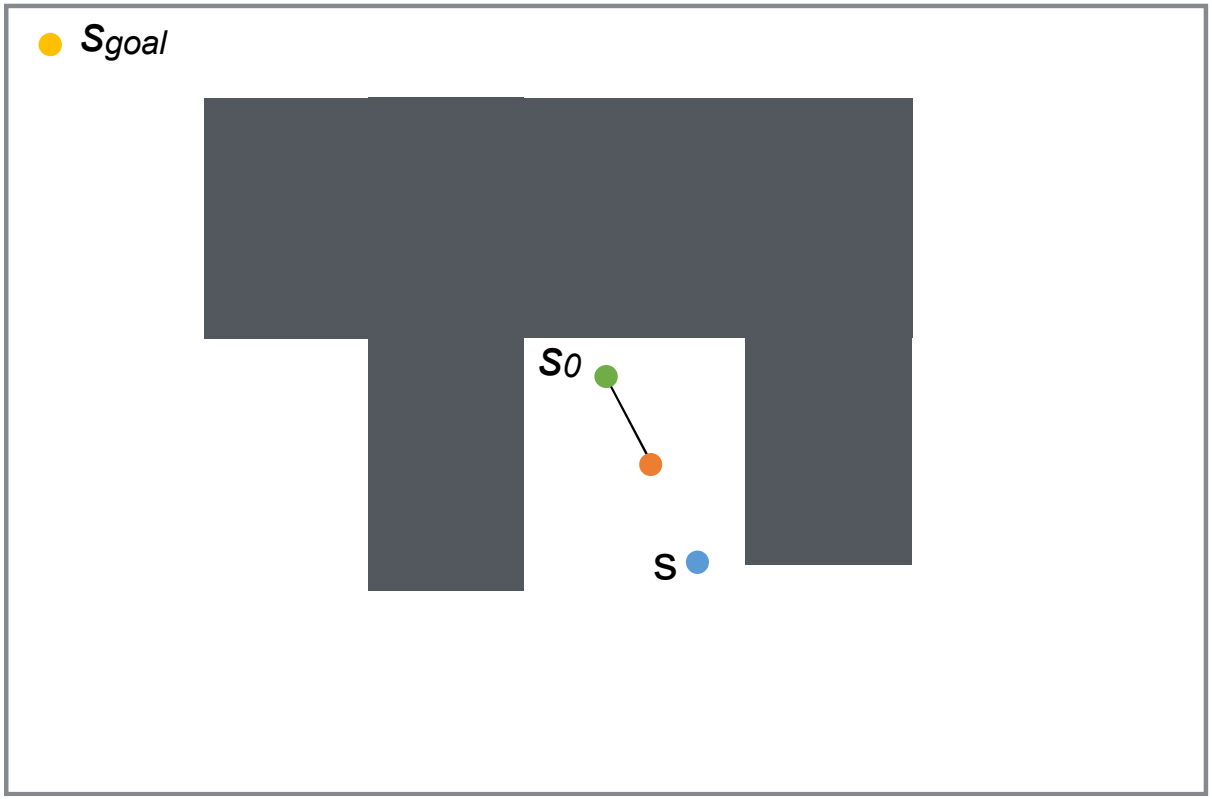
- **Sample states $s \in S = R^n$ until s is collision-free**
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



5 Rapidly Exploring Random Trees (RRTs)

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

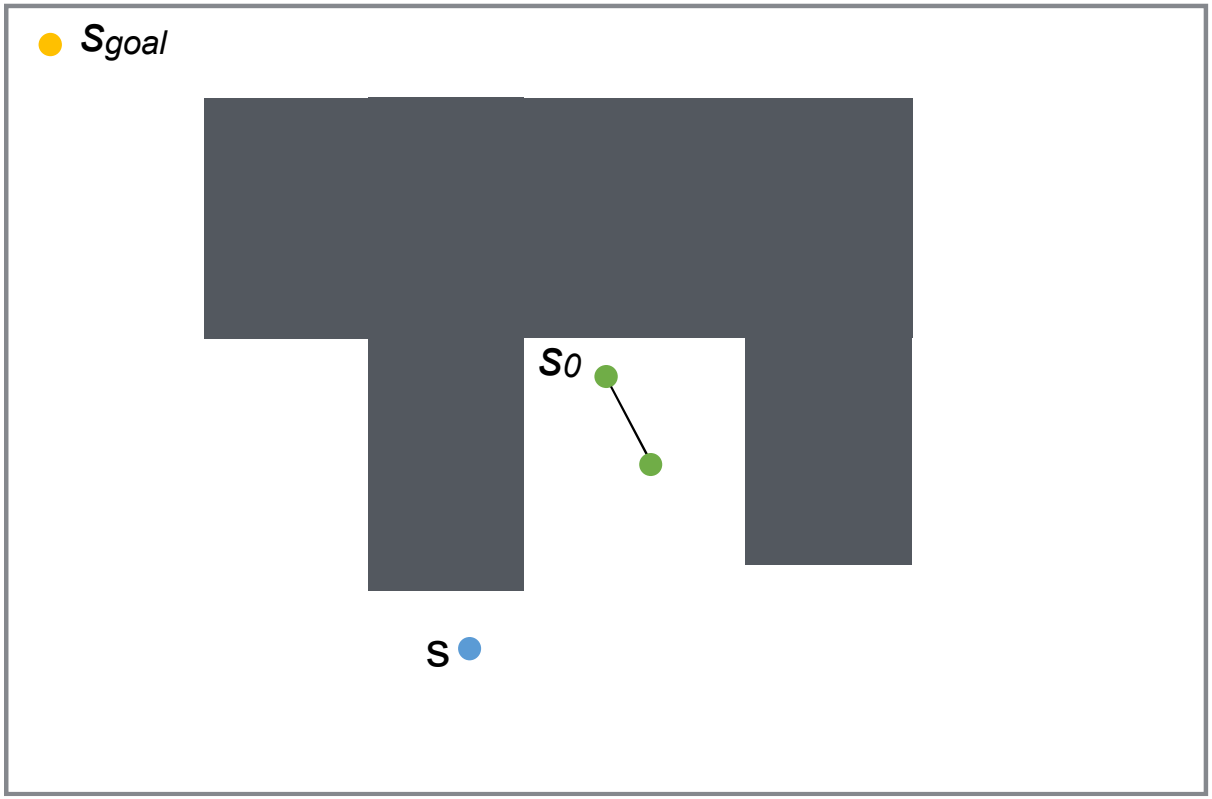
- Sample states $s \in S = R^n$ until s is collision-free
- **Find closest state $s_c \in T$**
- **Extend s_c toward s**
- **Add resulting state s' to T**
- Repeat until T contains a path from s_0 to s_{goal}



5 Rapidly Exploring Random Trees (RRTs)

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

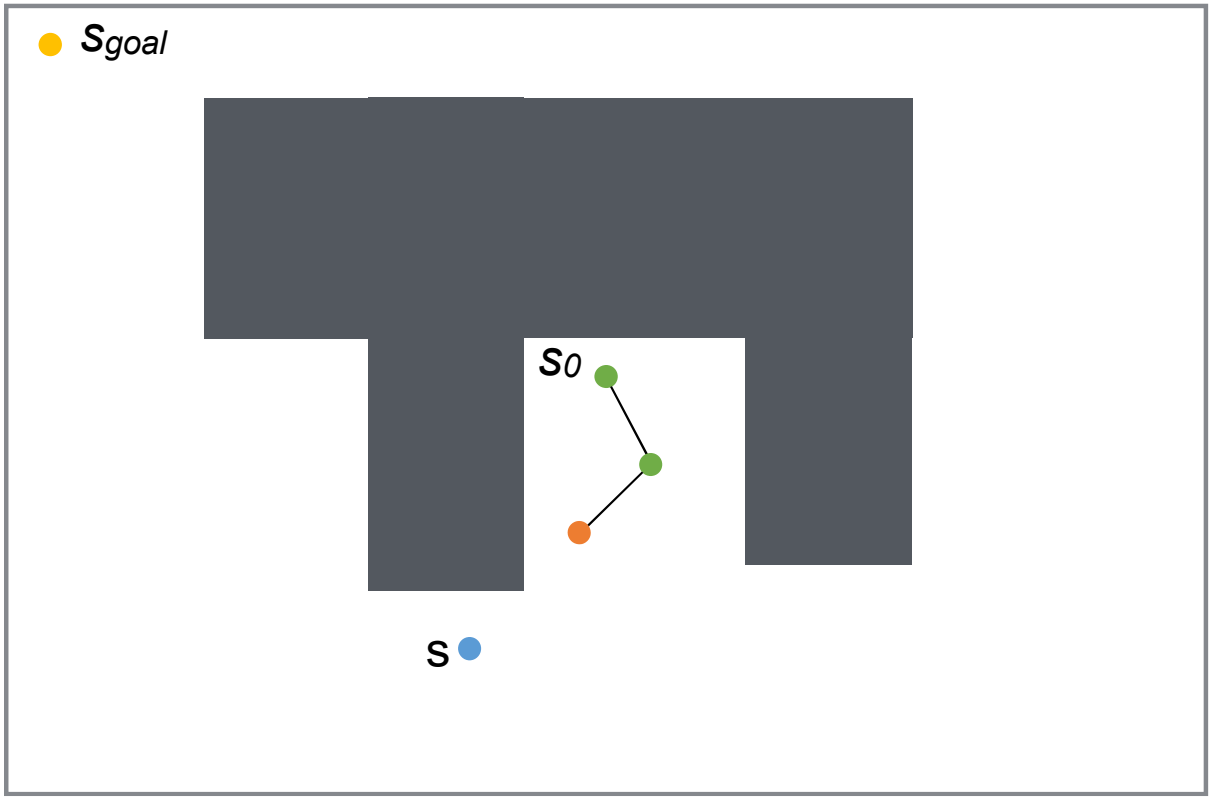
- **Sample states $s \in S = R^n$ until s is collision-free**
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



5 Rapidly Exploring Random Trees (RRTs)

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

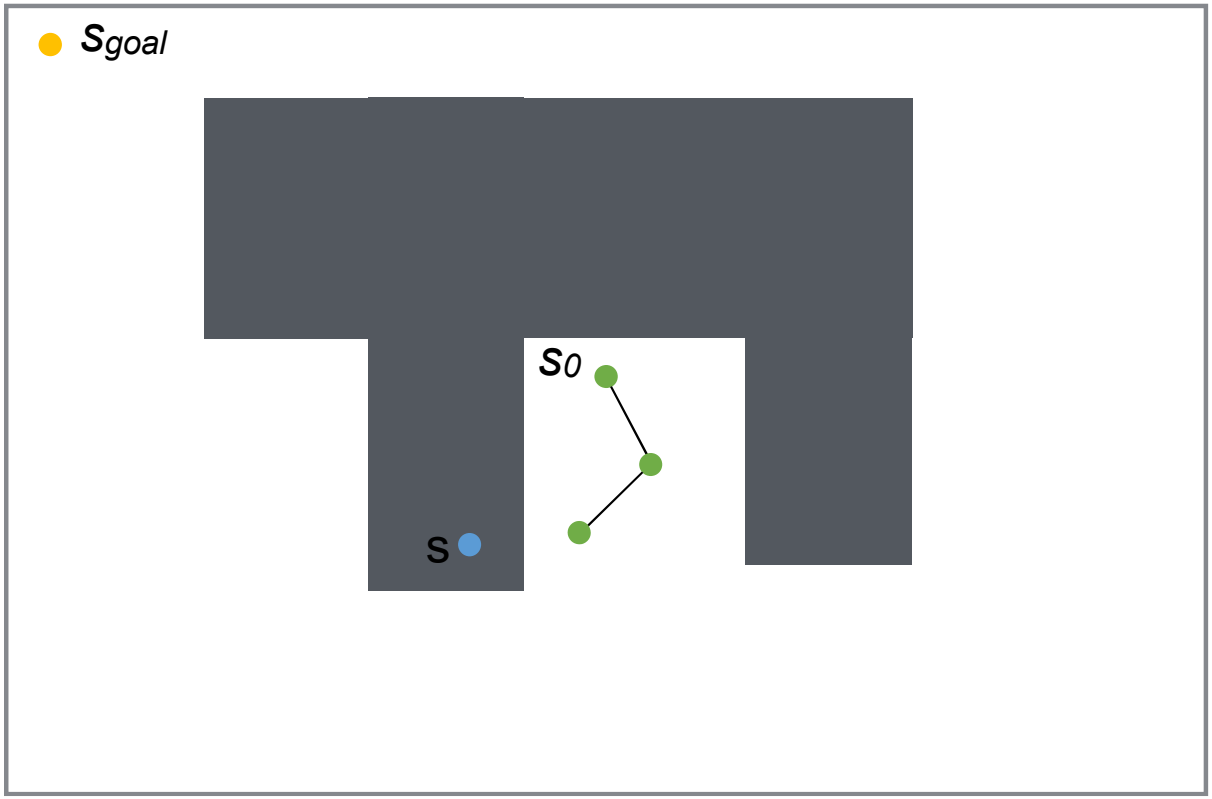
- Sample states $s \in S = R^n$ until s is collision-free
- **Find closest state $s_c \in T$**
- **Extend s_c toward s**
- **Add resulting state s' to T**
- Repeat until T contains a path from s_0 to s_{goal}



5 Rapidly Exploring Random Trees (RRTs)

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

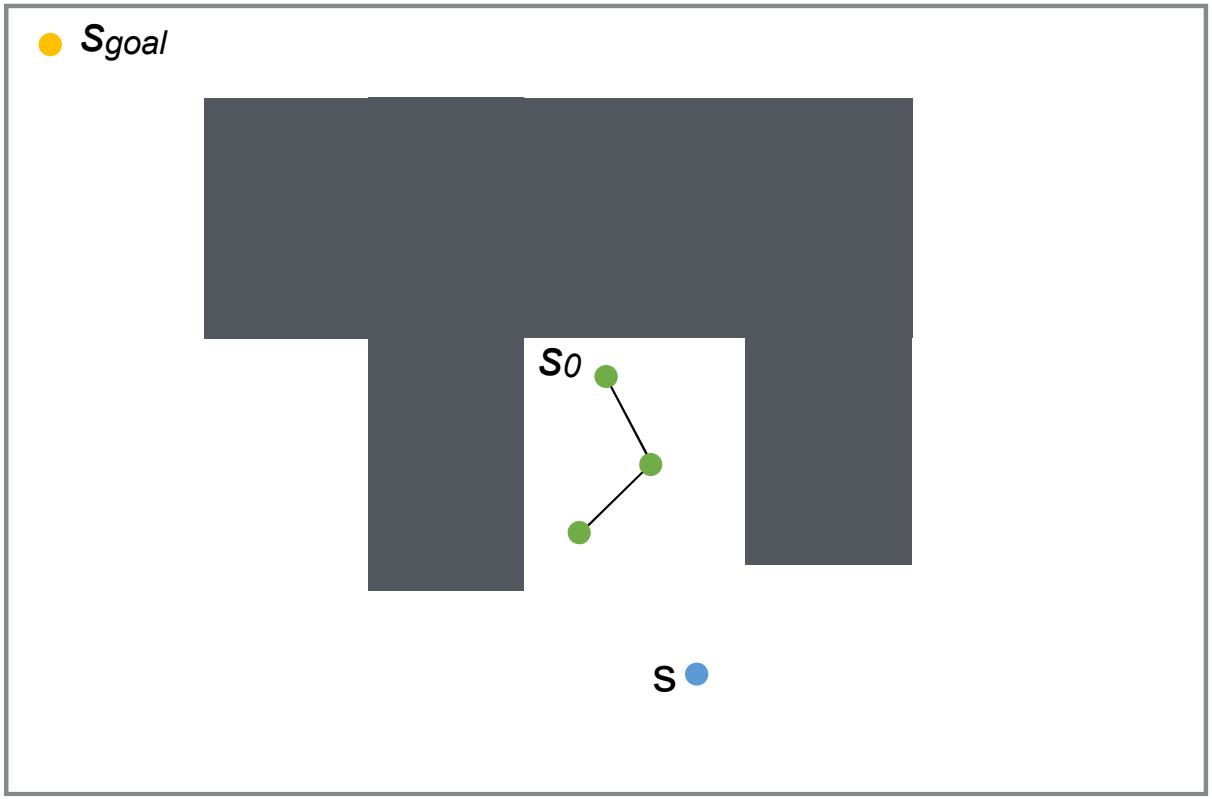
- **Sample states $s \in S = R^n$ until s is collision-free**
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



5 Rapidly Exploring Random Trees (RRTs)

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

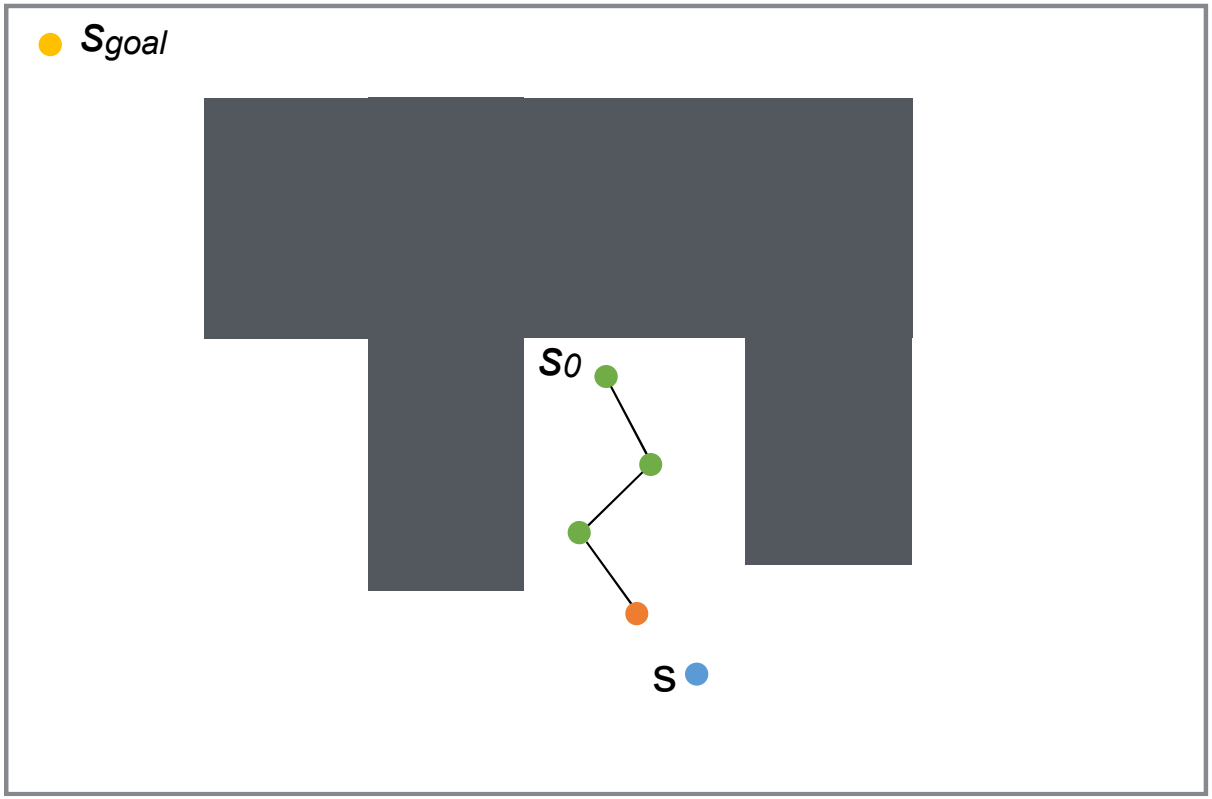
- **Sample states $s \in S = R^n$ until s is collision-free**
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



5 Rapidly Exploring Random Trees (RRTs)

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

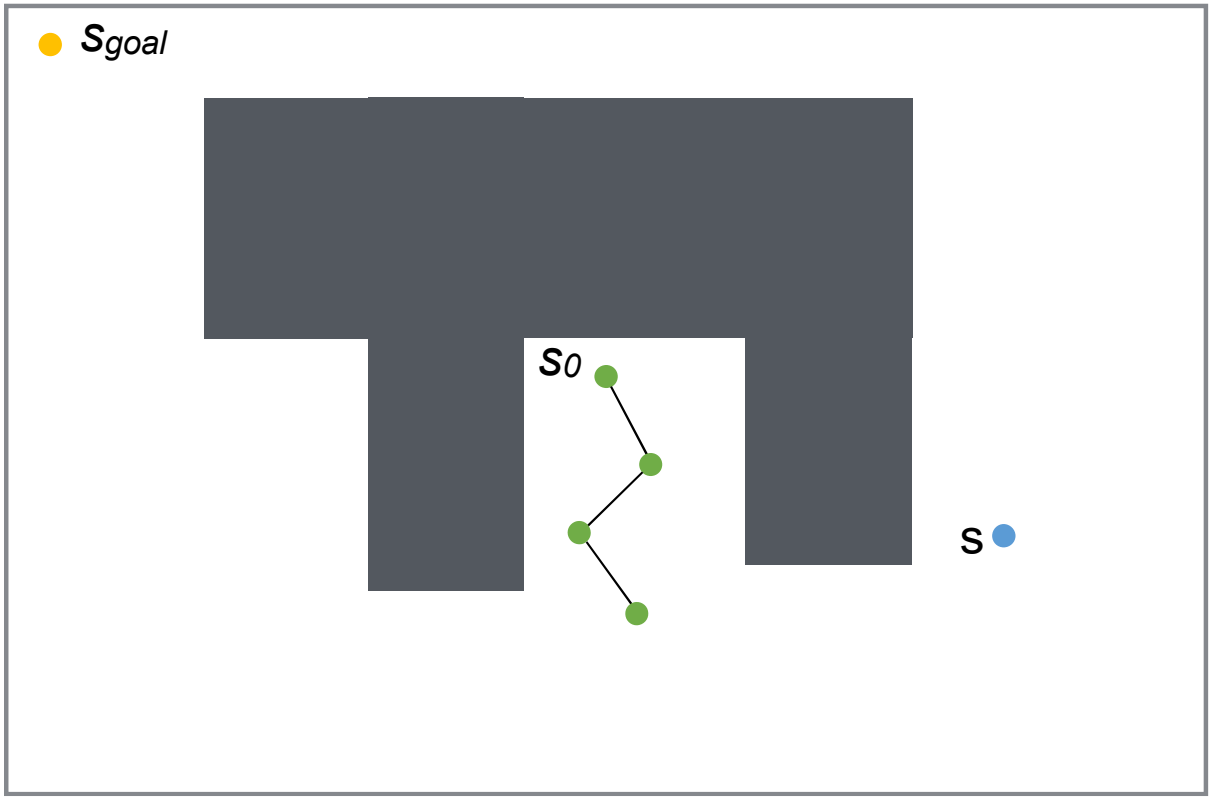
- Sample states $s \in S = R^n$ until s is collision-free
- **Find closest state $s_c \in T$**
- **Extend s_c toward s**
- **Add resulting state s' to T**
- Repeat until T contains a path from s_0 to s_{goal}



5 Rapidly Exploring Random Trees (RRTs)

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

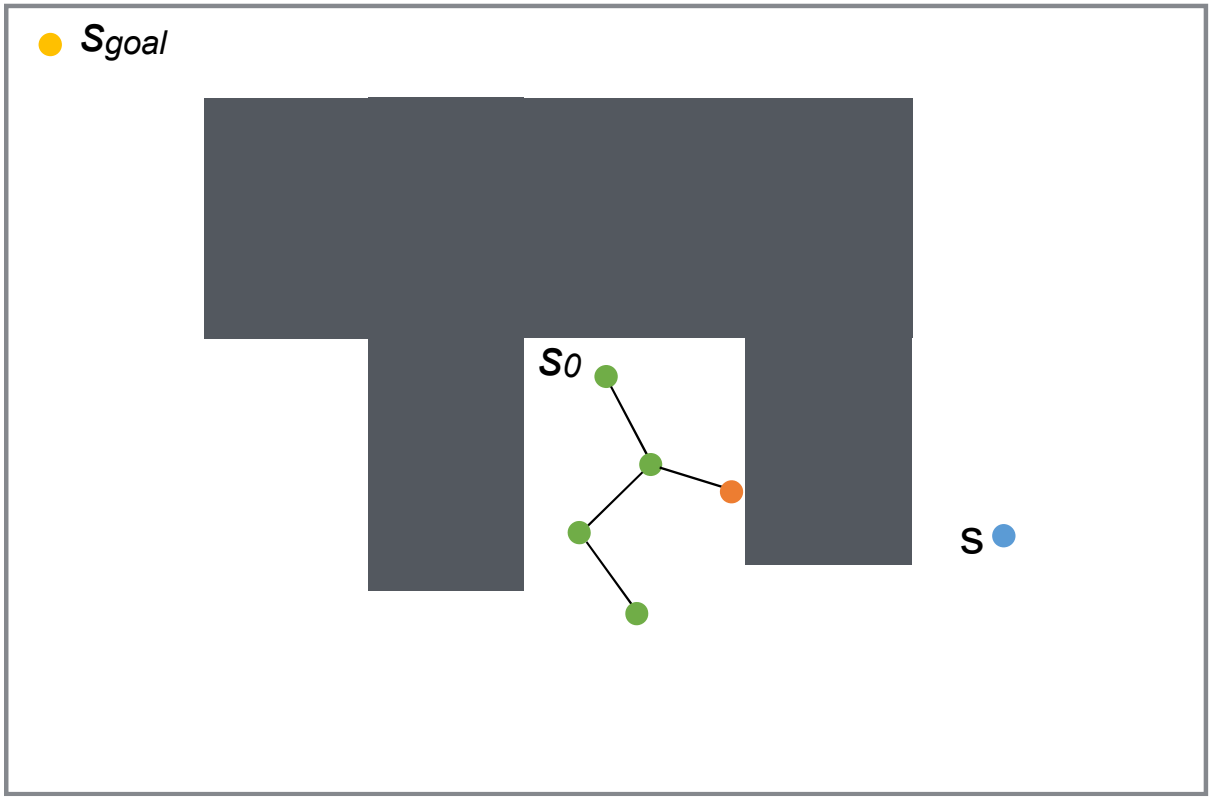
- **Sample states $s \in S = R^n$ until s is collision-free**
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



5 Rapidly Exploring Random Trees (RRTs)

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

- Sample states $s \in S = R^n$ 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}

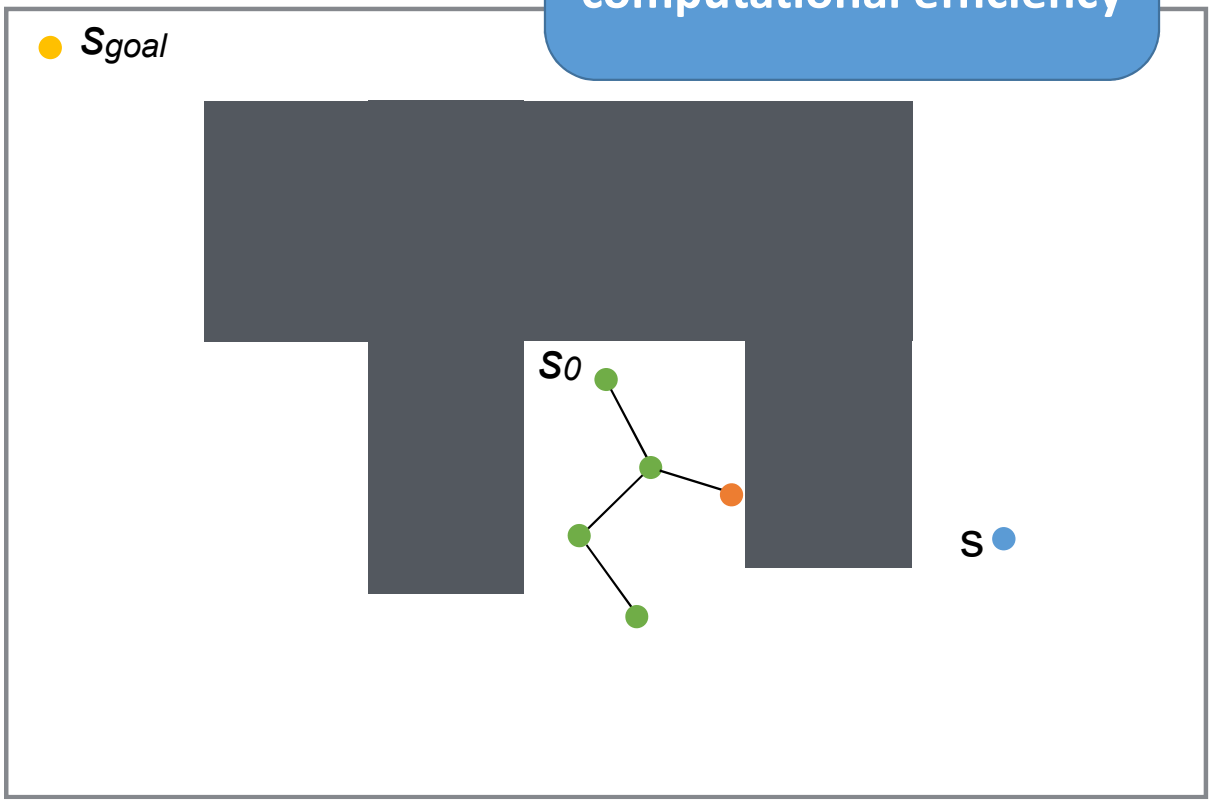


5 Rapidly Exploring Random Trees (RRTs)

Extend distance trades off sample vs. computational efficiency

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

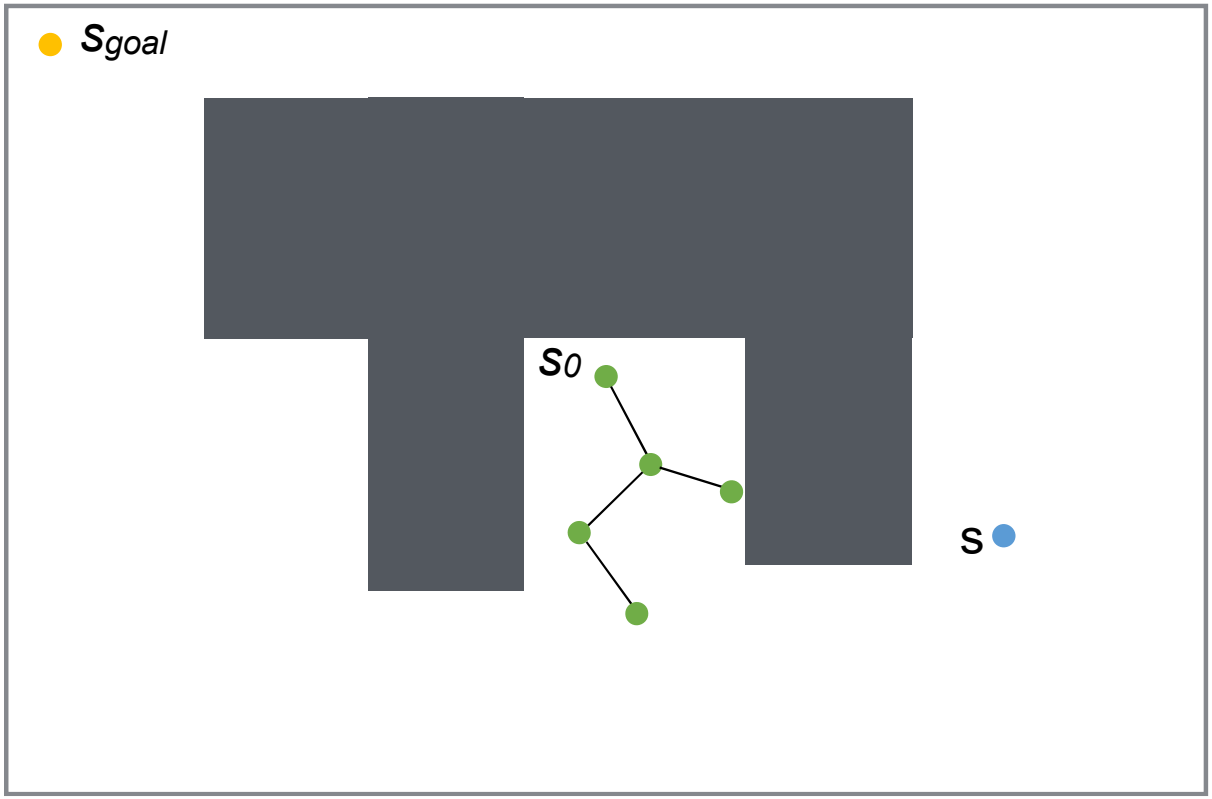
- Sample states $s \in S = R^n$ until s is collision-free
- **Find closest state $s_c \in T$**
- **Extend s_c toward s**
- **Add resulting state s' to T**
- Repeat until T contains a path from s_0 to s_{goal}



5 Rapidly Exploring Random Trees (RRTs)

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

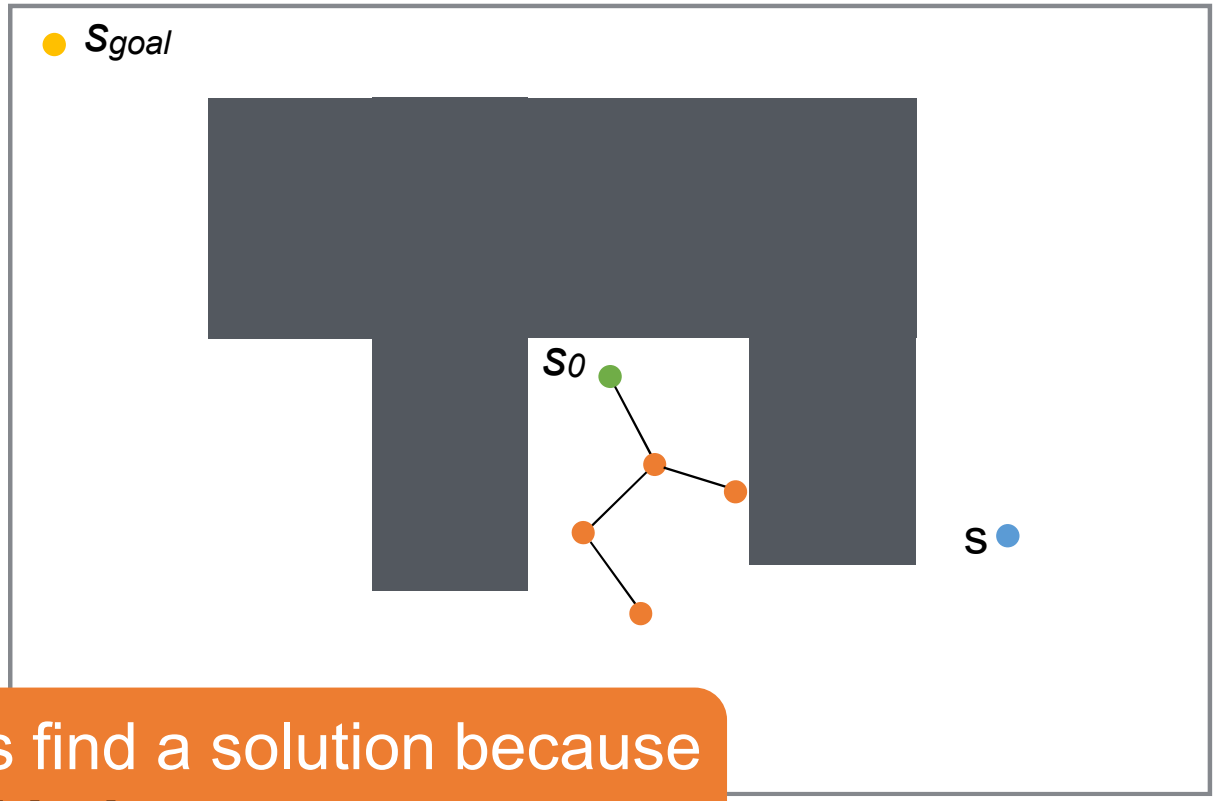
- Sample states $s \in S = R^n$ until s is collision-free
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- **Repeat until T contains a path from s_0 to s_{goal}**



5 Rapidly Exploring Random Trees (RRTs)

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

- Sample states $s \in S = R^n$ 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}**



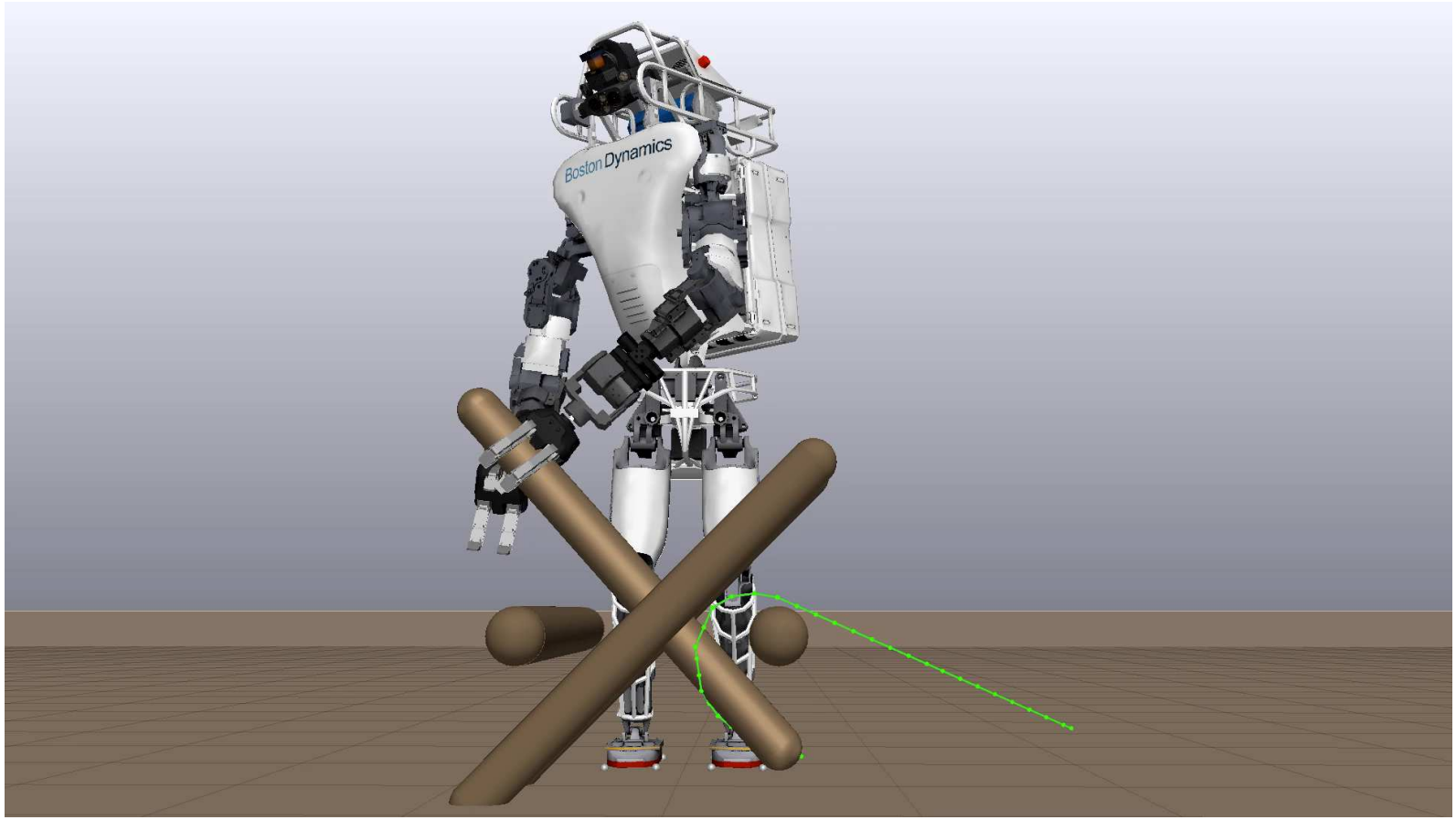
It will always find a solution because it is **probabilistically complete**

5 Rapidly Exploring Random Trees (RRTs)

5 Rapidly Exploring Random Trees (RRTs)

Biased sampling can help!

5 RRTs often works really well in practice



5 RRTs often works really well in practice

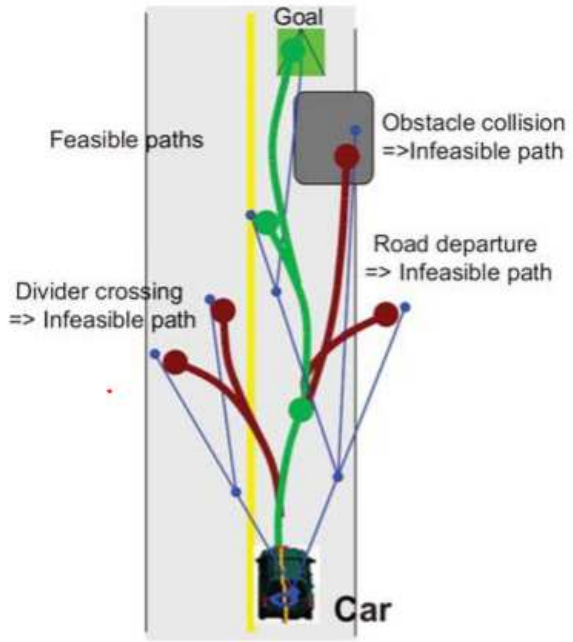
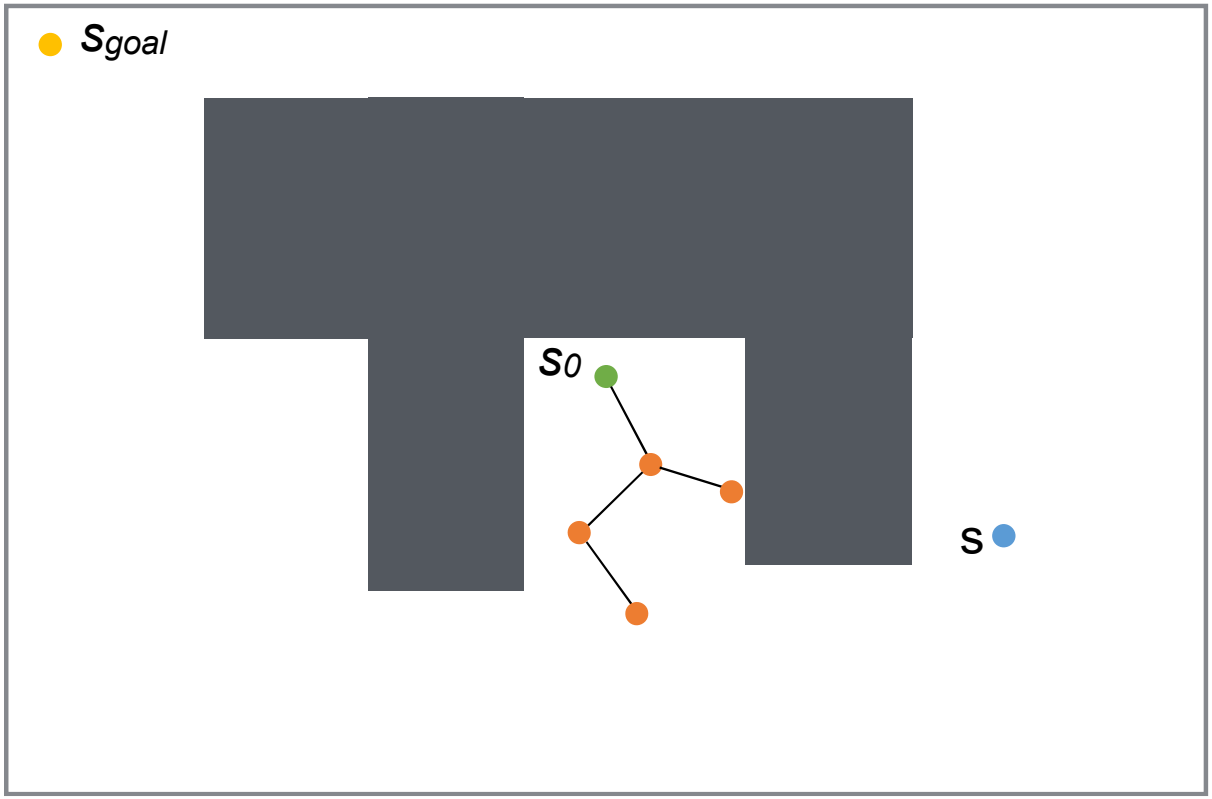


Fig. 22. Illustration of RRT Motion planning. Each leaf of the tree represents a stopping location. The motion control points (in blue) are translated into a predicted path. The predicted paths are checked for drivability (shown in green and red).

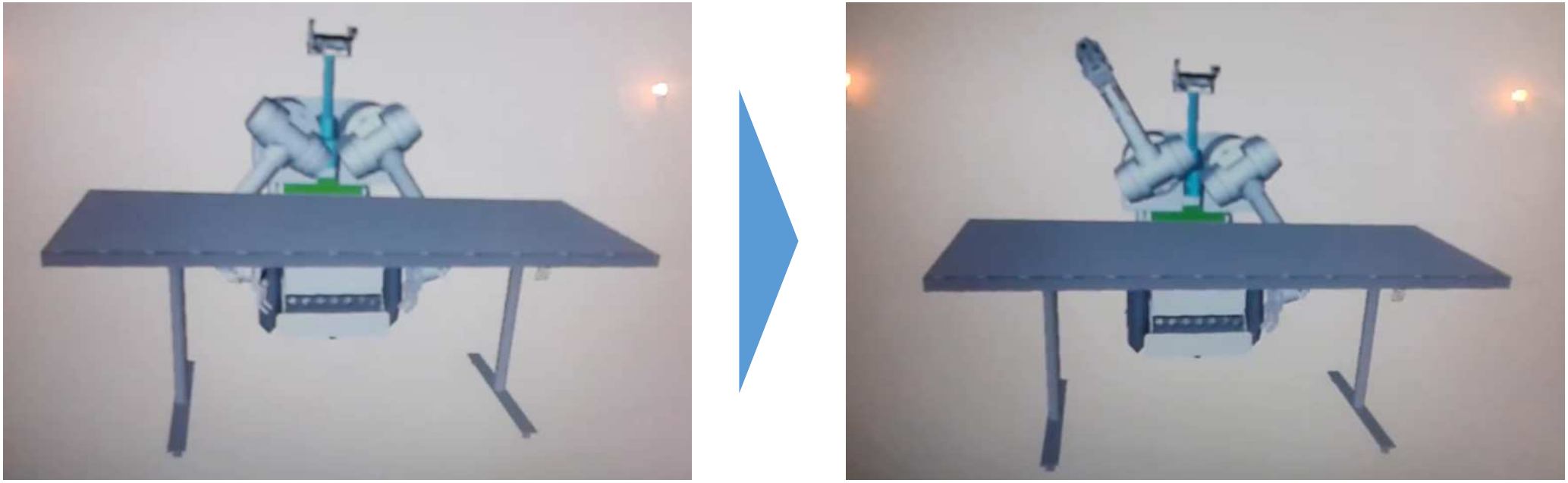
5 Questions about the RRT algorithm?

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

- Sample states $s \in S = R^n$ 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}**

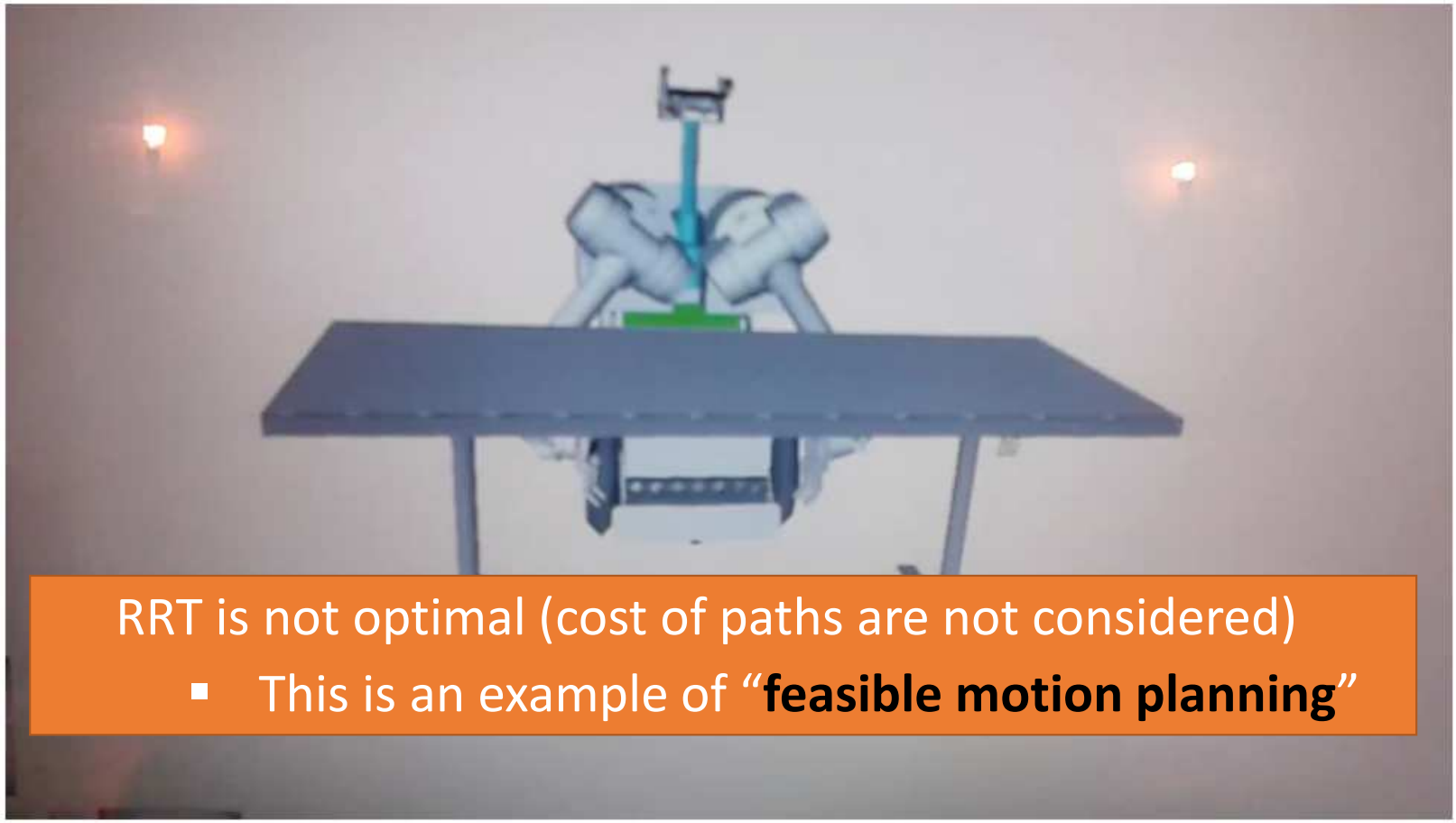


5 But we can get some WEIRD outputs…



5 But we can get some WEIRD outputs…

5 But we can get some WEIRD outputs…



RRT is not optimal (cost of paths are not considered)

- This is an example of "**feasible motion planning**"

5 We solve this problem with RRT*

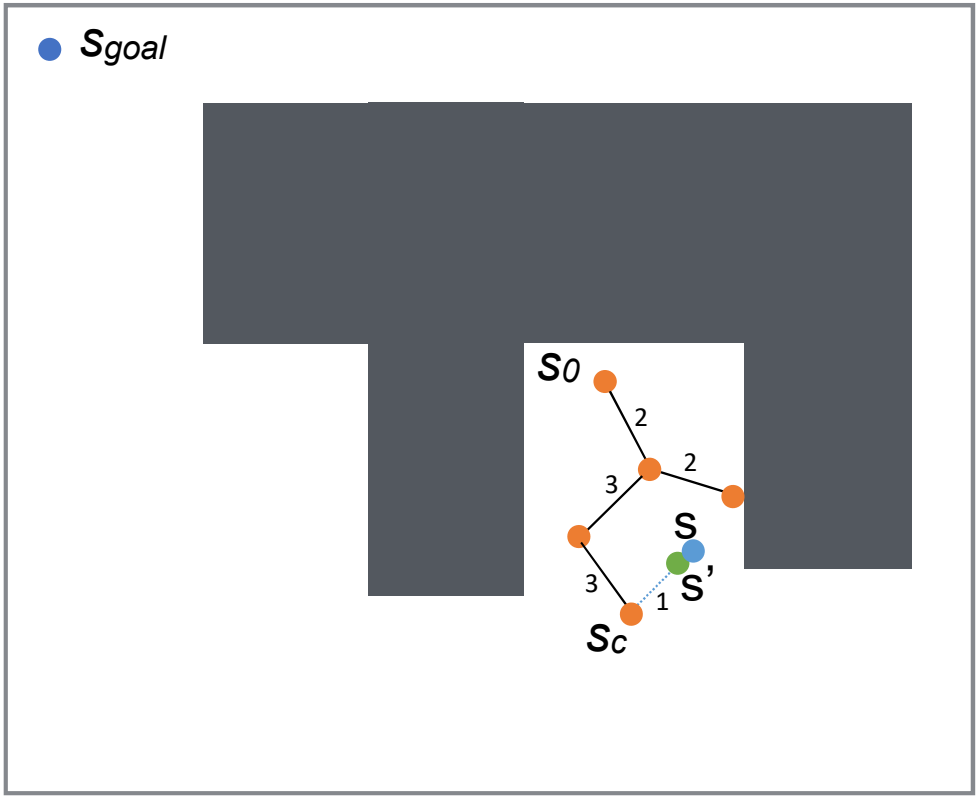
The big trick:

- incrementally "**re-wiring**" the tree to keep locally optimal paths

5 We solve this problem with RRT*

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

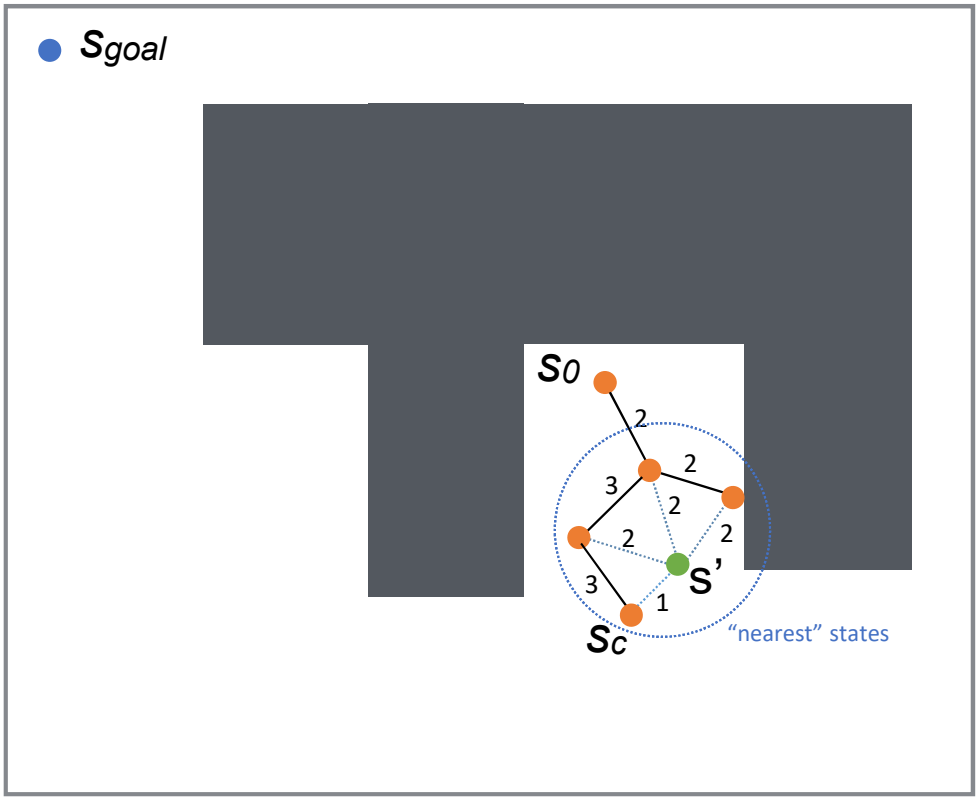
- Sample states $s \in S = R^{15}$ until s is collision-free (often goal directed)
- Find closest state $s_c \in T$
- **Extend s_c toward s resulting in state s'**
- Find all $s_{near} \subseteq T$ within a distance d to s'
- Find $s_{min} \in s_{near}$, that has the lowest *path cost* to s_0 -> s_{min} -> 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}



5 We solve this problem with RRT*

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

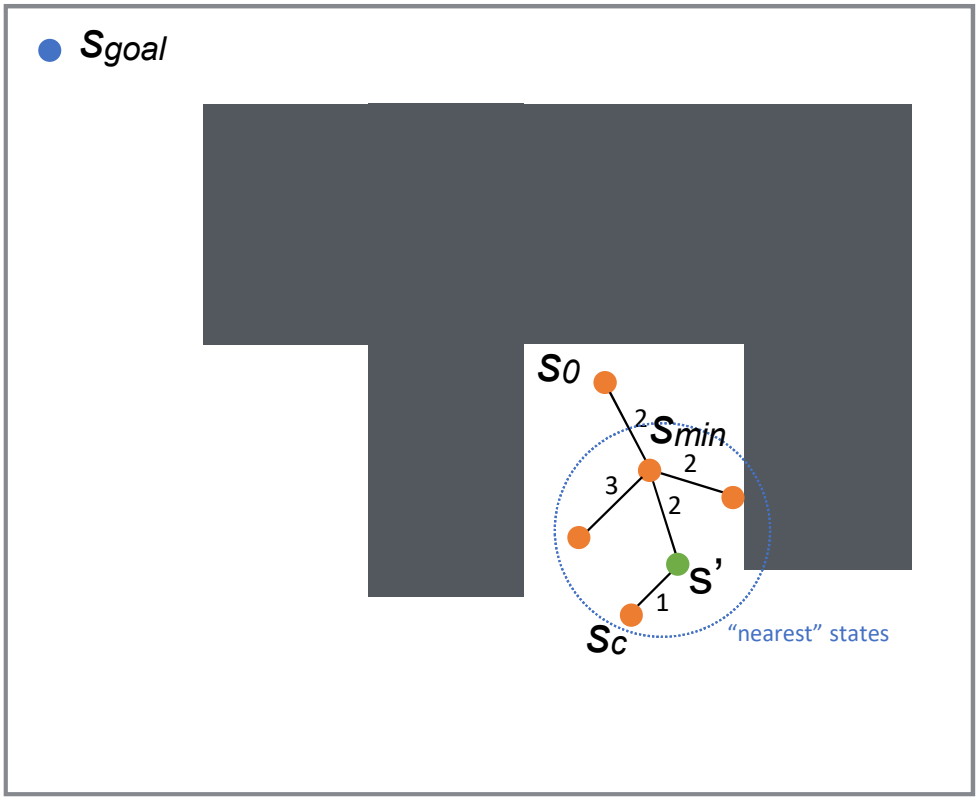
- Sample states $s \in S = R^{15}$ until s is collision-free (often goal directed)
- Find closest state $s_c \in T$
- Extend s_c toward s resulting in state s'
- **Find all $s_{near} \subseteq T$ within a distance d to s'**
- Find $s_{min} \in s_{near}$, that has the lowest *path cost* to s_0 -> s_{min} -> 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}



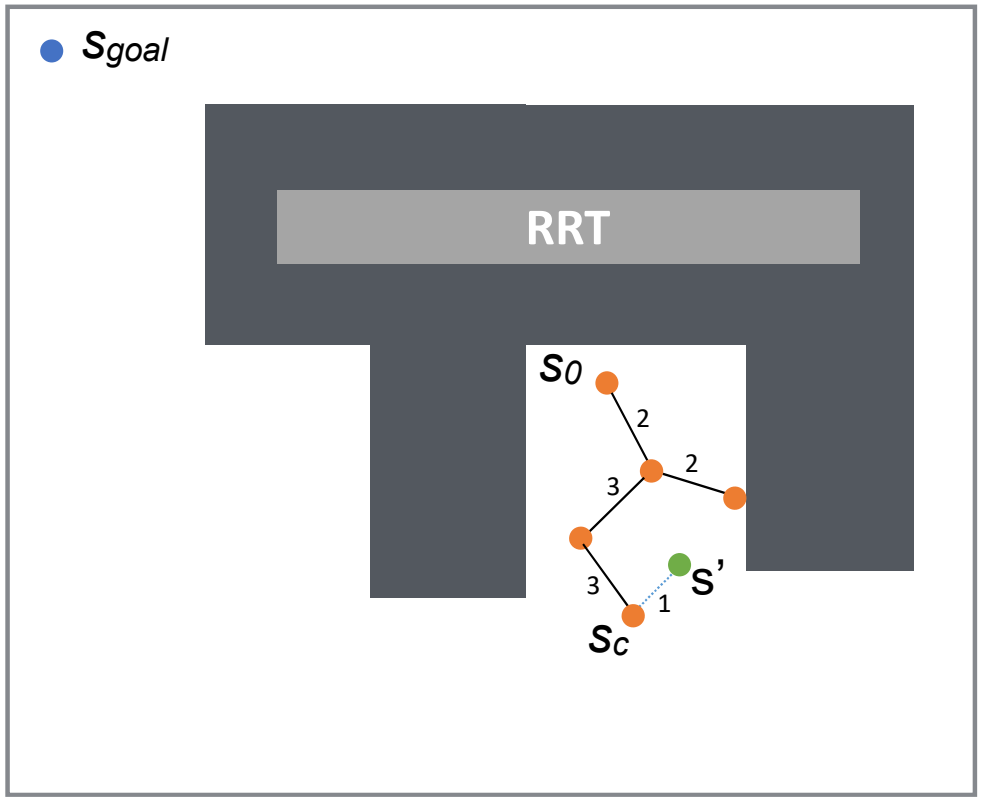
5 We solve this problem with RRT*

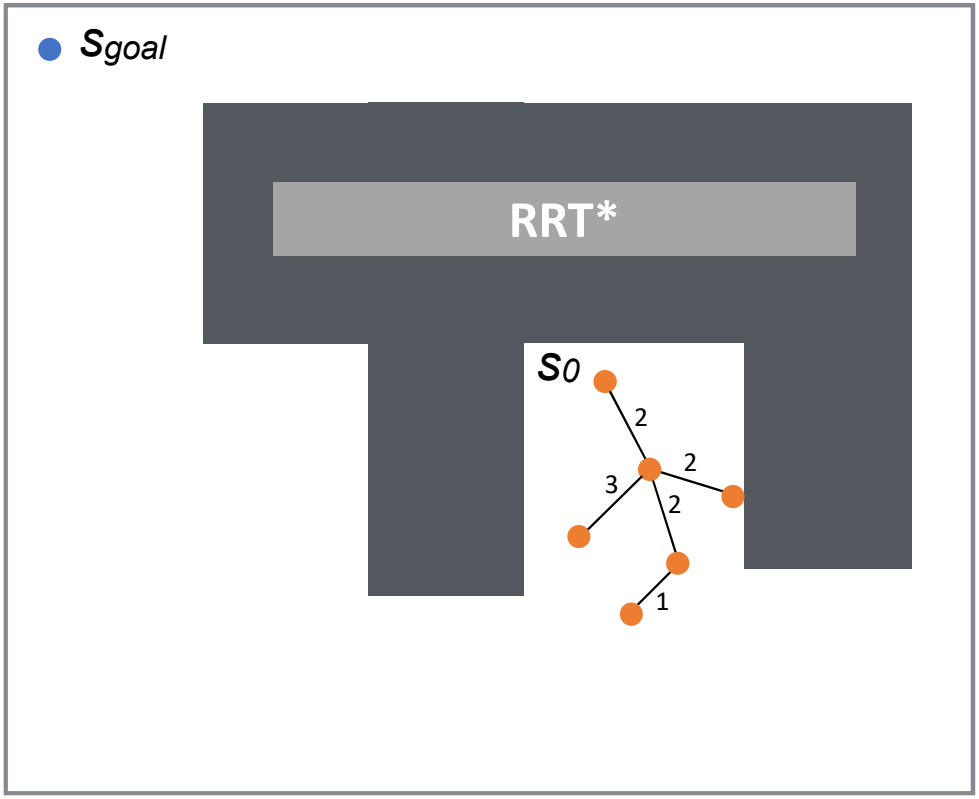
RRT* (input: s_0, s_{goal}, initial state tree T)

- 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}



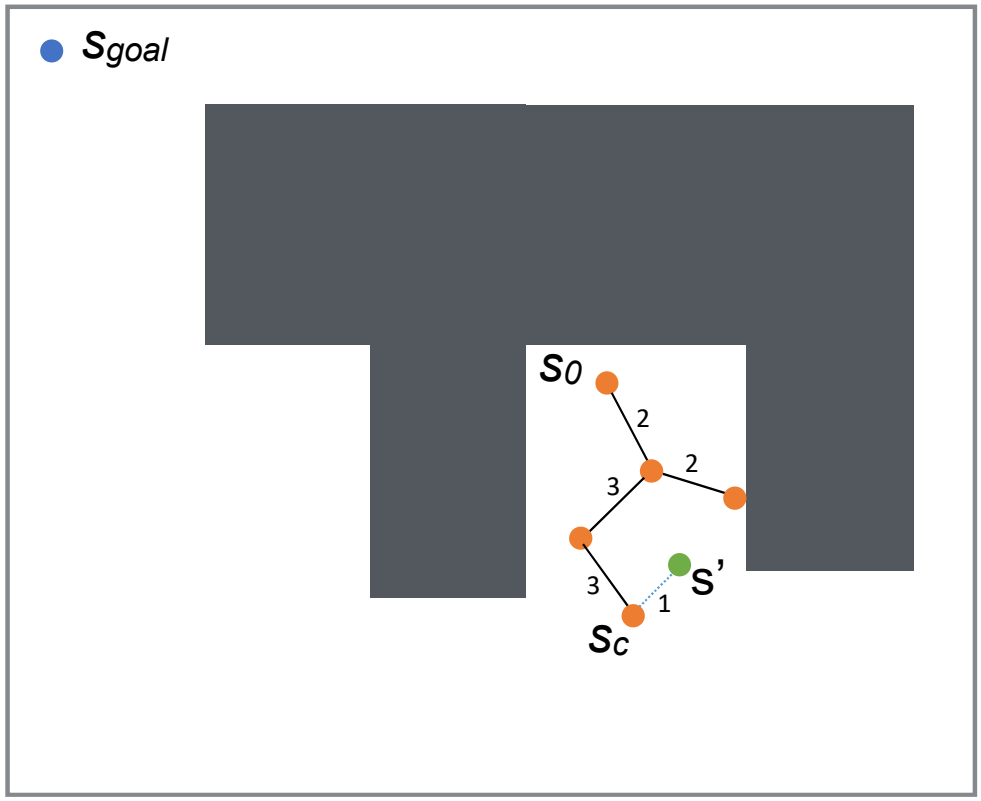
5 We solve this problem with RRT*

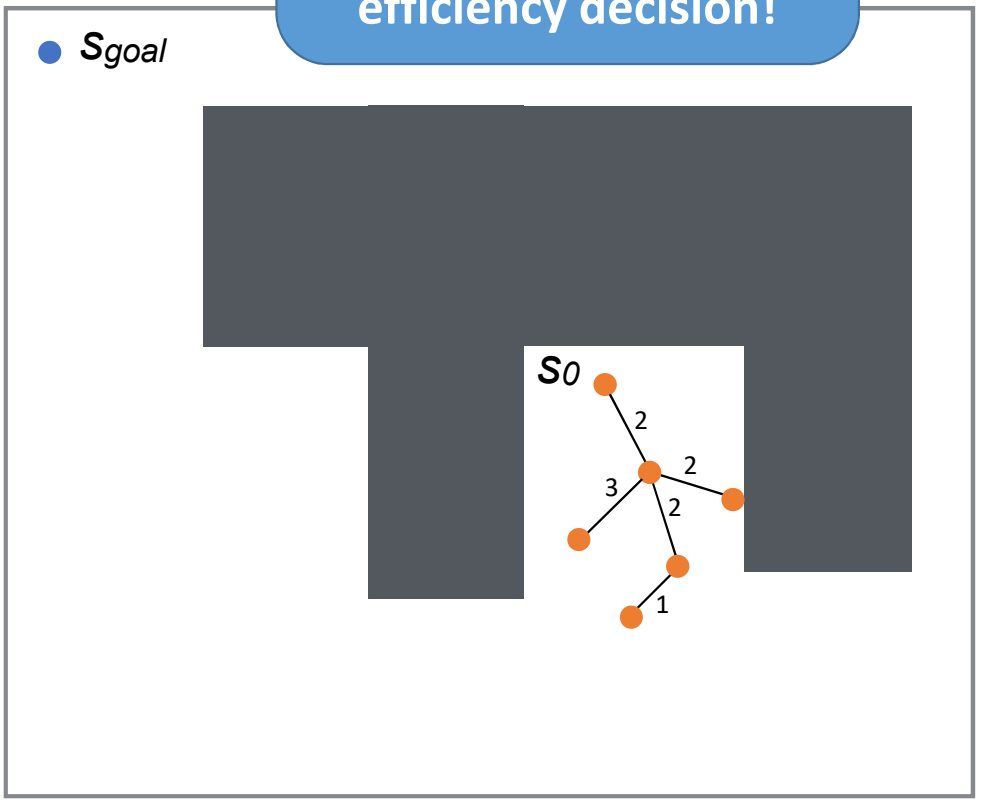




5 We solve this problem with RRT*

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





5 RRT*

[Karaman & Fazzoli Sampling-based Algorithms for Optimal Motion Planning]

RRT

RRT*

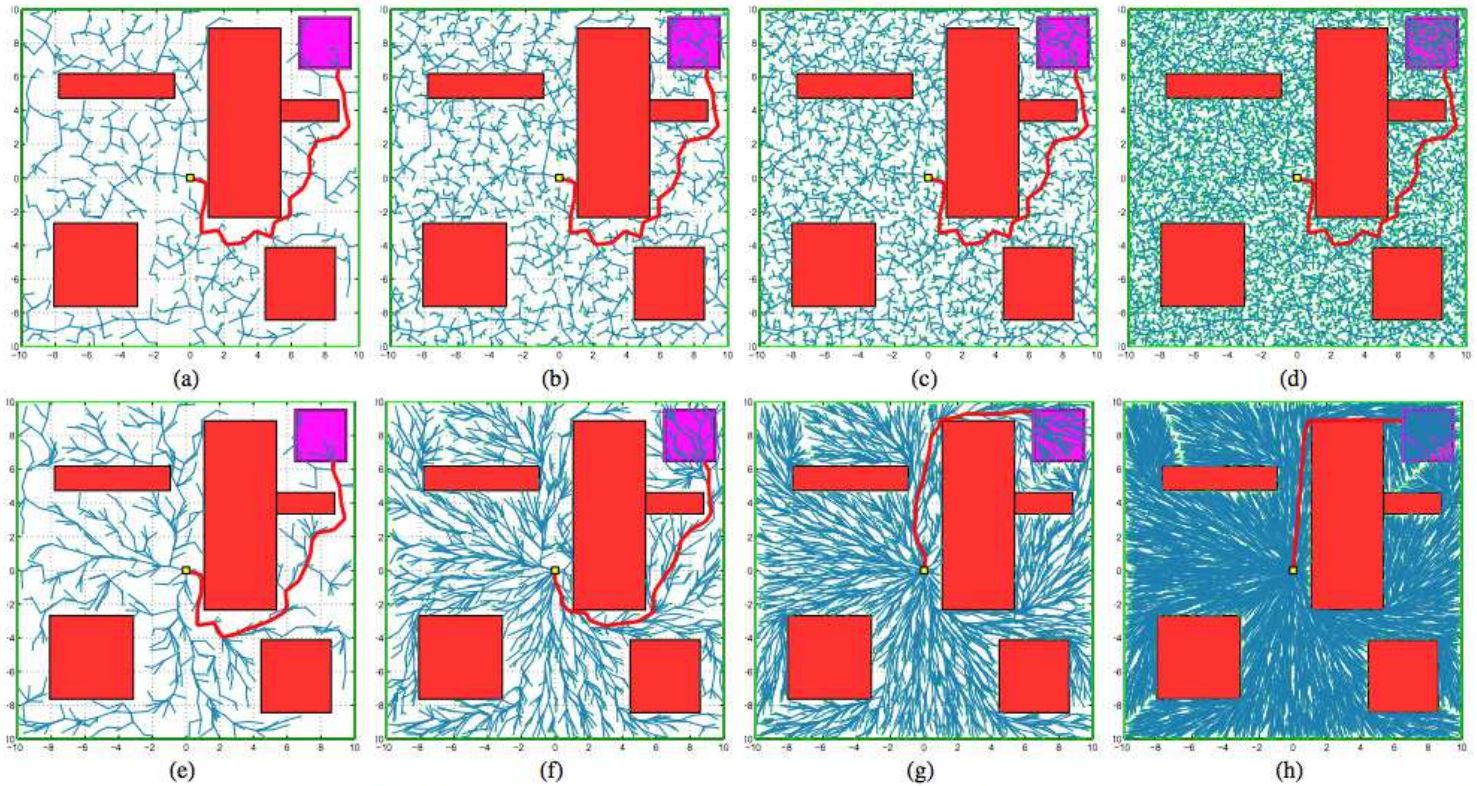


Fig. 1. A Comparison of the RRT* and RRT algorithms on a simulation example. The tree maintained by the RRT algorithm is shown in (a)-(d) in different stages, whereas that maintained by the RRT* algorithm is shown in (e)-(h). The tree snapshots (a), (e) are at 1000 iterations , (b), (f) at 2500 iterations, (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.

5 So what have we learned so far?

1. Robot planning usually involves thinking about both **task and configuration spaces**
2. For many real problems, **collision checking** can be expensive
3. **RRT**: a powerful algorithm based on a very simple idea!
   - **Probabilistically complete:** If there's a solution it will find it eventually (but can still be slow for some problems)!
   - BUT RRT is not optimal (cost of paths are not considered)
     - This is an example of "**feasible motion planning**"
     - **RRT*** fixes that by incrementally rewiring the tree

5 To RRT or not to RRT that is the question!

1. Why might RRTs not be the best algorithmic choice for a robot that repeatedly does the same task?
2. How might you adapt RRT to fix this issue?

5 To RRT or not to RRT that is the question!

1. Why might RRTs not be the best algorithmic choice for a robot that repeatedly does the same task?
2. How might you adapt RRT to fix this issue?

1. RRT is a "**single-query**" algorithm – it starts from scratch each time "forgetting" all of the connections it found in previous solves

5 To RRT or not to RRT that is the question!

1. Why might RRTs not be the best algorithmic choice for a robot that repeatedly does the same task?
2. How might you adapt RRT to fix this issue?

1. RRT is a "**single-query**" algorithm – it starts from scratch each time "forgetting" all of the connections it found in previous solves
2. Instead of building a tree lets build a **reusable graph *G***

5 To RRT or not to RRT that is the question!

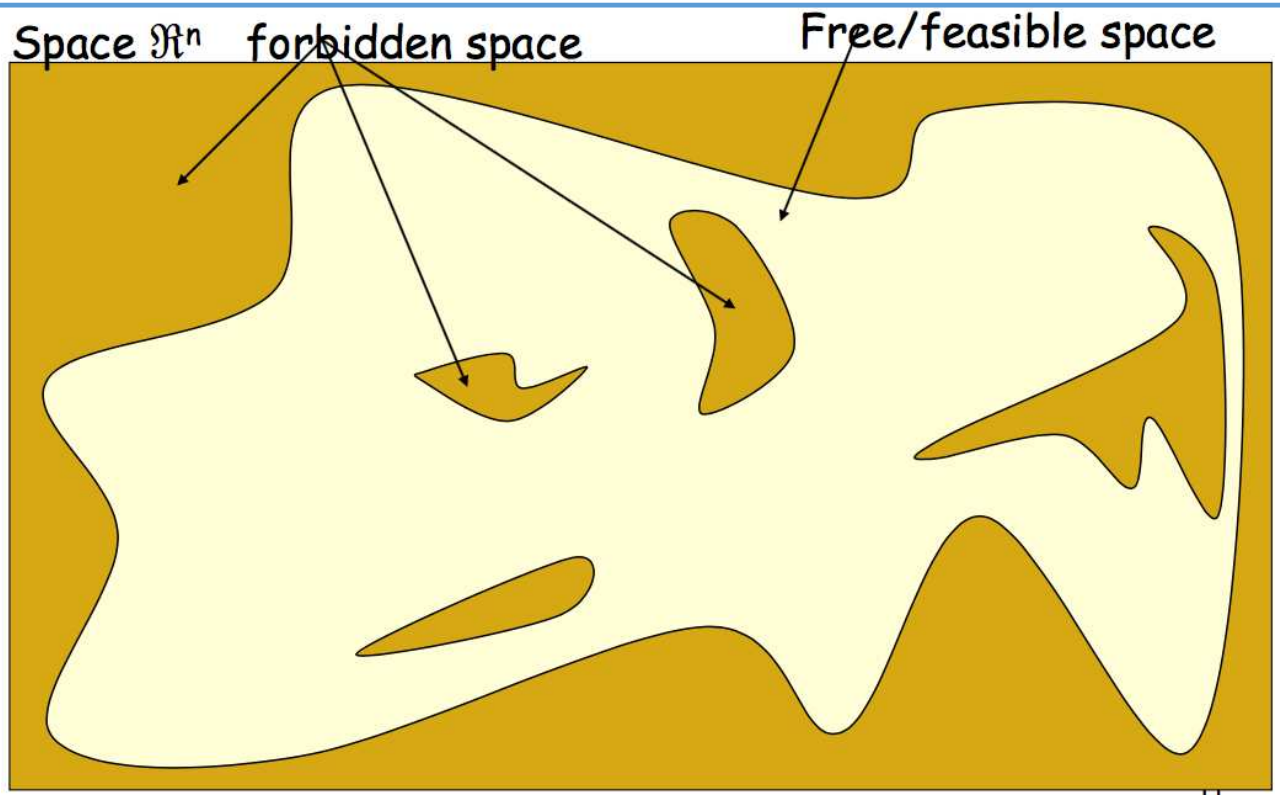
1. Why might RRTs not be the best algorithmic choice for a robot that repeatedly does the same task?
2. How might you adapt RRT to fix this issue?

1. RRT is a "**single-query**" algorithm – it starts from scratch each time "forgetting" all of the connections it found in previous solves
2. Instead of building a tree lets build a **reusable graph *G***

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

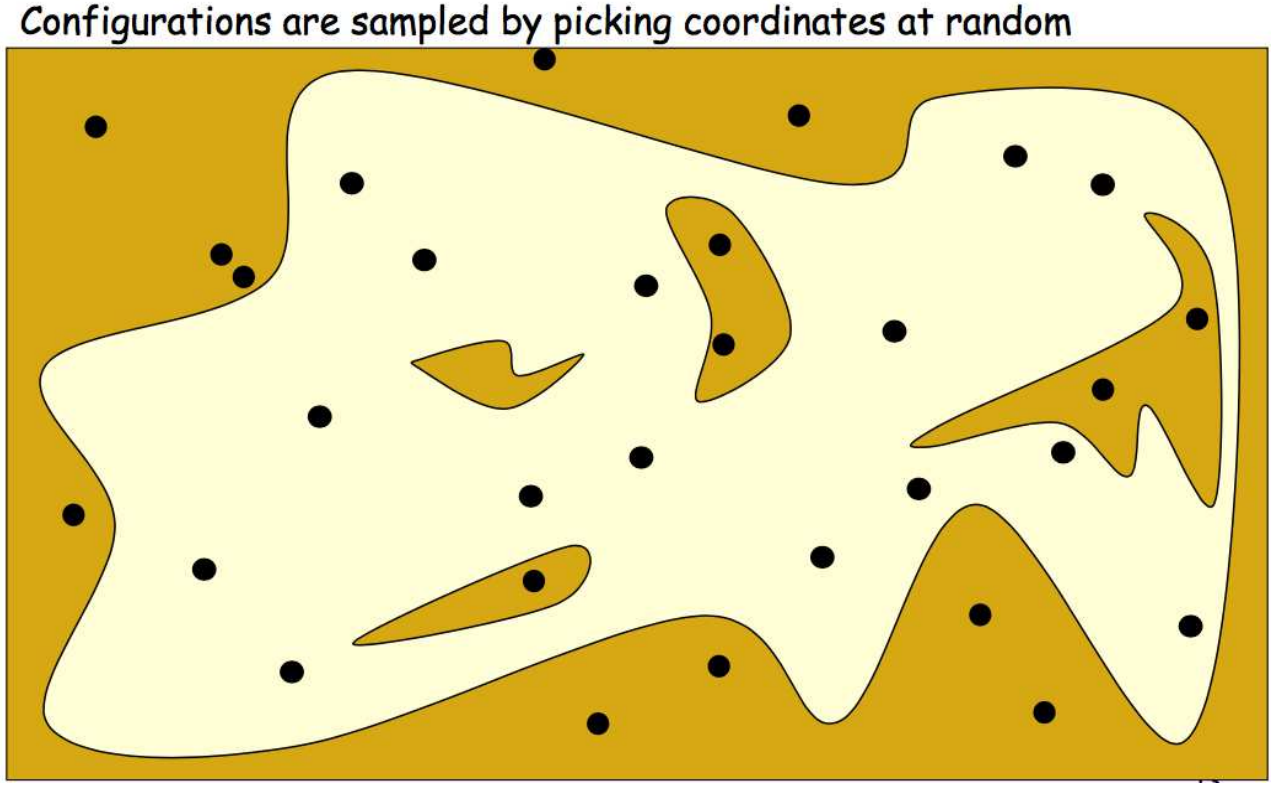
Probabilistic Roadmaps (PRMs) leverage an offline and an online computation phase

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



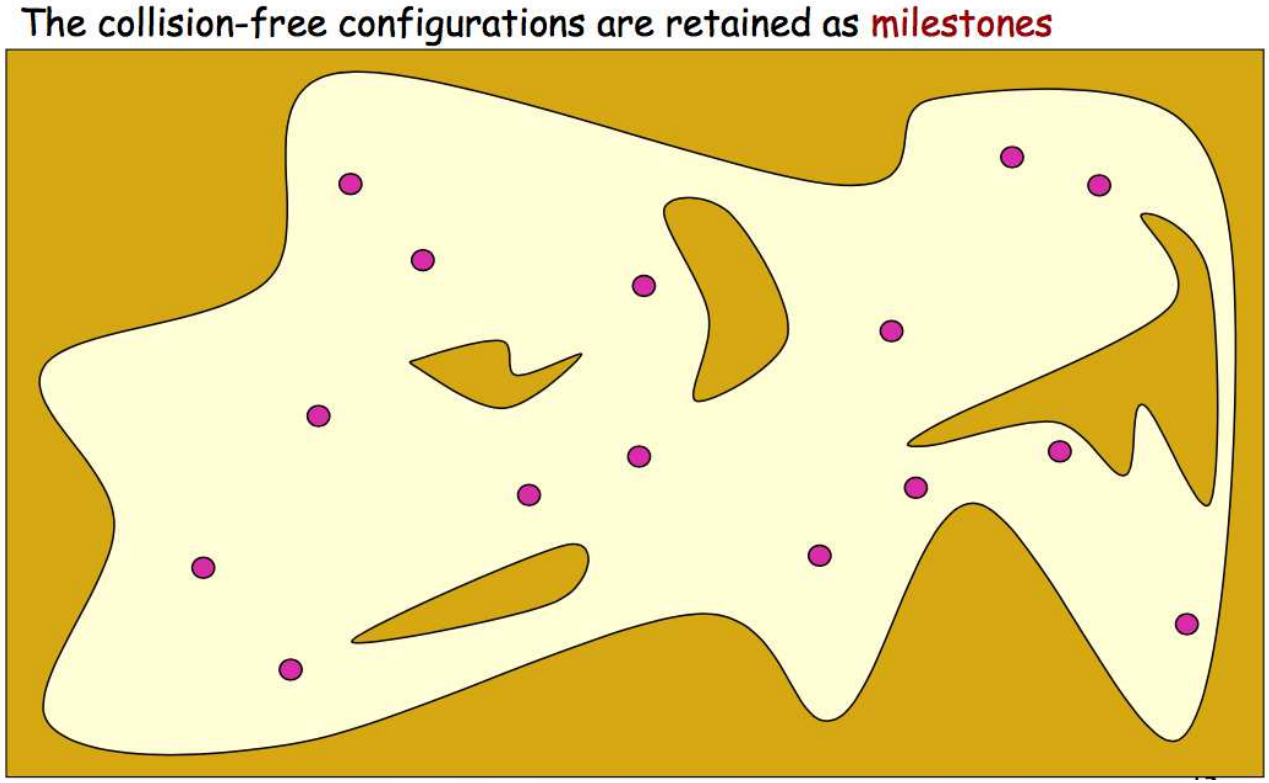
Probabilistic Roadmaps (PRMs) leverage an offline and an online computation phase

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



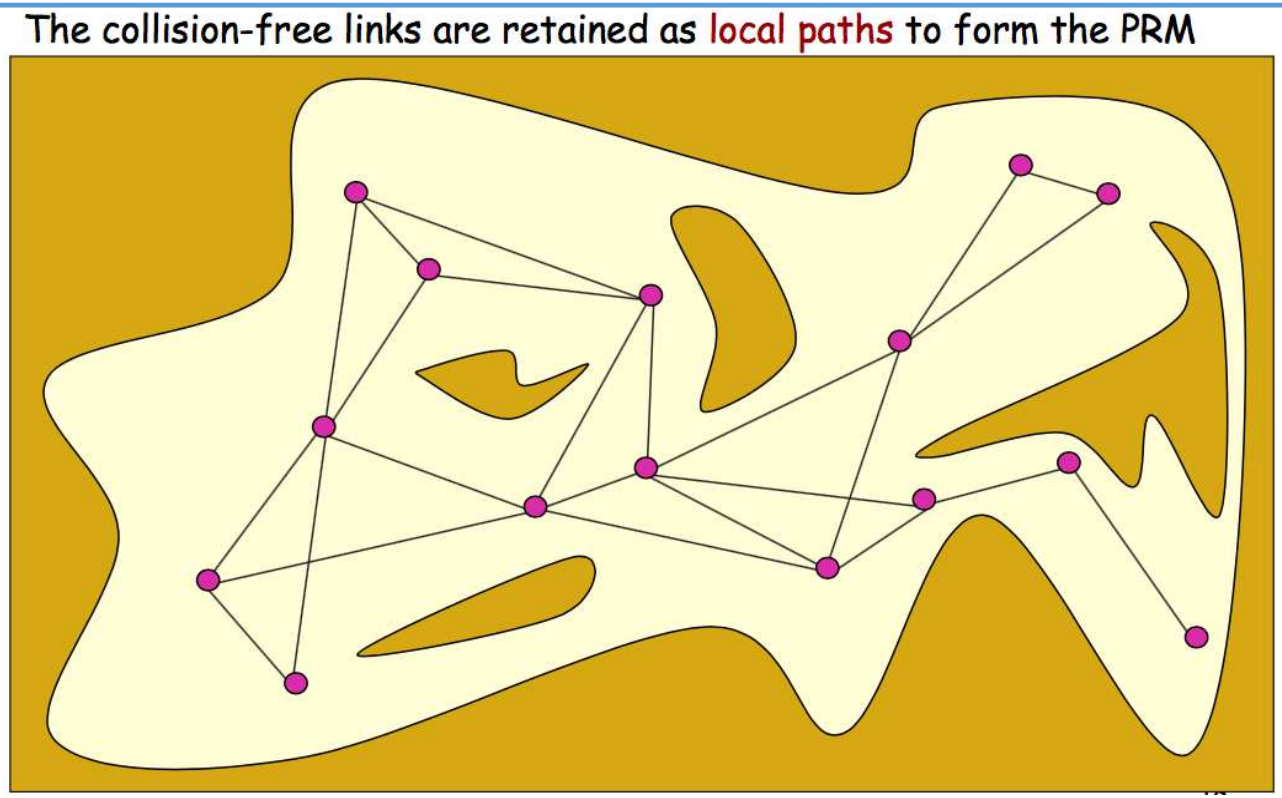
Probabilistic Roadmaps (PRMs) leverage an offline and an online computation phase

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



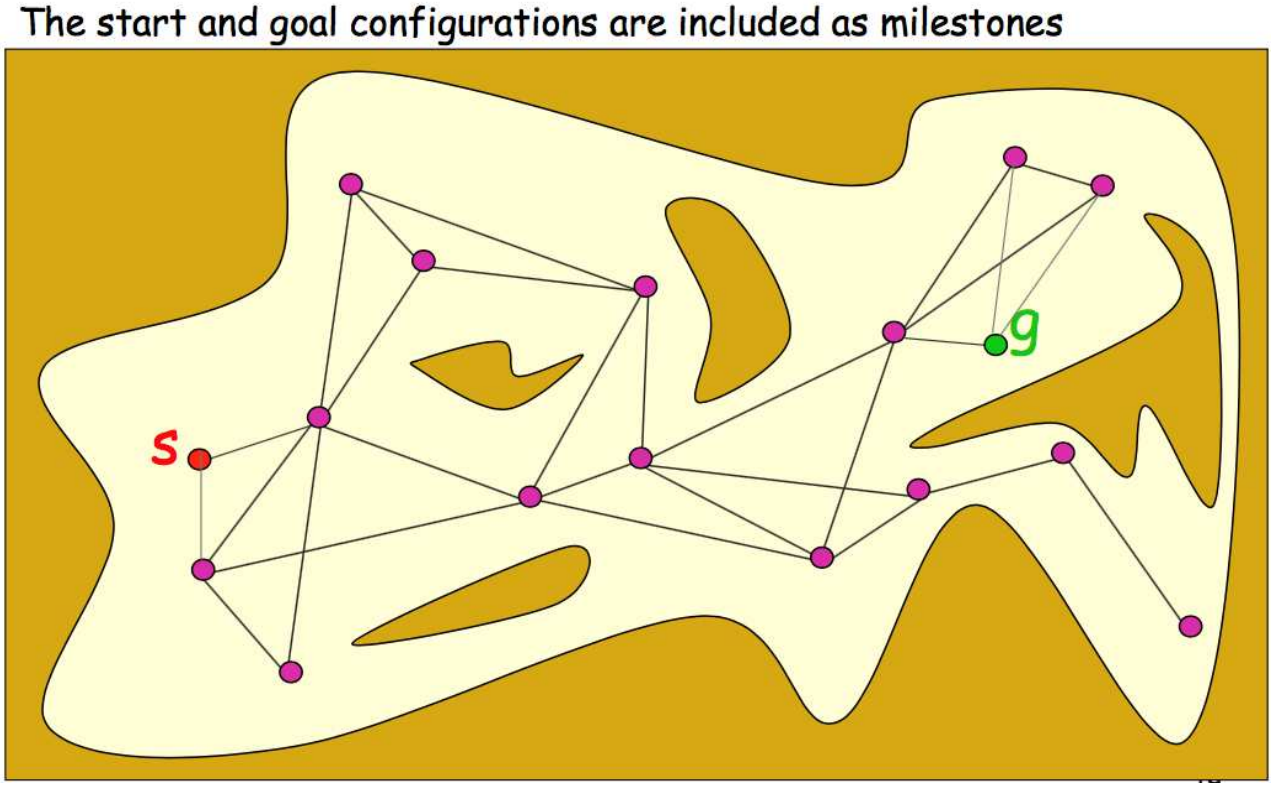
Probabilistic Roadmaps (PRMs) leverage an offline and an online computation phase

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



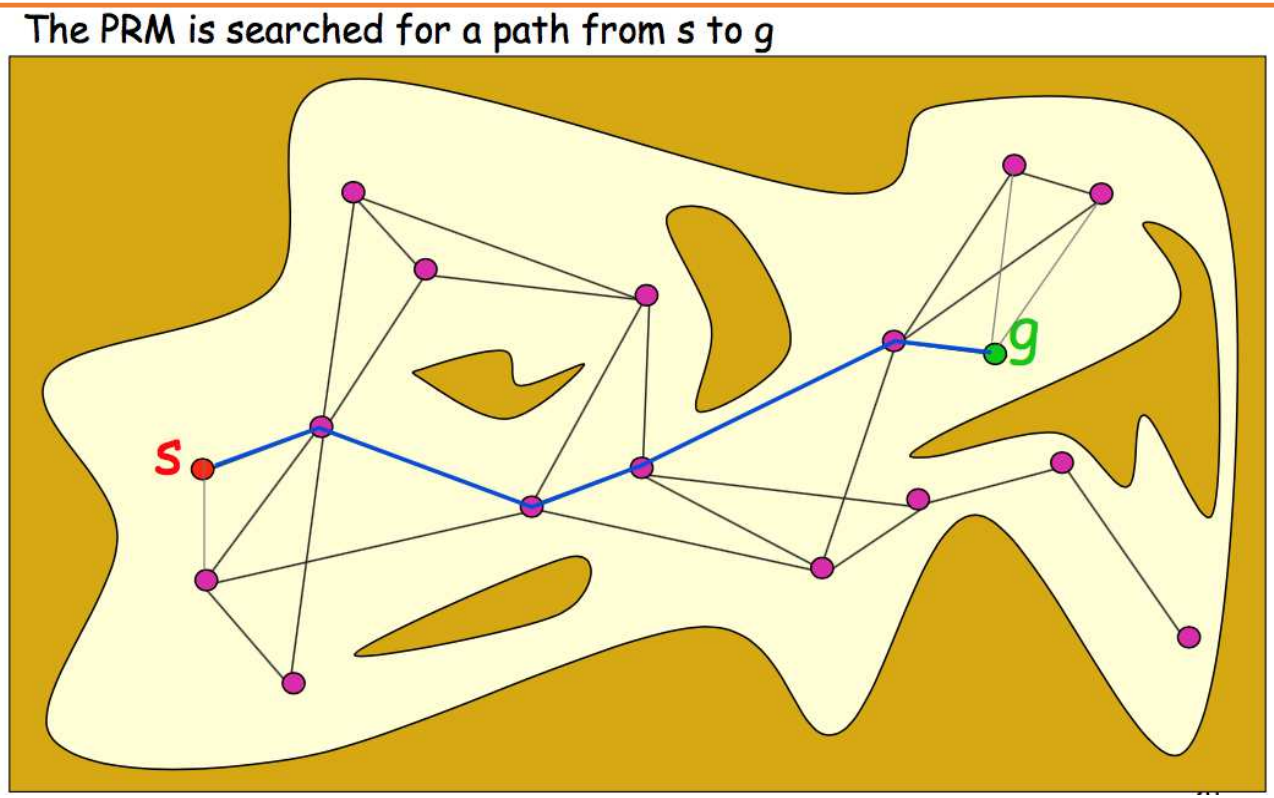
Probabilistic Roadmaps (PRMs) leverage an offline and an online computation phase

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

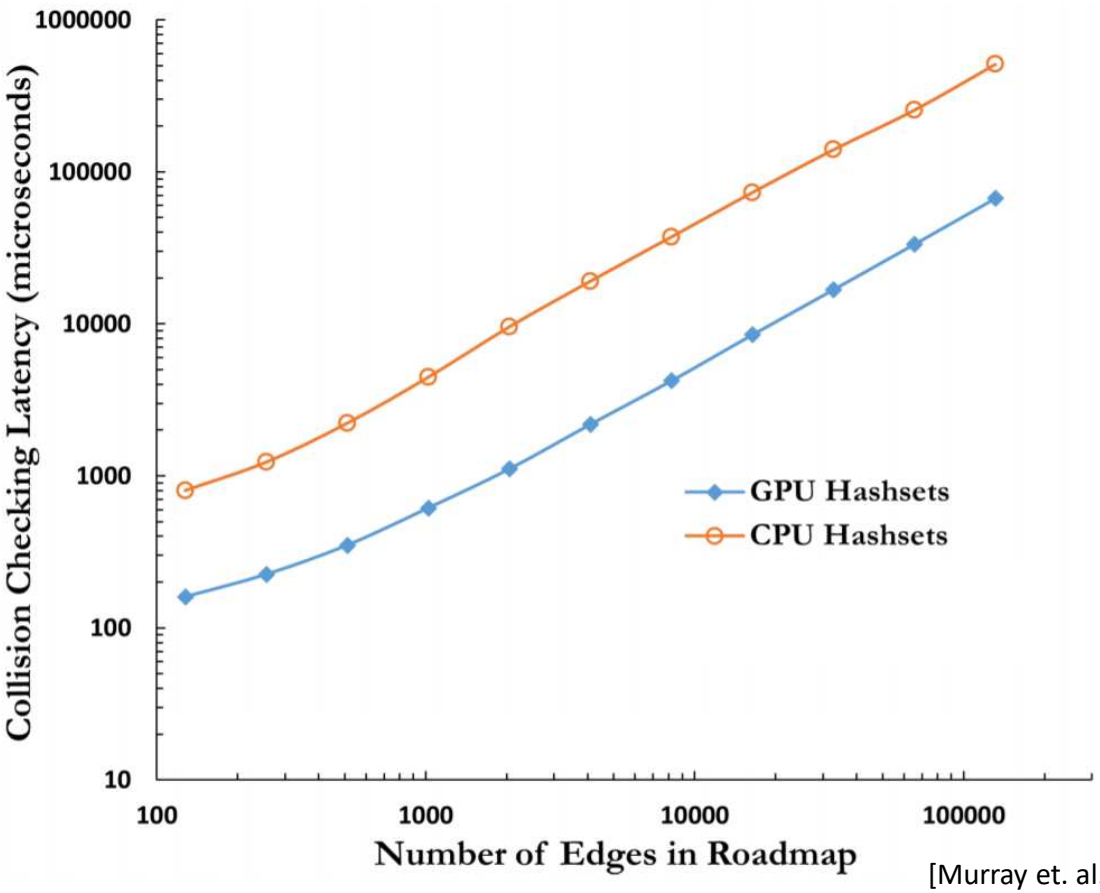


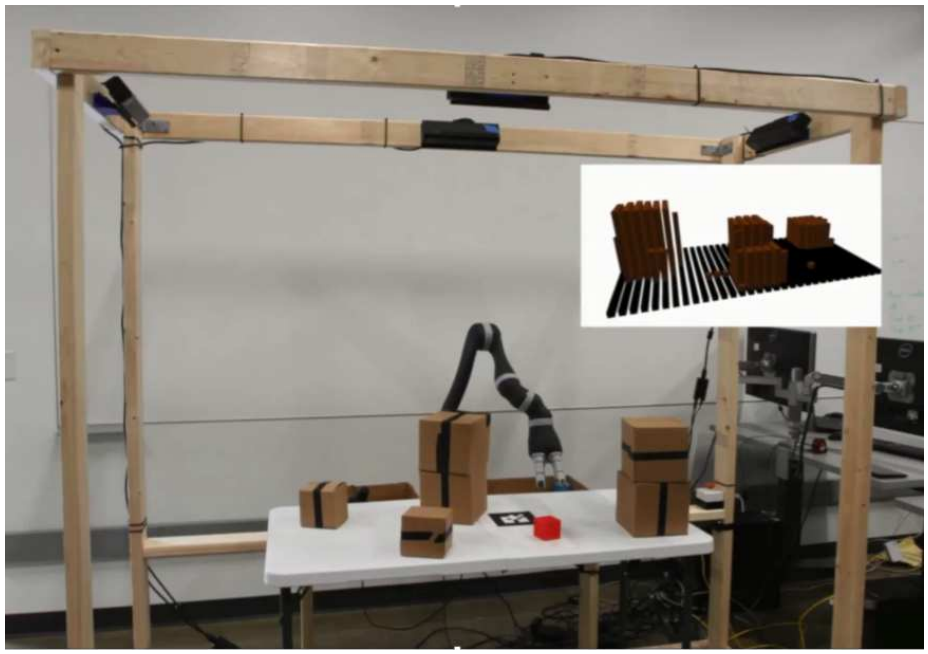
Probabilistic Roadmaps (PRMs) leverage an offline and an online computation phase

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



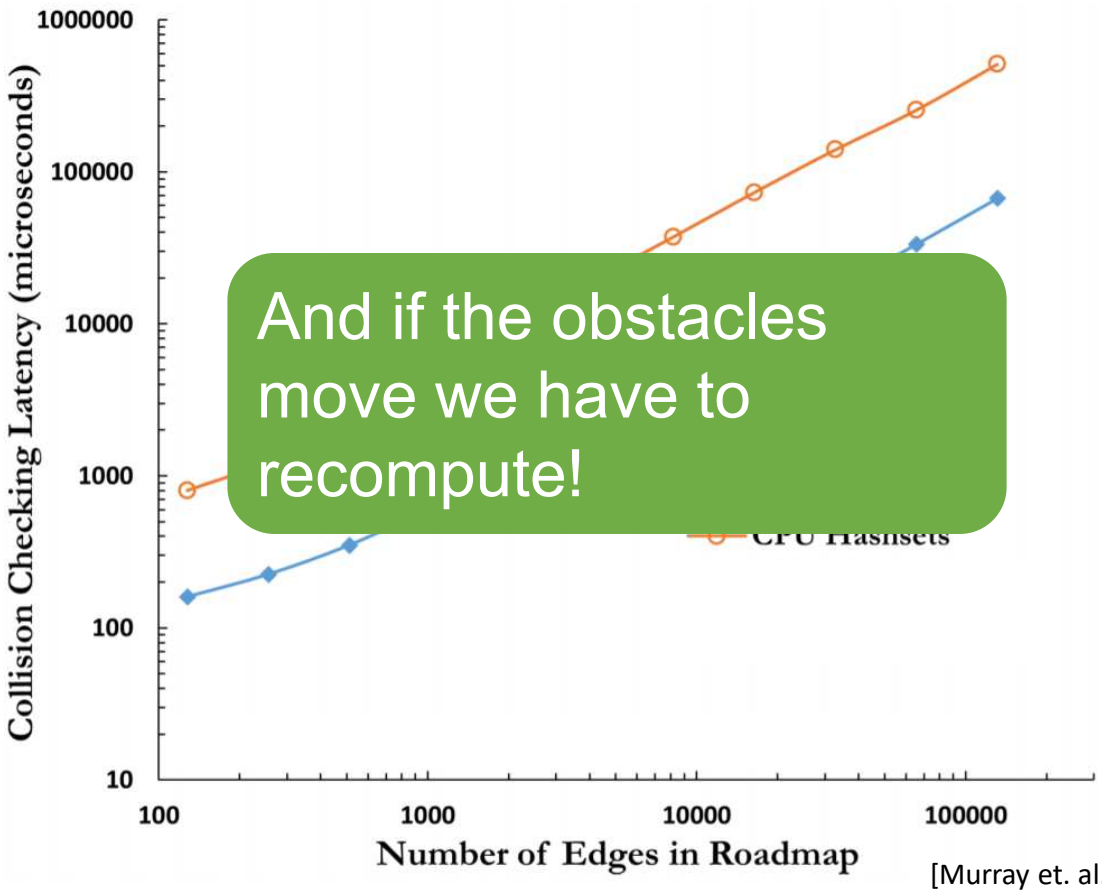
Collision detection for each connecting path in the construction of the PRM can be very expensive



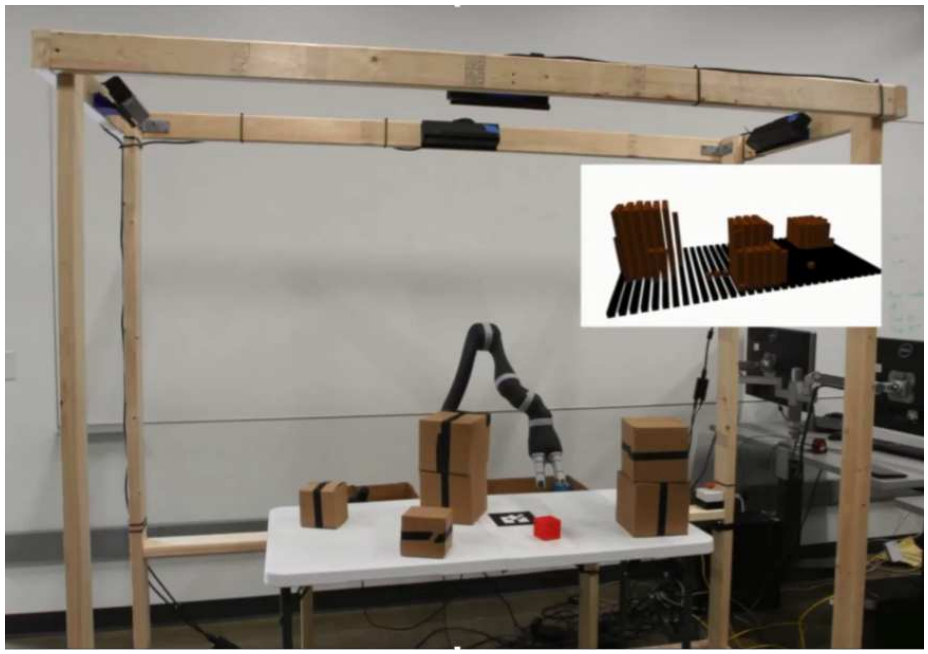


[Murray et. al. The Microarchitecture of a Real-Time Robot Motion Planning Accelerator]

5 Collision detection for each connecting path in the construction of the PRM can be very expensive

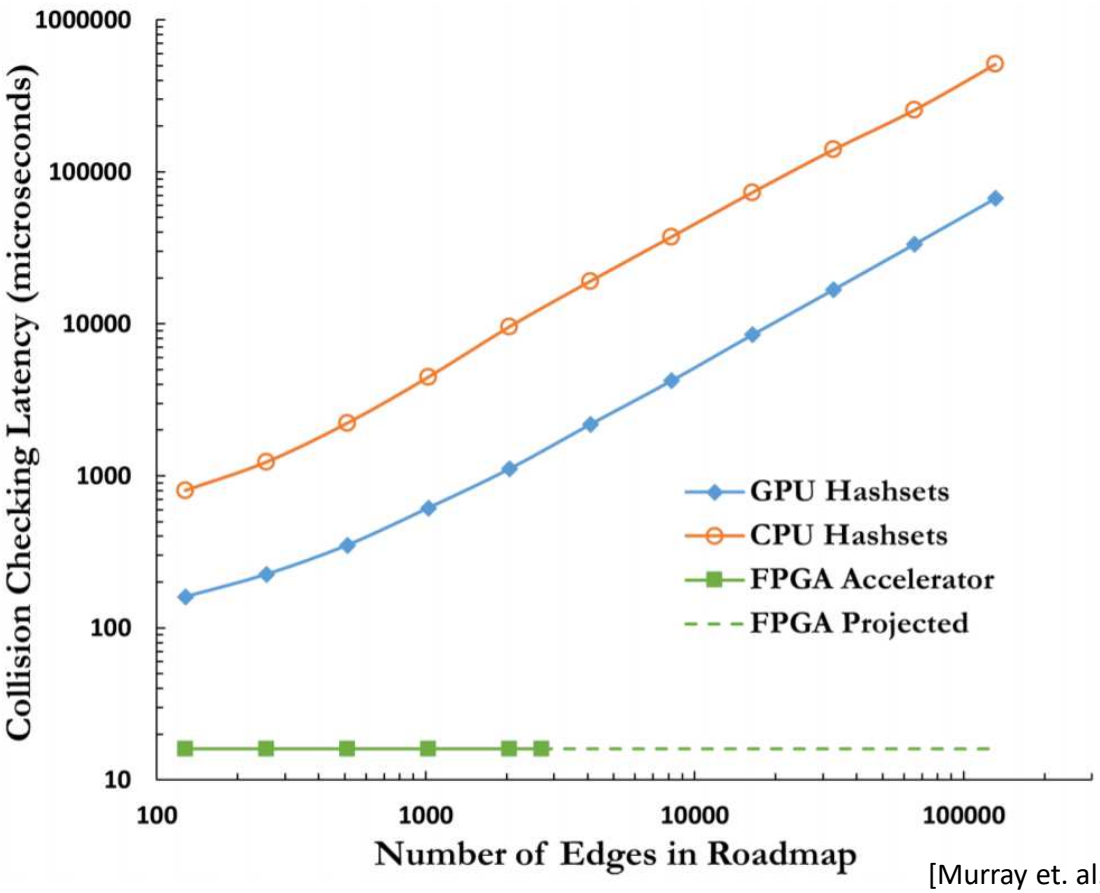


And if the obstacles move we have to recompute!



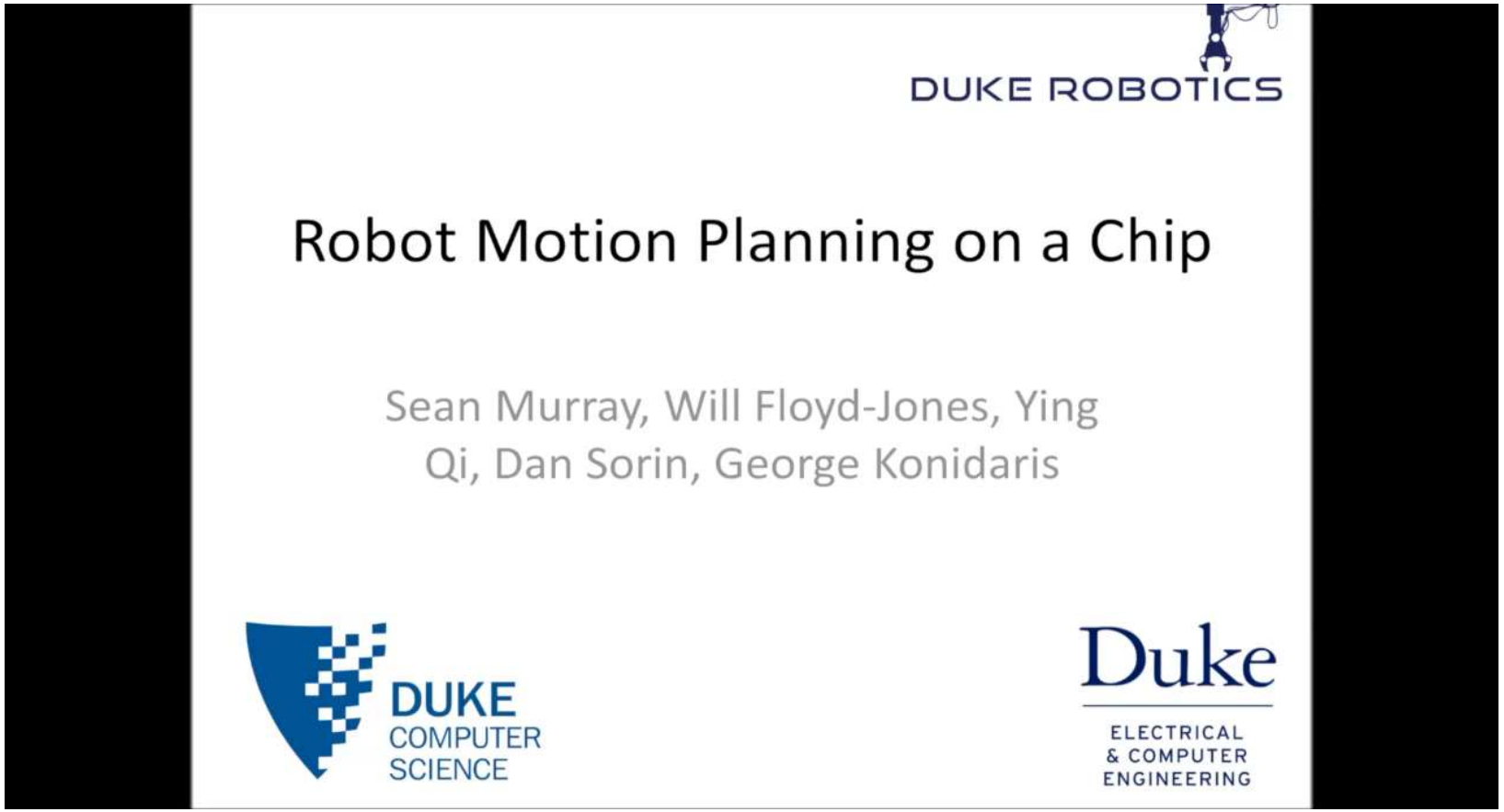
[Murray et. al. The Microarchitecture of a Real-Time Robot Motion Planning Accelerator]

Collision detection for each connecting path in the construction of the PRM can be very expensive



[Murray et. al. The Microarchitecture of a Real-Time Robot Motion Planning Accelerator]

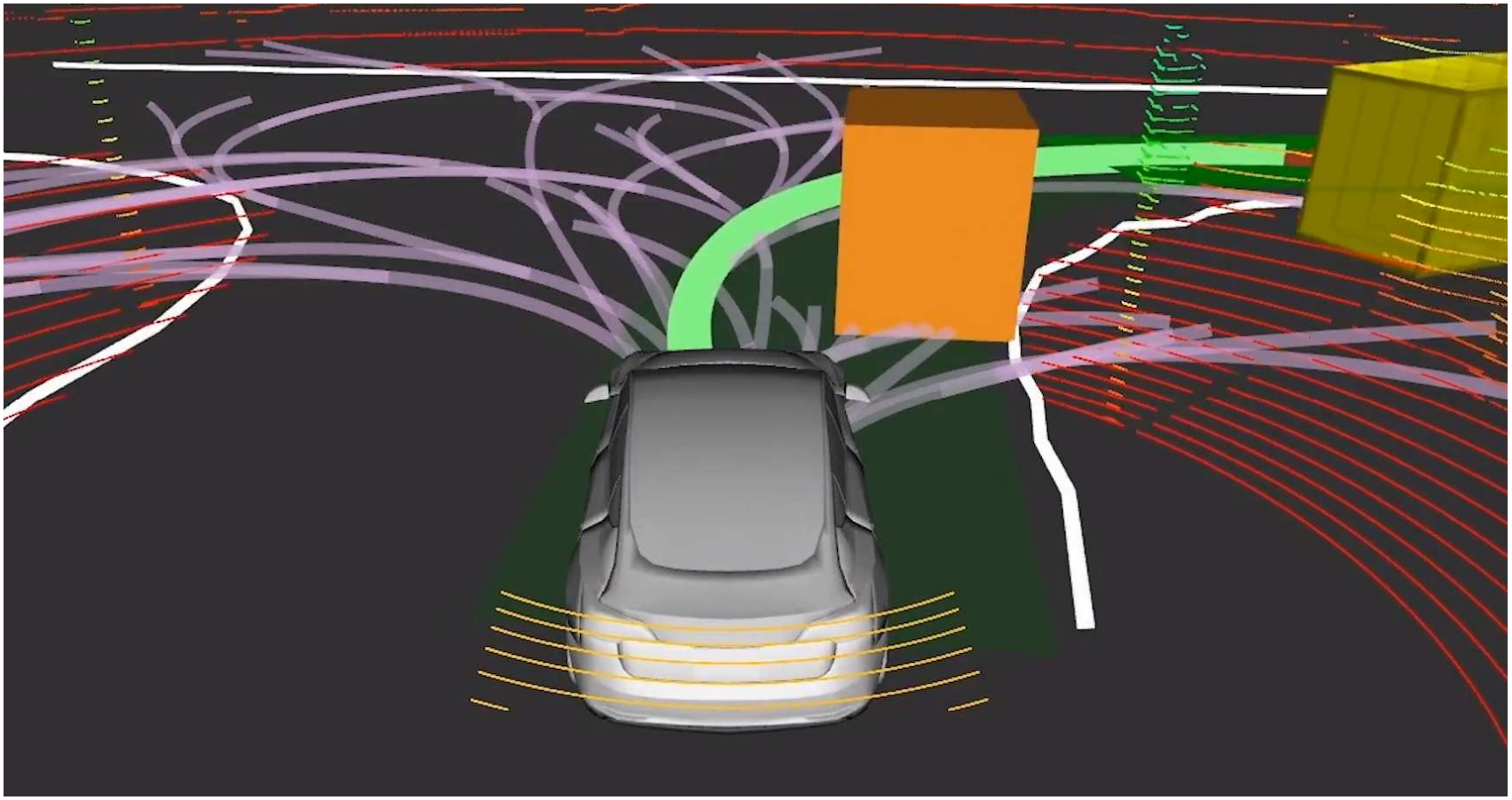
Custom hardware can lead to near-instantaneous collision checking!



5

Custom hardware can lead to near-instantaneous collision checking!

Custom hardware can lead to near-instantaneous collision checking!



5

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

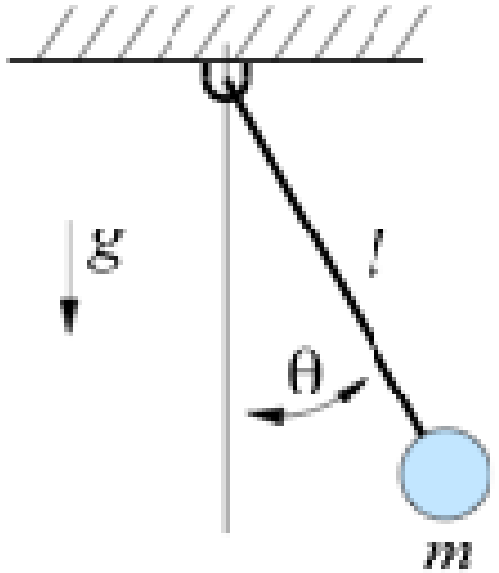
5

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"

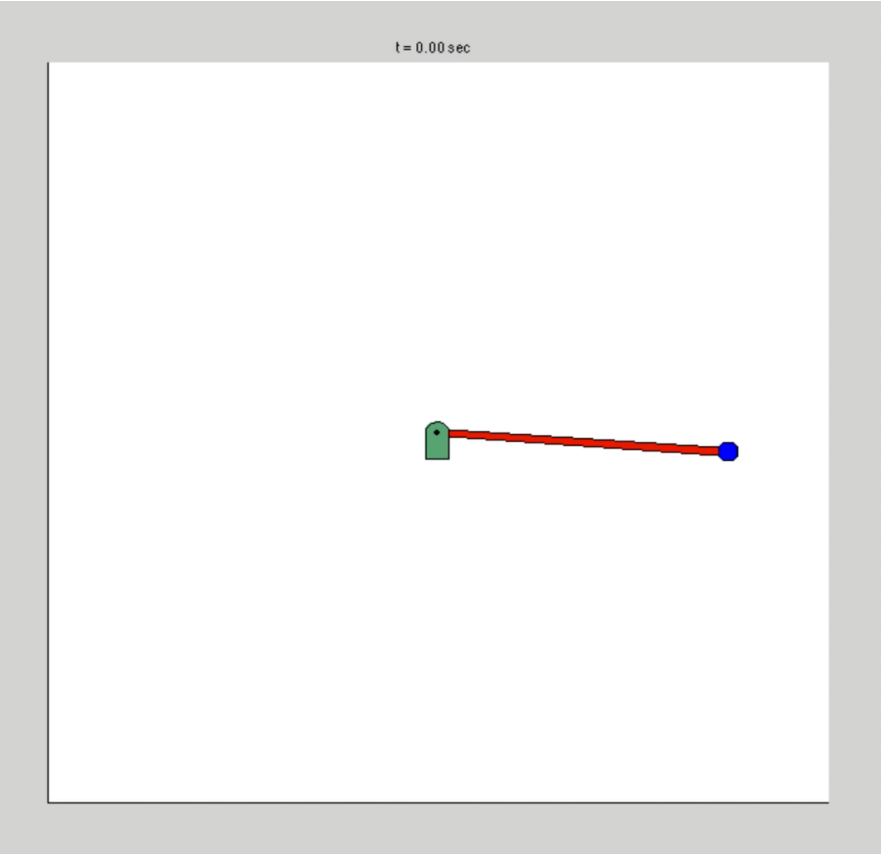


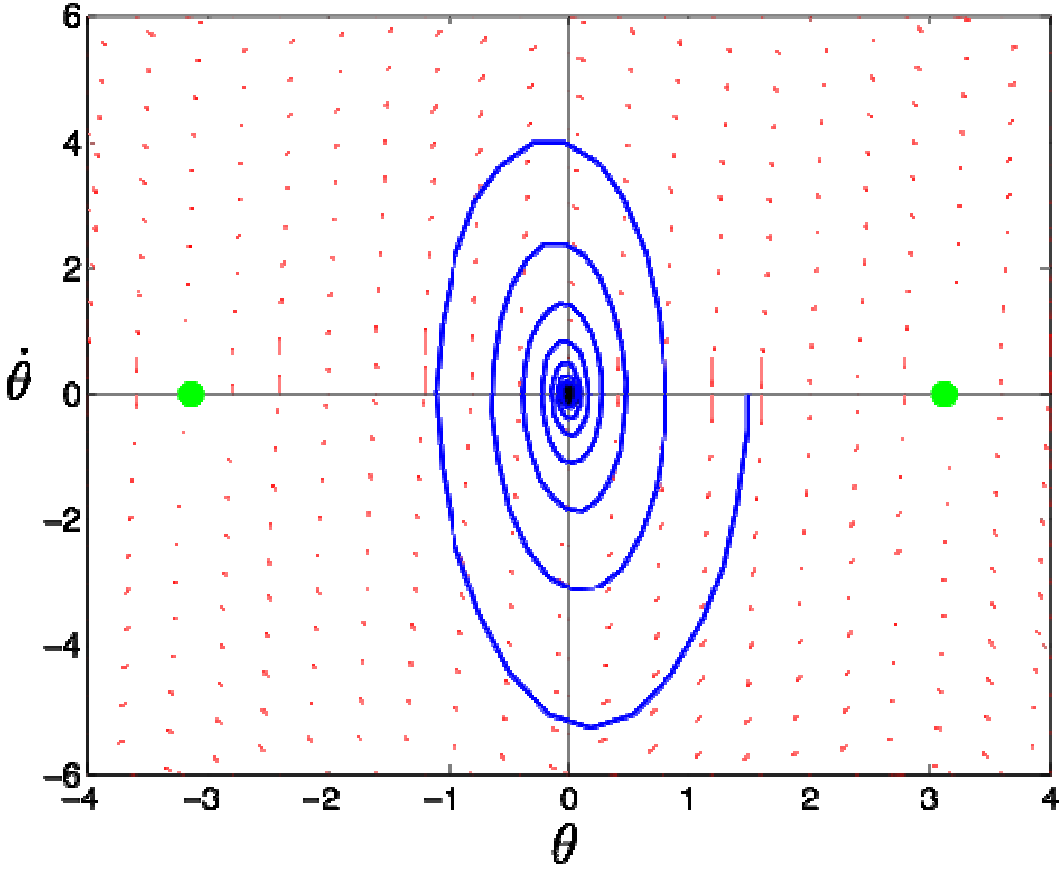
Dynamics (aka Physics)

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

5

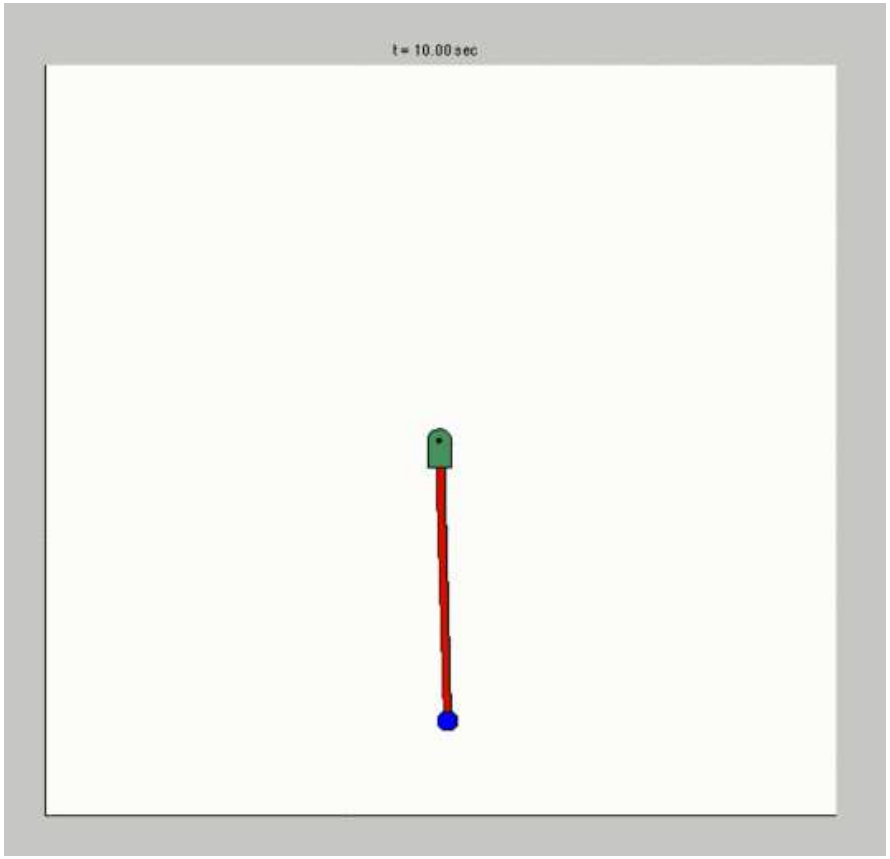
Ok so why can't robots use these awesome kinematic planning algorithms all the time and be better at life?!?

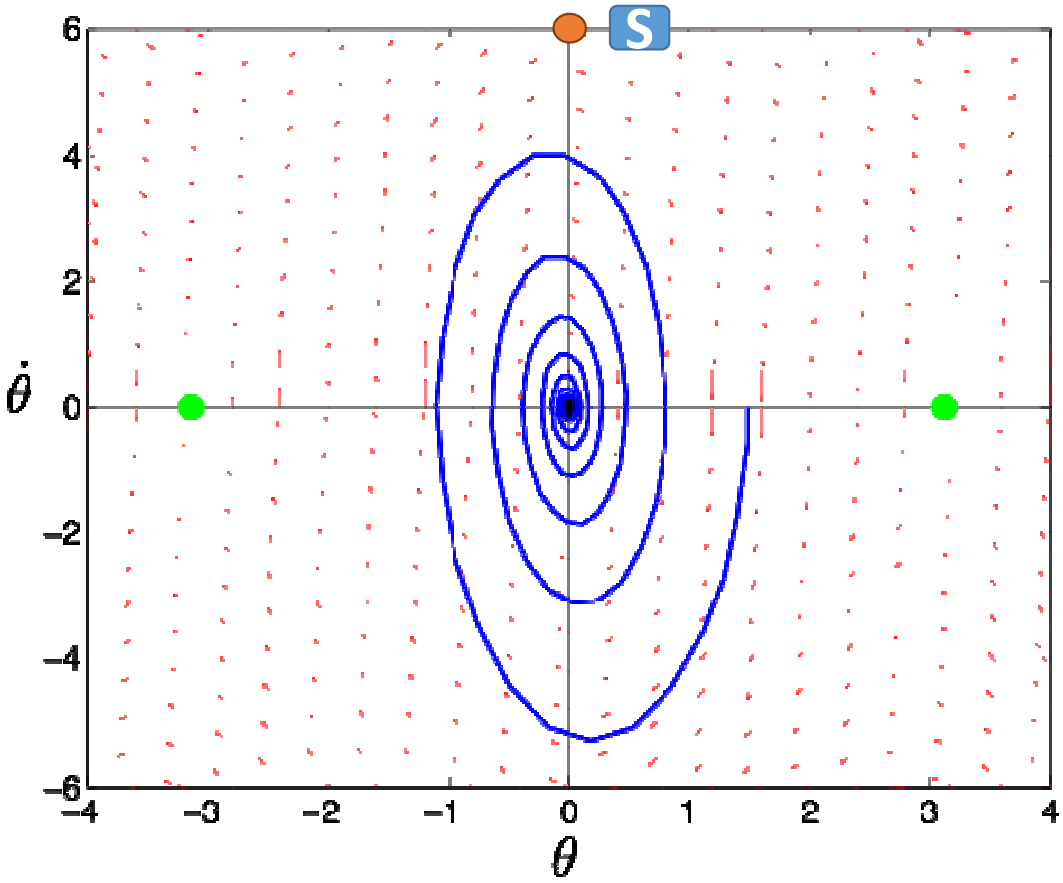




5

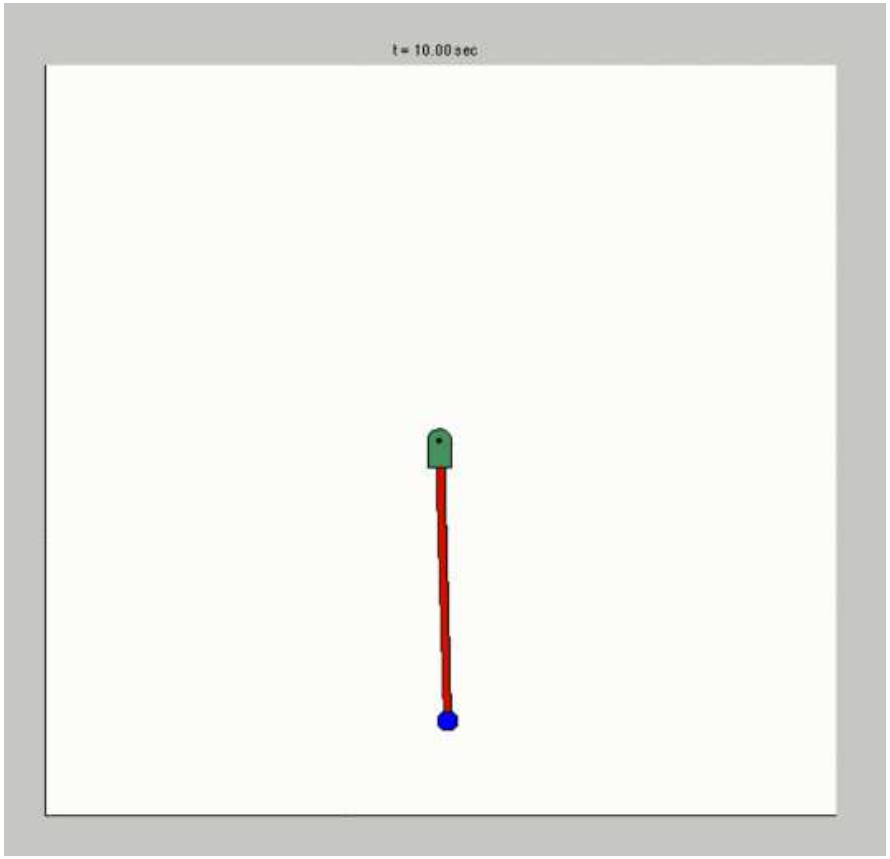
Ok so why can't robots use these awesome kinematic planning algorithms all the time and be better at life?!?

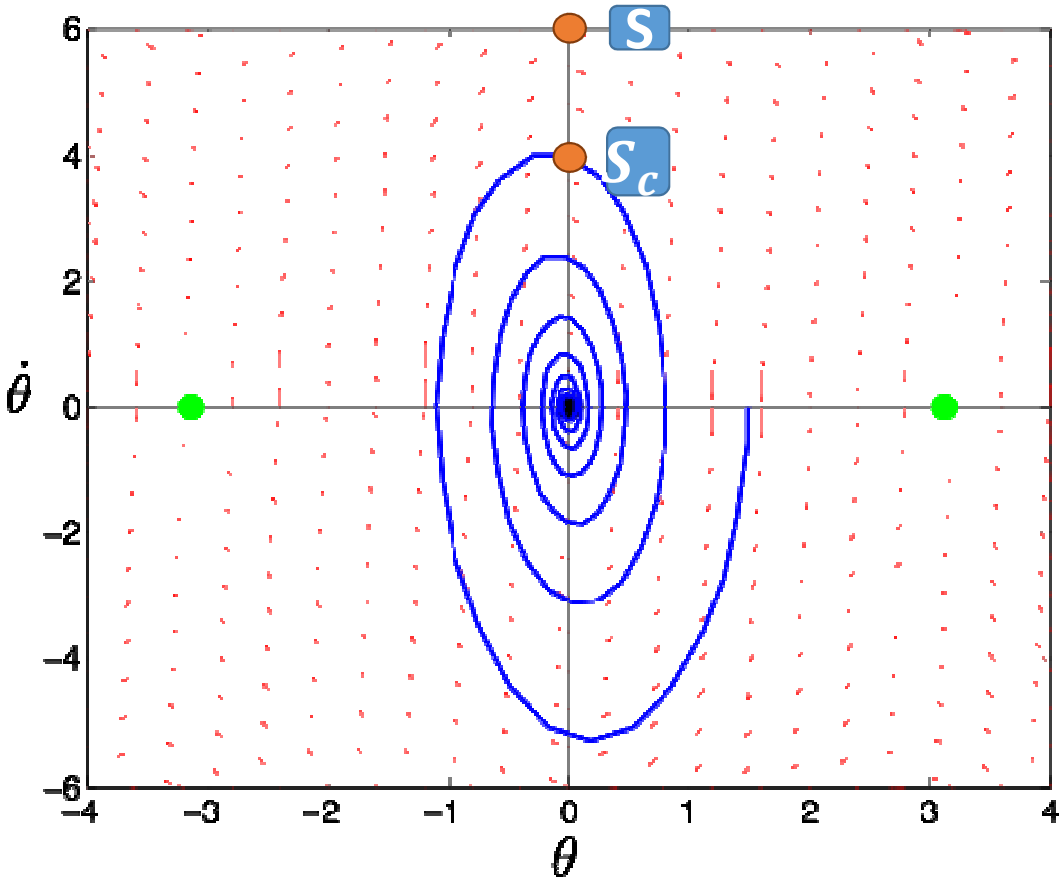




5

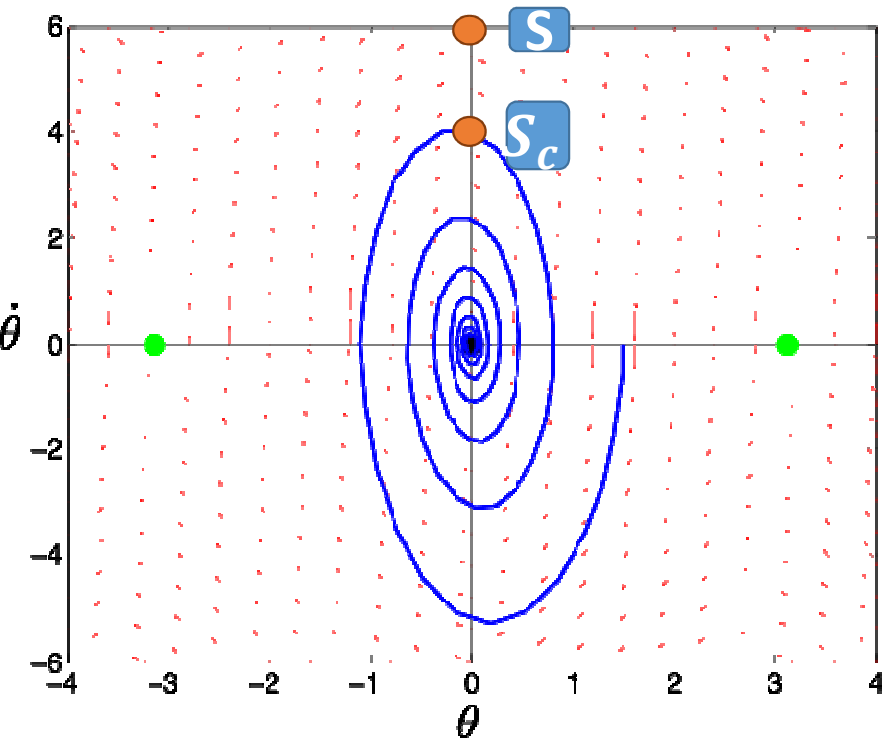
Ok so why can't robots use these awesome kinematic planning algorithms all the time and be better at life?!?





5

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



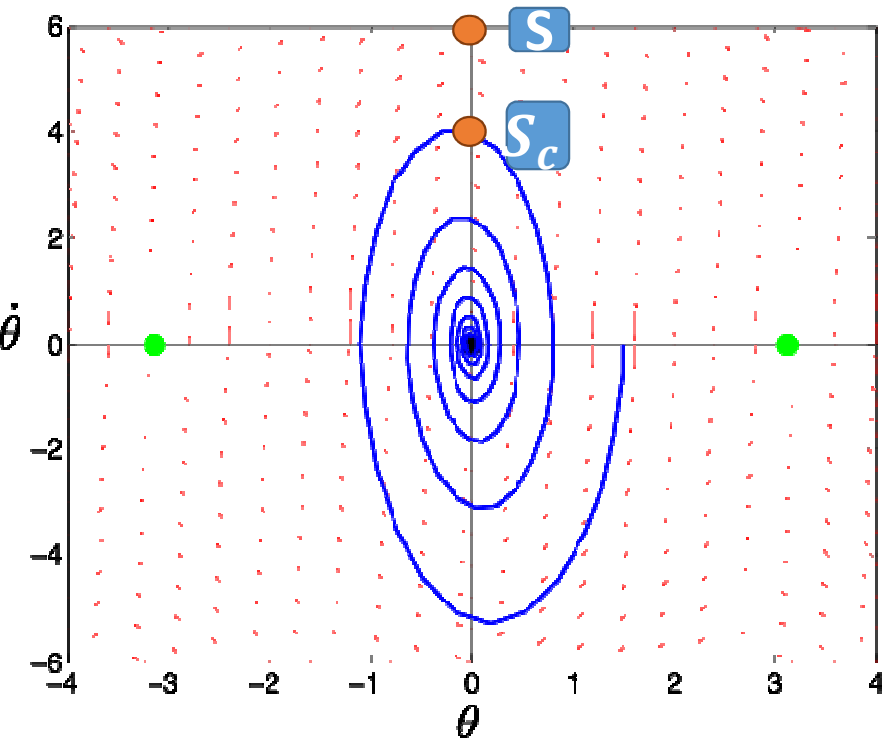
Challenges for Dynamic RRTs

The "extend" operation is complex!

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

5

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



Challenges for Dynamic RRTs

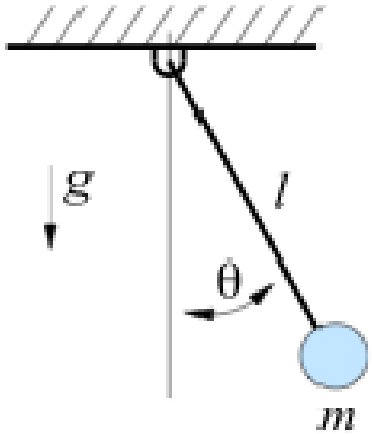
The "extend" operation is complex!

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

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

5

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

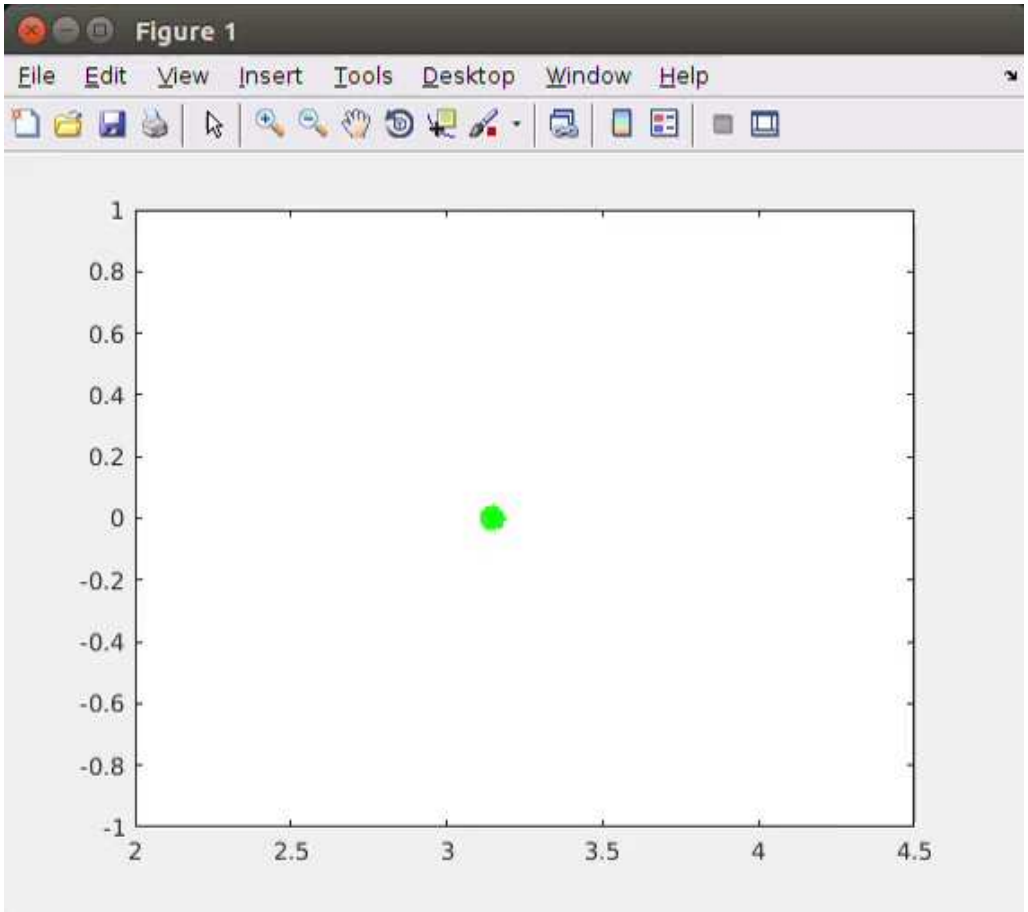


- States: $s = \{\theta, \dot{\theta}\}$ aka angle and angular velocity
- Actions: $a = \tau$ aka torque at joint
- Transitions: $s' = f(s, a)$ aka physics

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

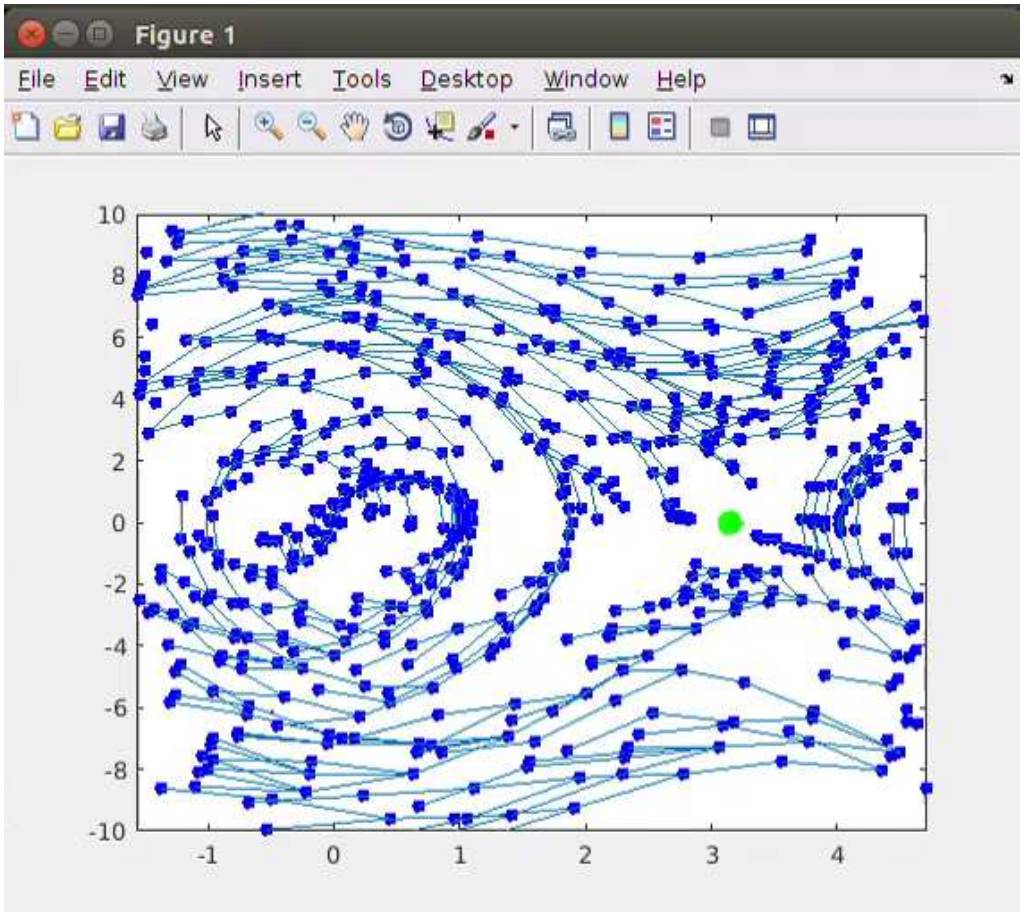
5

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



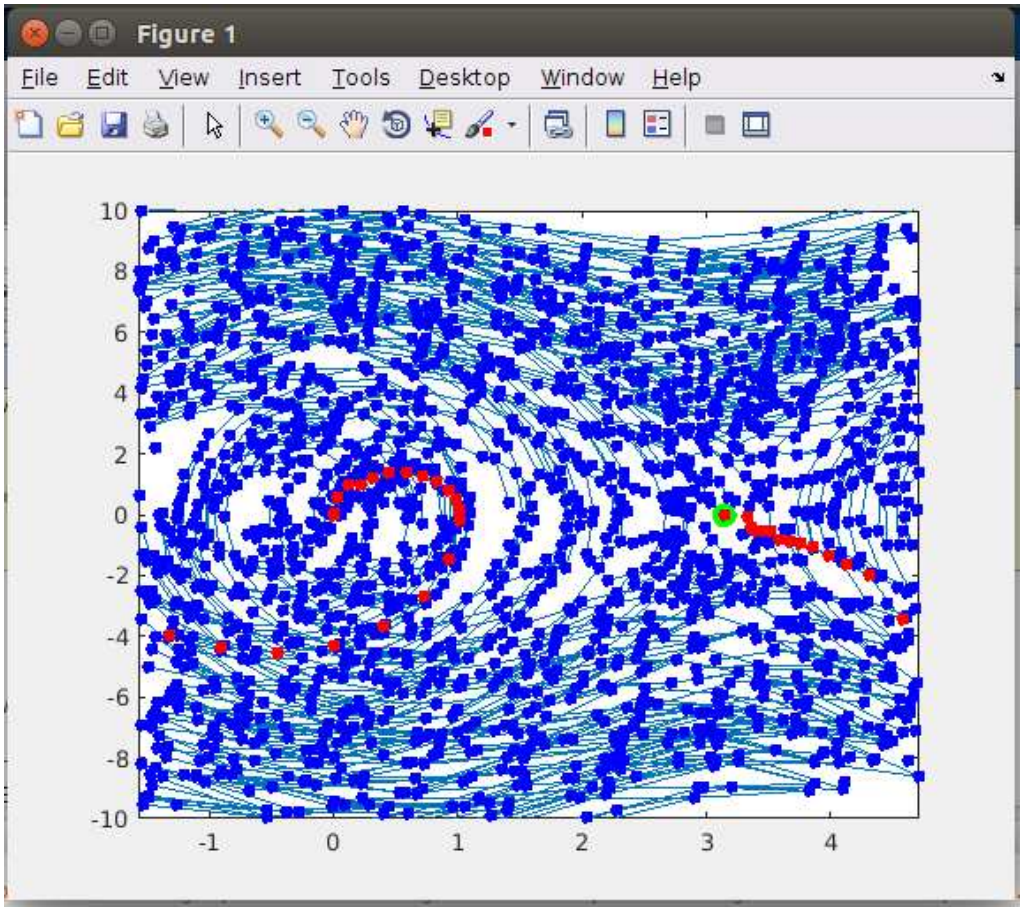
5

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



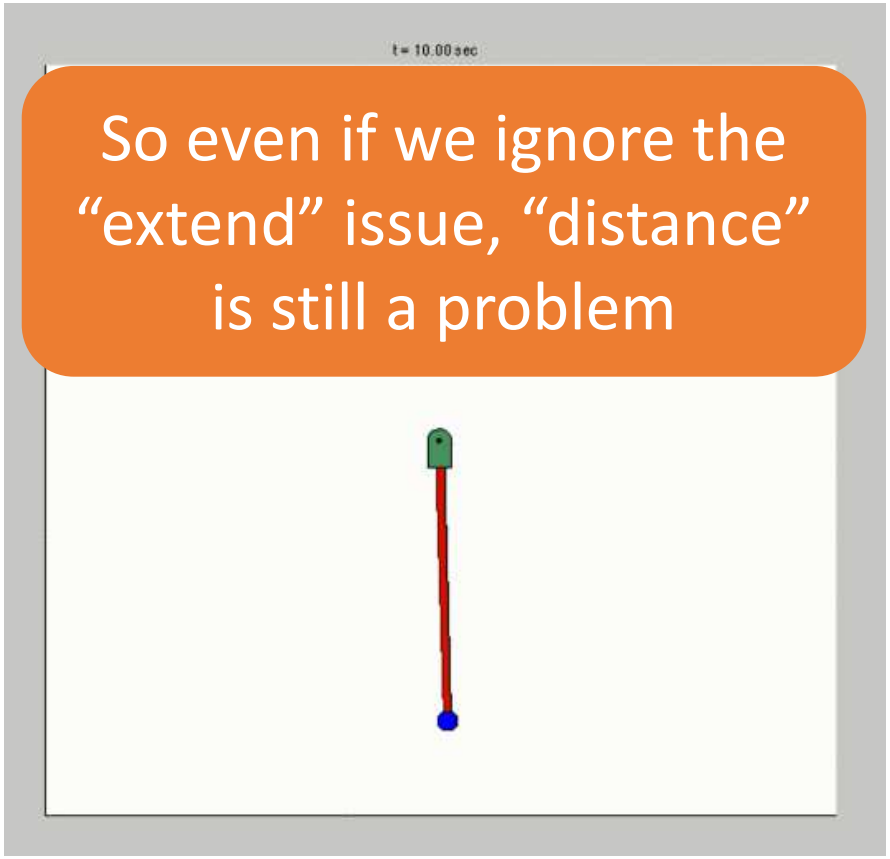
5

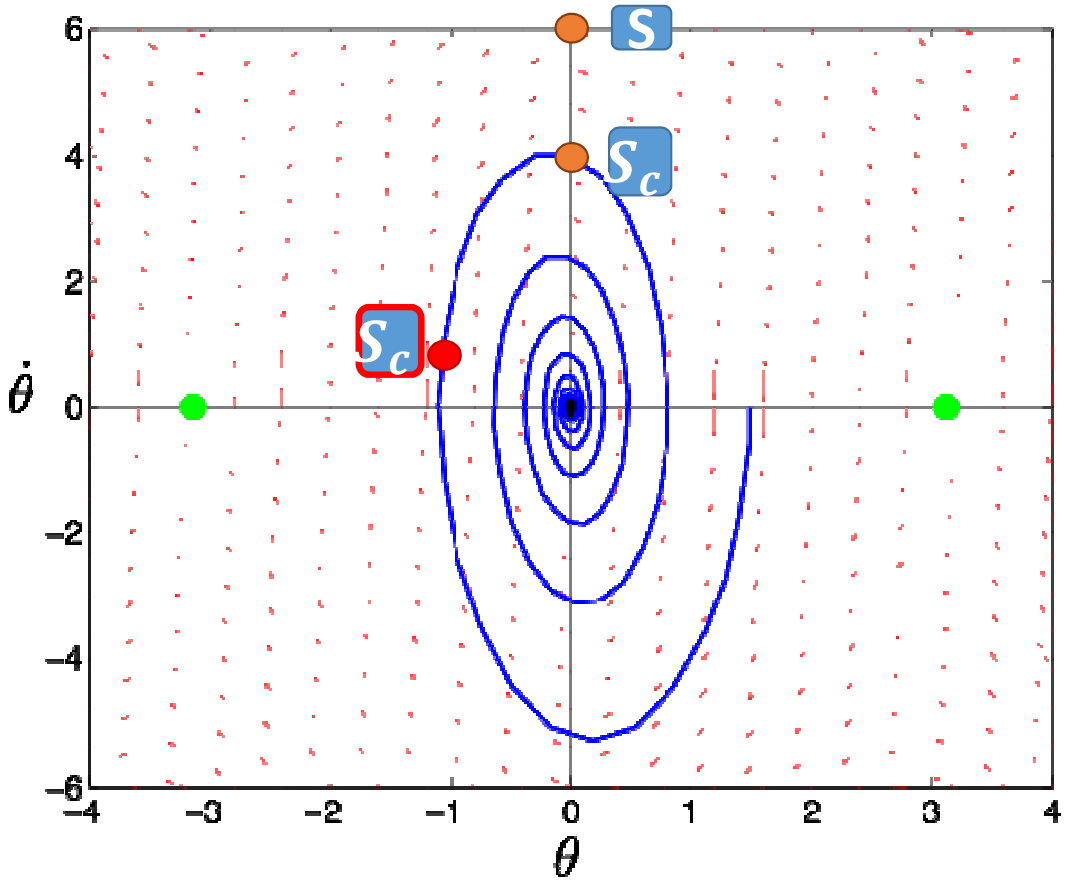
Ok so why can’t robots use these awesome kinematic planning algorithms all the time and be better at life?!?



5

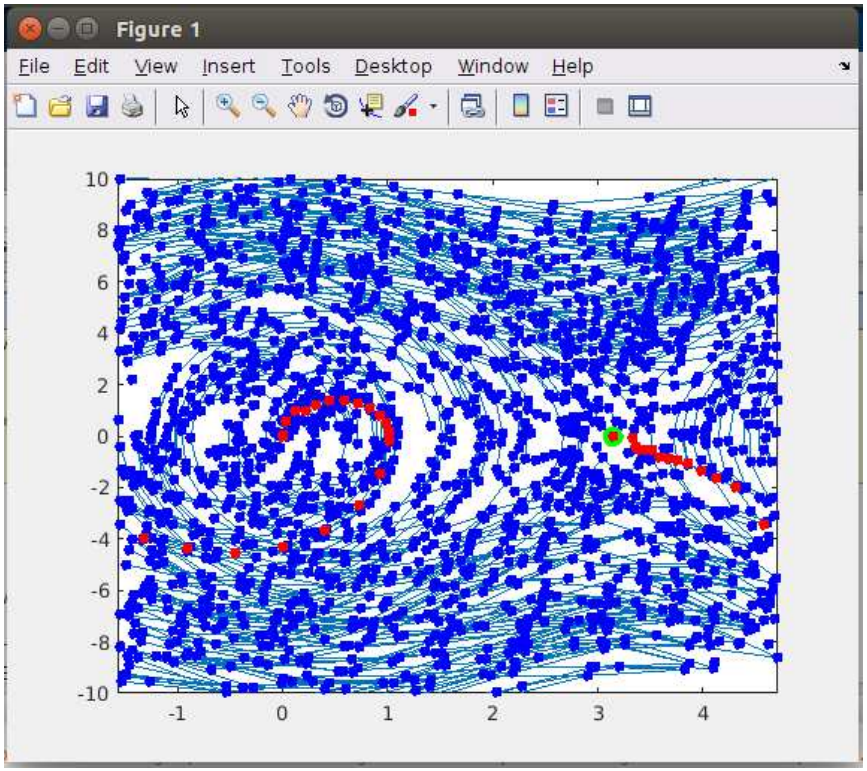
Ok so why can't robots use these awesome kinematic planning algorithms all the time and be better at life?!?





5

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 "extend" operation is complex!

- We need to solve a **boundary value problem** (find a path from sc to s such that it 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**

5 So what do we do?

5 So what do we do?

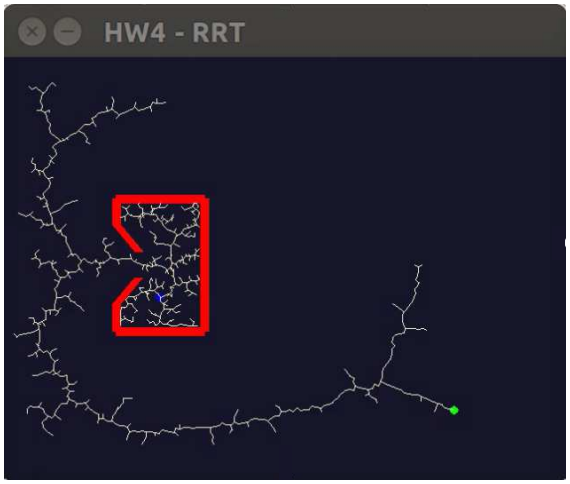
Give up and make the computer solve it for us?

5 So what do we do?

#Learning
#EfficientUseOfHumans

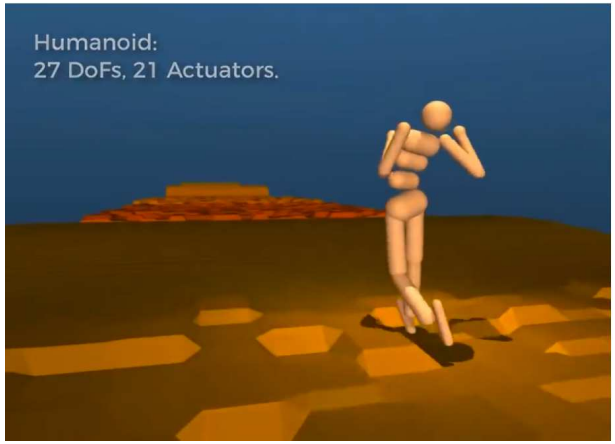
Give up and make the computer solve it for us?

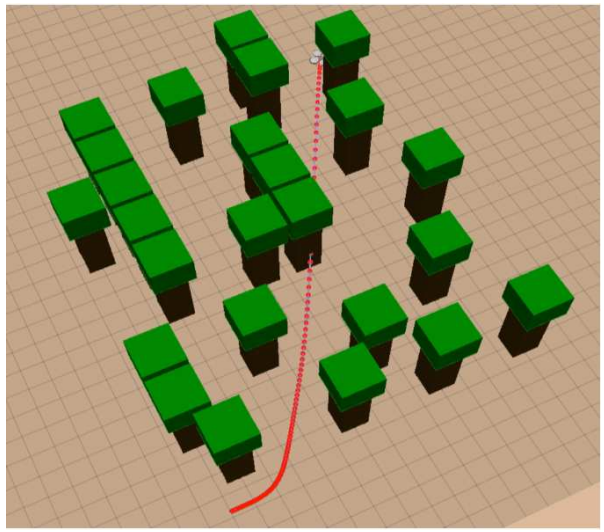
5 Planning in Configuration Space



Random Search

Machine Learning

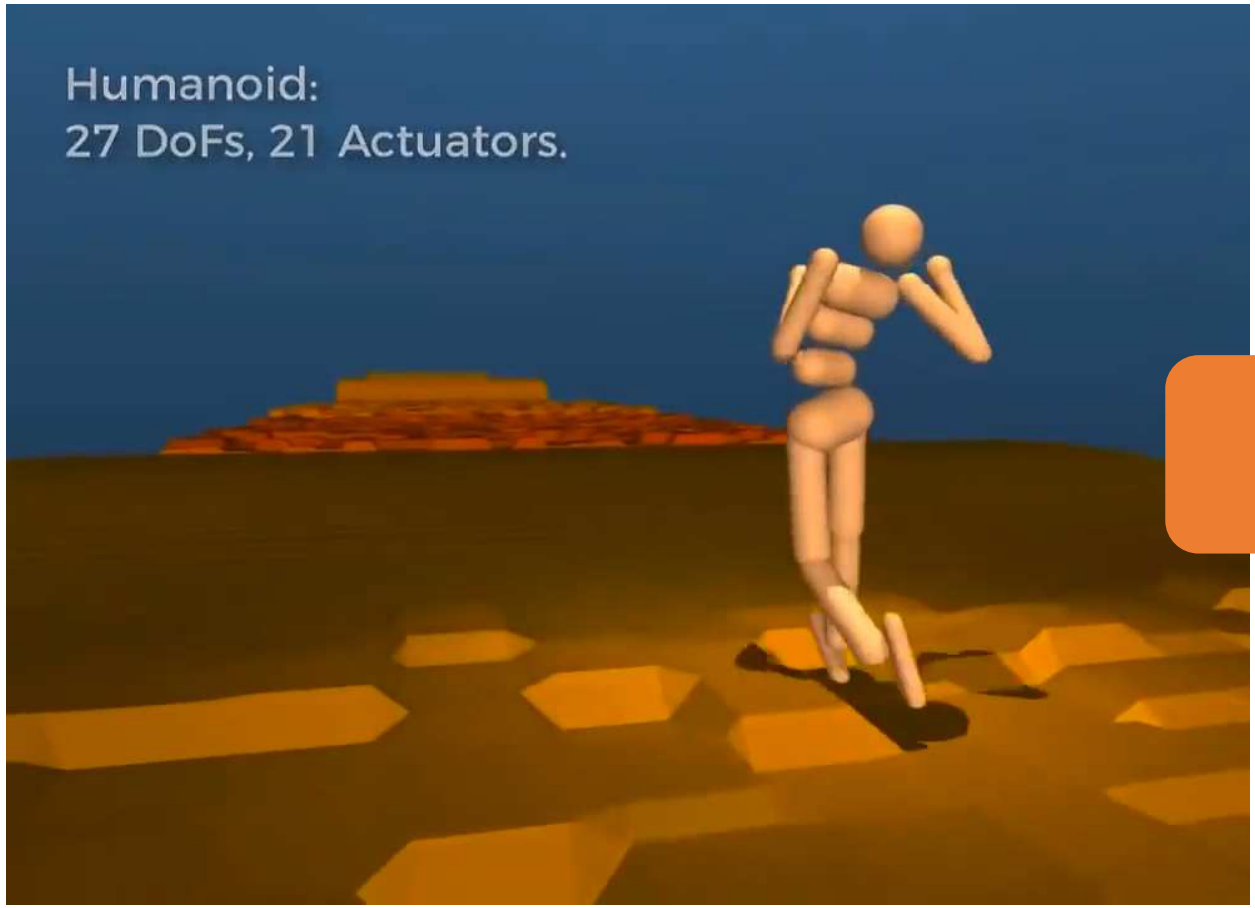




Local Search

5

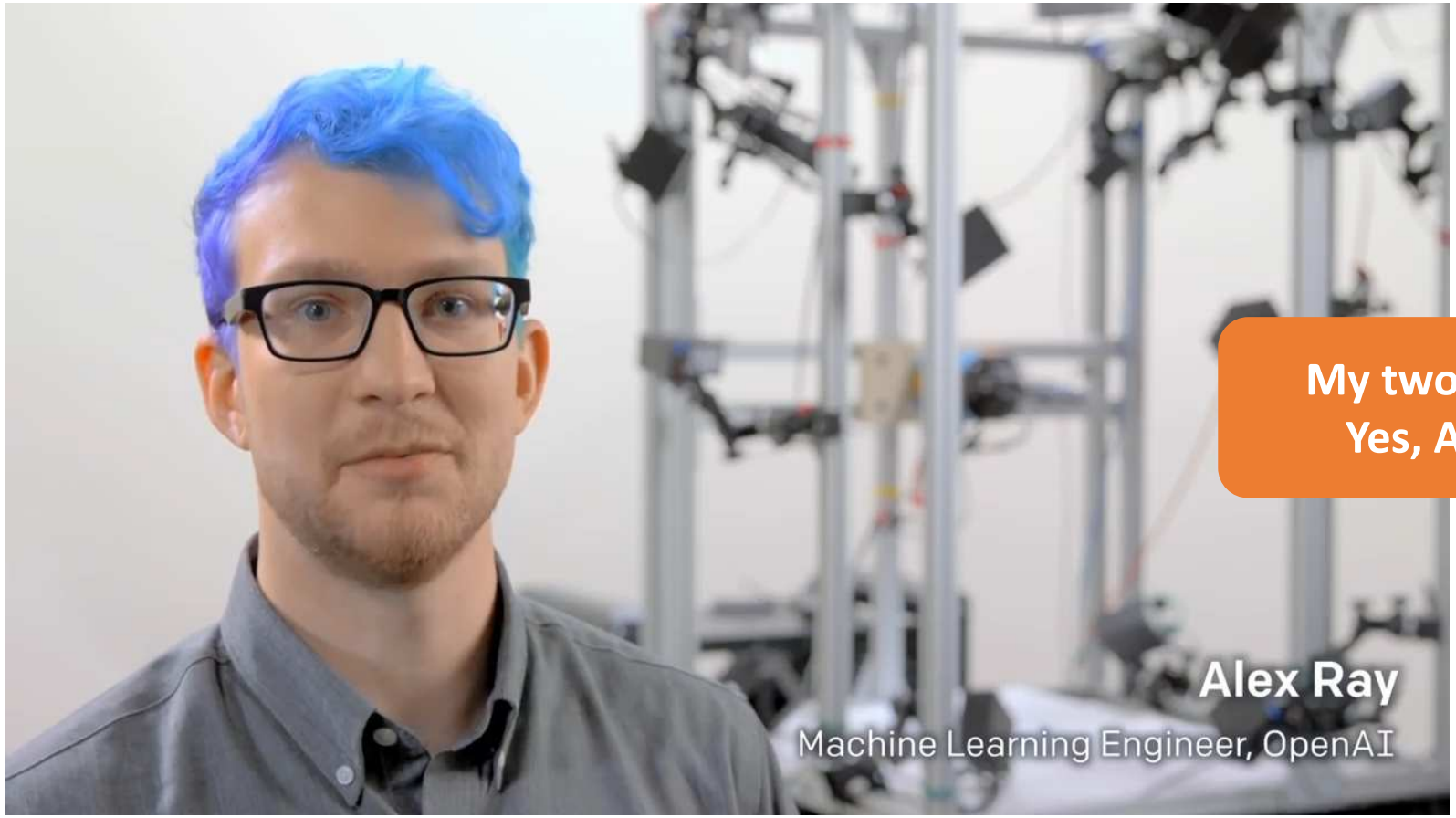
Guest Lecture in two weeks: Can I make the computer learn all of this for me automatically?



My two cents:
Yes, And…

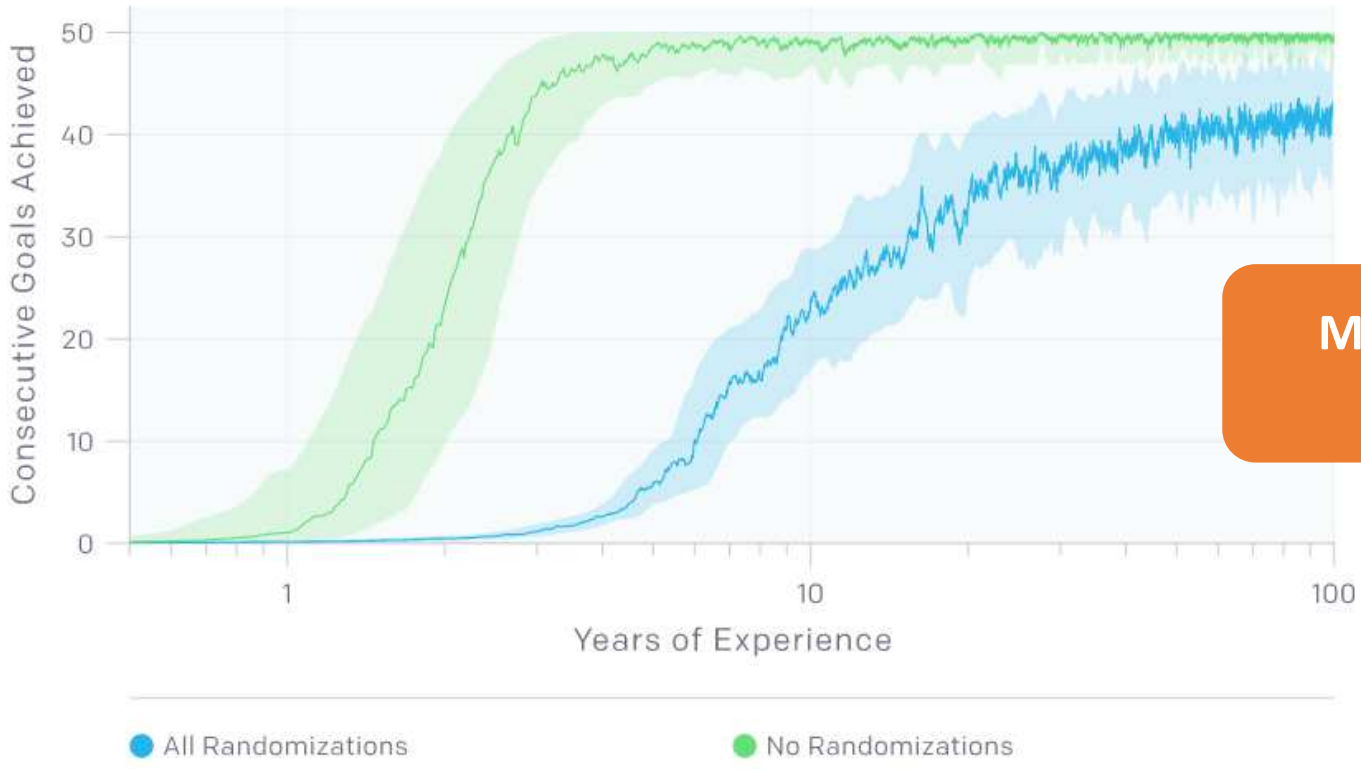
5

Guest Lecture in two weeks: Can I make the computer learn all of this for me automatically?



My two cents:
Yes, And…

5 Guest Lecture in two weeks: Can I make the computer learn all of this for me automatically?



My two cents:
Yes, And…

5 So what else can we do?

5 So what else can we do?

Lots of math!

5 So what else can we do?

Lots of math!

5 So what else can we do?

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

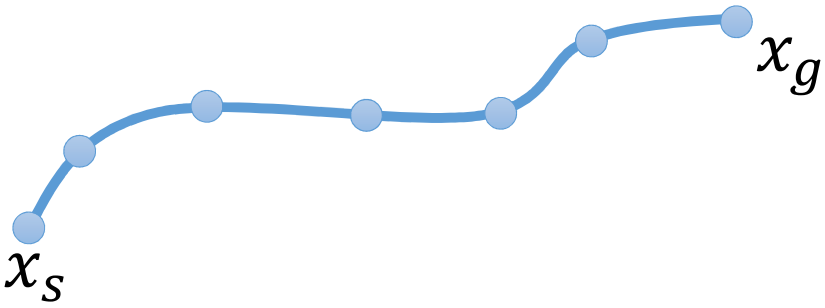
5 Optimization

We can write the planning problem down as an optimization problem!

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

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

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



5 Optimization

We can 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

5 Optimization

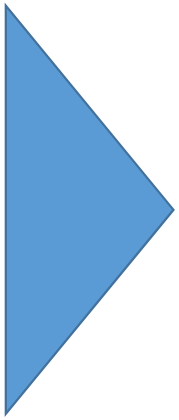
We can use Bellman updates to solve this:

- 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

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

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

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



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

$$V_{N-1}(s) = \min_a c(s_{N-1}, a_{N-1}) + V_N\big(f(s_{N-1}, a_{N-1})\big)$$

This leads to the classic ***Value Iteration*** algorithm

5 Optimization

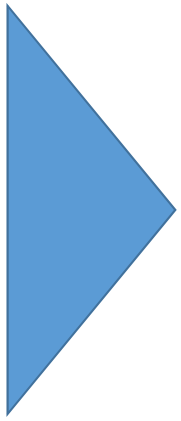
We can use Bellman updates to solve this:

- 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

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

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

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



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

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

This leads to the classic ***Value Iteration*** algorithm

5 Optimization

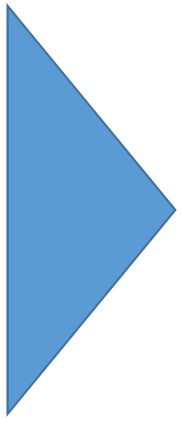
We can use Bellman updates to solve this

- 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

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

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

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



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

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

Sadly again the complexity scales with $d^{|S|=|A|}$ and those can get **HUGE** fast! This is the **"curse of dimensionality"** again

5 Optimization

Lets lower our expectations!
#localOptima #efficientUseOfComputers

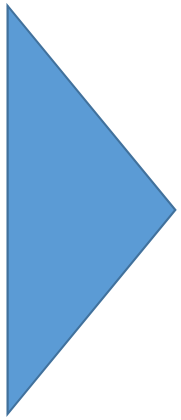
We can use Bellman updates to solve this

- 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

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

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

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

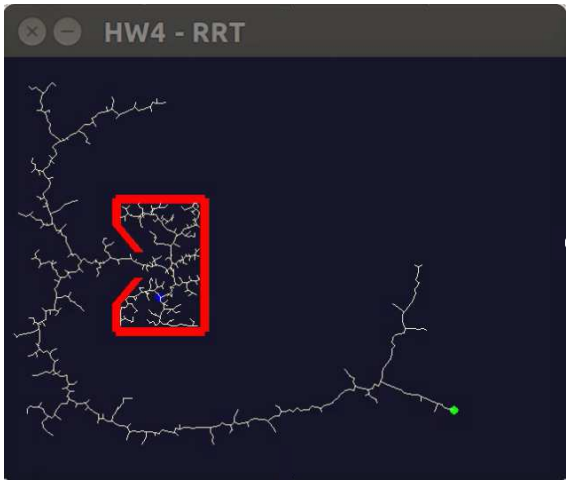


$$V_N(s_N) = c(s_N, a_N)$$

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

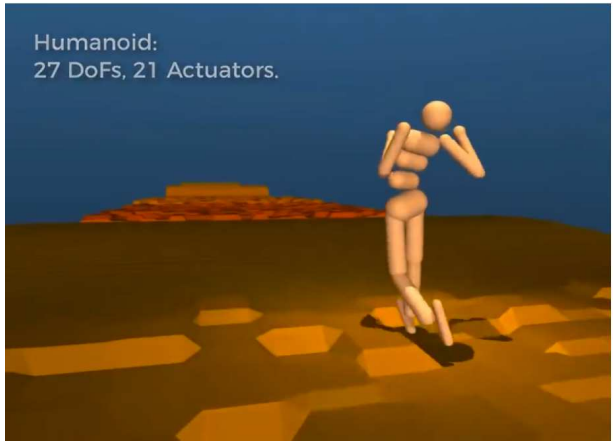
Sadly again the complexity scales with $d^{|S|=|A|}$ and those can get **HUGE** fast! This is the **"curse of dimensionality"** again

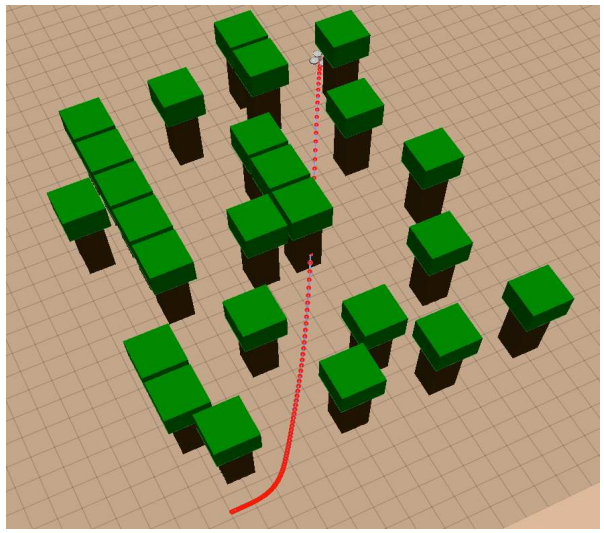
5 Planning in Configuration Space



Random Search

Machine Learning





Local Search

5 Trajectory Optimization

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

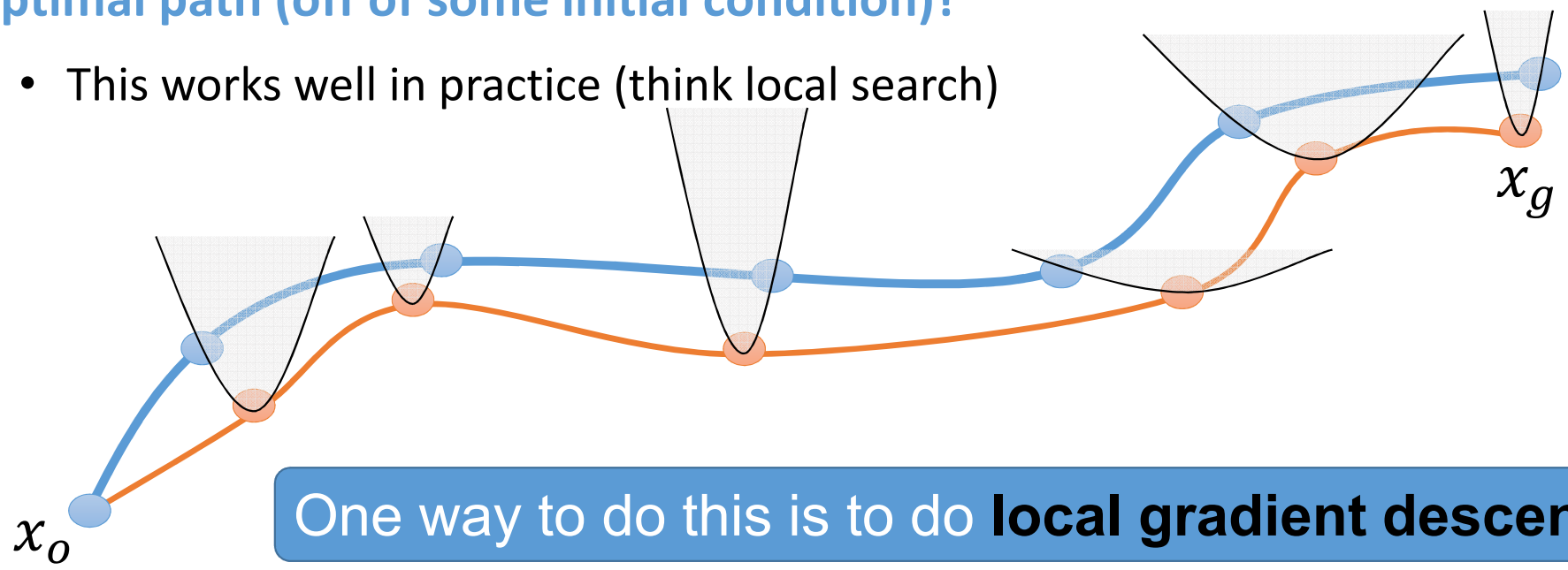
- This works well in practice (think local search)

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

5 Trajectory Optimization

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

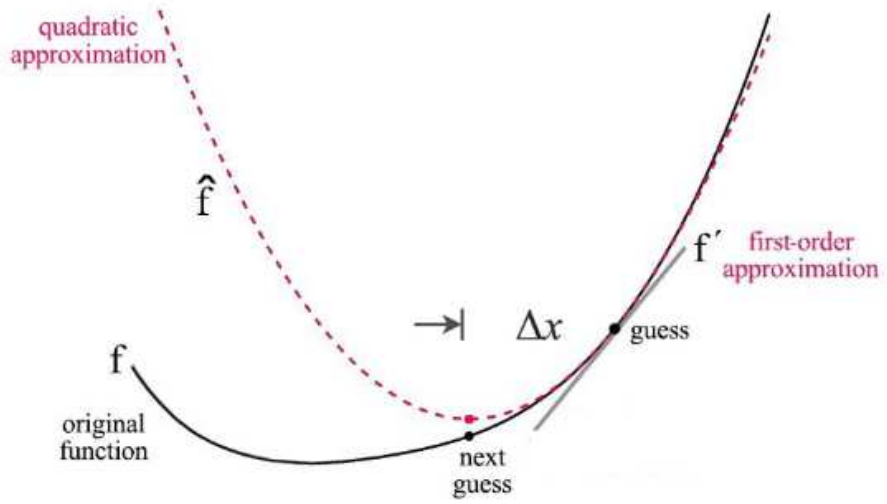
- This works well in practice (think local search)



One way to do this is to do **local gradient descent**

5 Trajectory Optimization

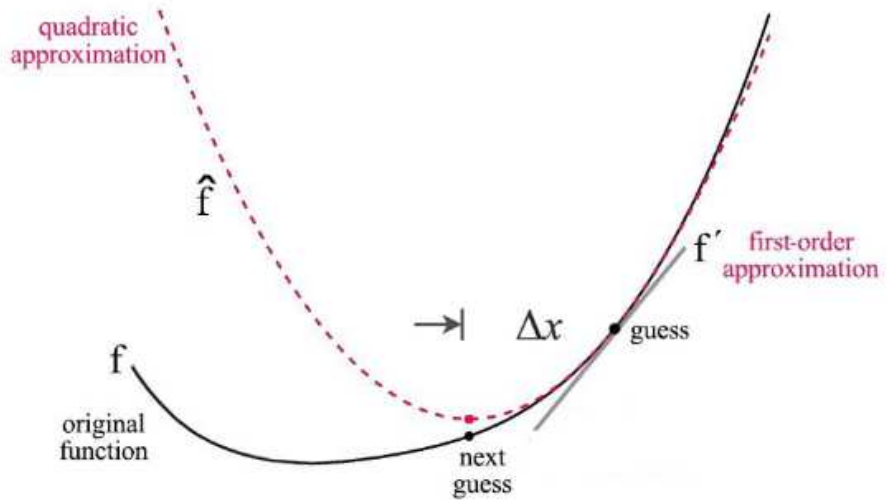
I'm drawing small quadratic bowls because most (if not all) of the practical algorithms make **linear and quadratic approximations** of the nonlinear functions allowing for efficient gradient descent



5 Trajectory Optimization

And convex optimization tells us how to descend to the minima of a quadratic function

I'm drawing small quadratic bowls because most (if not all) of the practical algorithms make **linear and quadratic approximations** of the nonlinear functions allowing for efficient gradient descent



5 Trajectory Optimization

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

- DDP, SQP, Interior-Point Methods, Trust-Region Methods, Stochastic Gradient Descent Methods, etc.

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

- SNOPT, IPOPT, NLOPT, fmincon (MATLAB), etc.
- **Most people use off the shelf solvers!**

5 Trajectory Optimization

So trajectory optimization solves everything right?

- Can handle full robot **dynamics**
- No need for distance metrics
- Can use **off the shelf solvers** reducing the coding burden
- Finds a **locally optimal** solution – no weird paths coming out!
  - Extra motions are “optimized away”

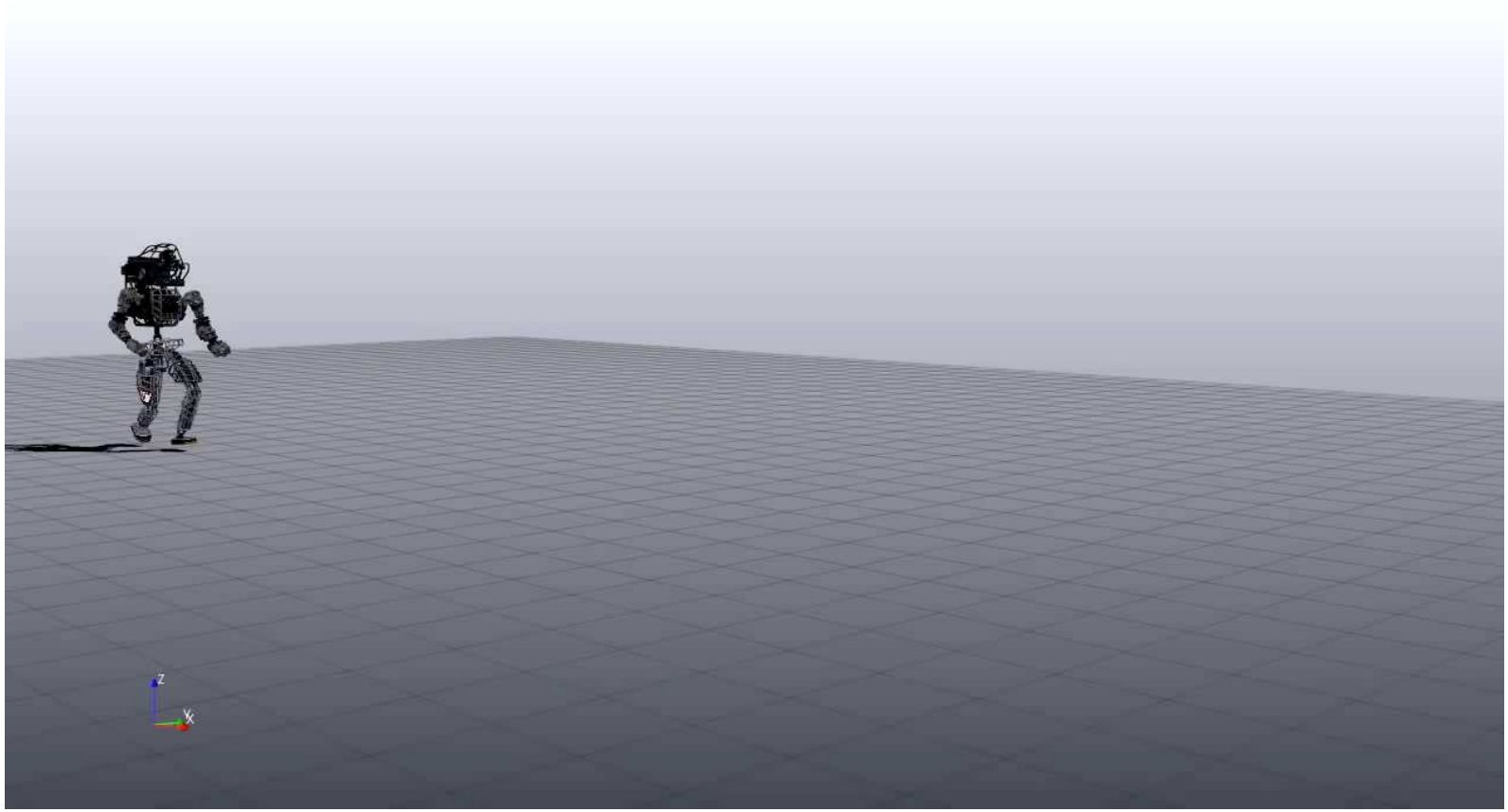
5 Trajectory Optimization

So trajectory optimization solves everything right?

- Can handle full robot **dynamics**
- No need for distance metrics
- Can use **off the shelf solvers** reducing the coding burden
- Finds a **locally optimal** solution – no weird paths coming out!
  - Extra motions are “optimized away”

And optimal motions often look bio-inspired as nature generally uses optimally efficient motions!

5 Atlas 1.0 Trajectory Optimization



5 Trajectory Optimization

So trajectory optimization solves everything right?

- Can handle full robot **dynamics**
- No need for distance metrics
- Can use **off the shelf solvers** reducing the coding burden
- 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**

5 Trajectory Optimization

So trajectory optimization solves everything right?

- Can handle full robot **dynamics**
- No need for distance metrics
- Can use **off the shelf solvers** reducing the coding burden
- 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**

Lets dive a little deeper into solvers!

5 There are two popular classes of solvers

	Shooting Methods (e.g., DDP, iLQR)	Direct Methods (e.g., DIRTRAN using SQP or IP)
Pros	• Known fast	• Easy to add constraints (e.g., torque limits, obstacle avoidance) • Easy to leverage **off the shelf solvers** (e.g., SNOPT, IPOPT)
Cons	• Hard to add constraints (e.g., torque limits, obstacle avoidance) • Generally people code it themselves	• Considered slow

5 There are two popular classes of solvers

Pros

Cons

Technical note: DDP reduces to a specific factorization of the **KKT matrix** solve in a direct method to exploit sparsity!

$$\begin{aligned} \text{minimize} \quad & (1/2)x^T P x + q^T x + r \\ \text{subject to} \quad & Ax = b \end{aligned}$$

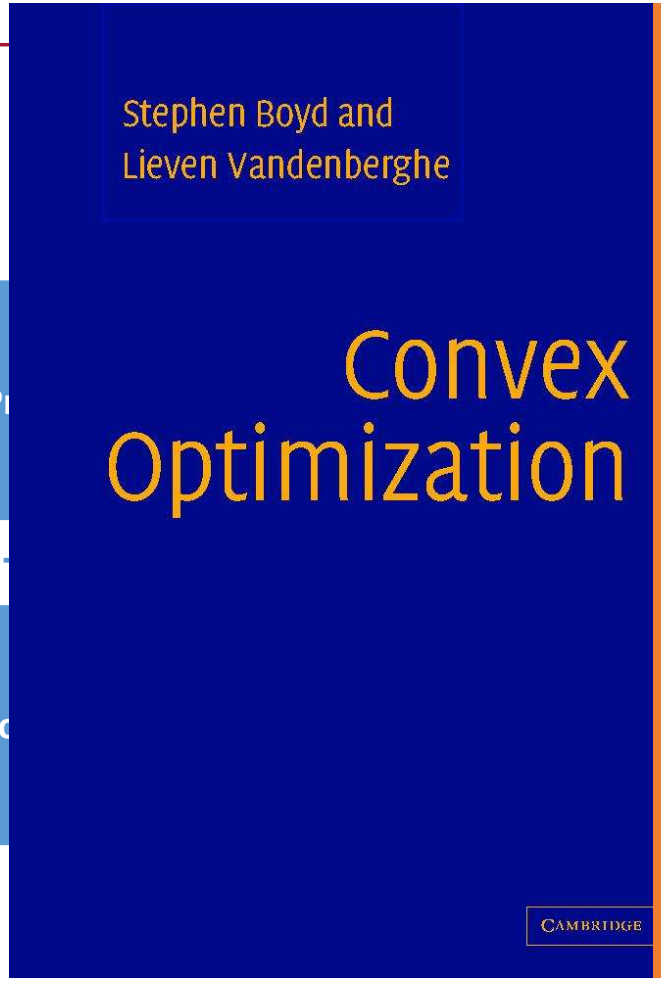
$$\begin{bmatrix} P & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} x^\star \\ \nu^\star \end{bmatrix} = \begin{bmatrix} -q \\ b \end{bmatrix}$$

5 There are two popular classes of solvers

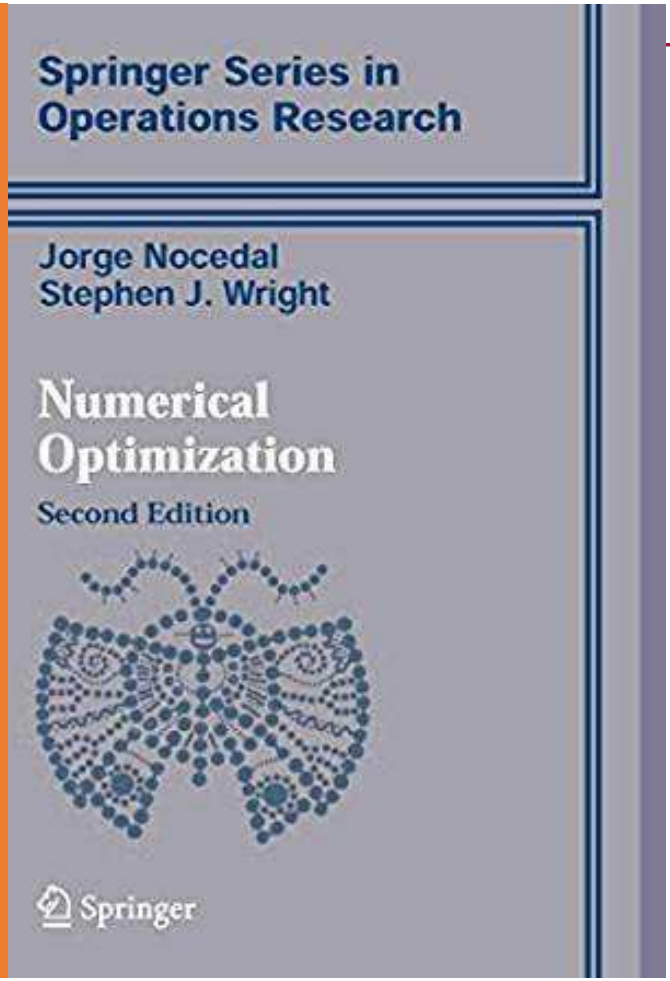
	Shooting M (e.g., DDP, i	Direct Methods (e.g., DIRTRAN using SQP or IP)
Pros	• Known fast	asy to add constraints (e.g., orque limits, obstacle avoidance) asy to leverage **off the shelf olvers** (e.g., SNOPT, IPOPT)
Cons	• Hard to add constra torque limits, obsta • Generally people c themselves	onsidered slow

I'll dig a little deeper / explain this more in 2 weeks when I present my research on parallel shooting methods

5 There are two popular classes of solvers



But/and these are two great textbooks if you want to learn more about the math!



5 Practical Challenges for Trajectory Optimization: Robustness

Manchester and Kuindersma 2017
Plancher and Kuindersma 2018

1. Solvers are (numerically) sensitive to:
   - Cost function designs and dynamic range
   - Regularization scheme

2. Solutions are sensitive to:
   - Initial state and input trajectories
   - Perturbations (solutions are often on constraint boundaries)

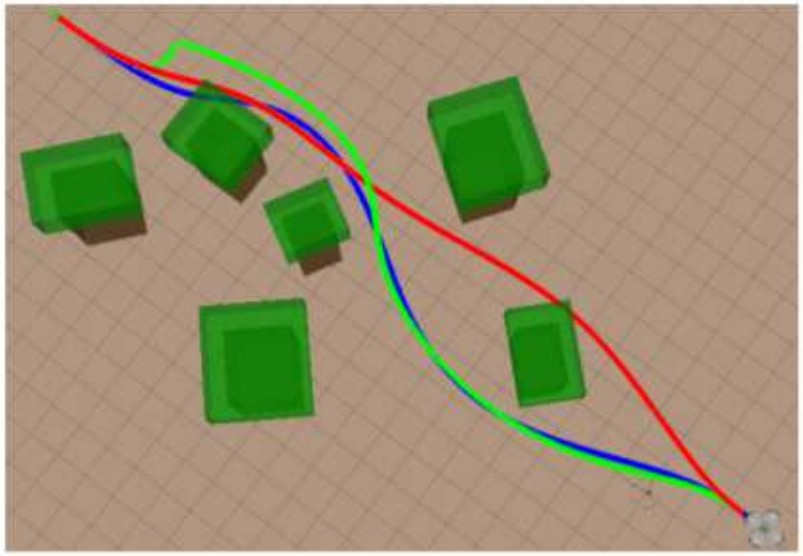


Fig. 4. DIRTRAN (red), DIRTREL-1 (blue), and DIRTREL-2 (green) quadrotor trajectories.

5 Practical Challenges for Trajectory Optimization: Contact

Tedrake Underactuated

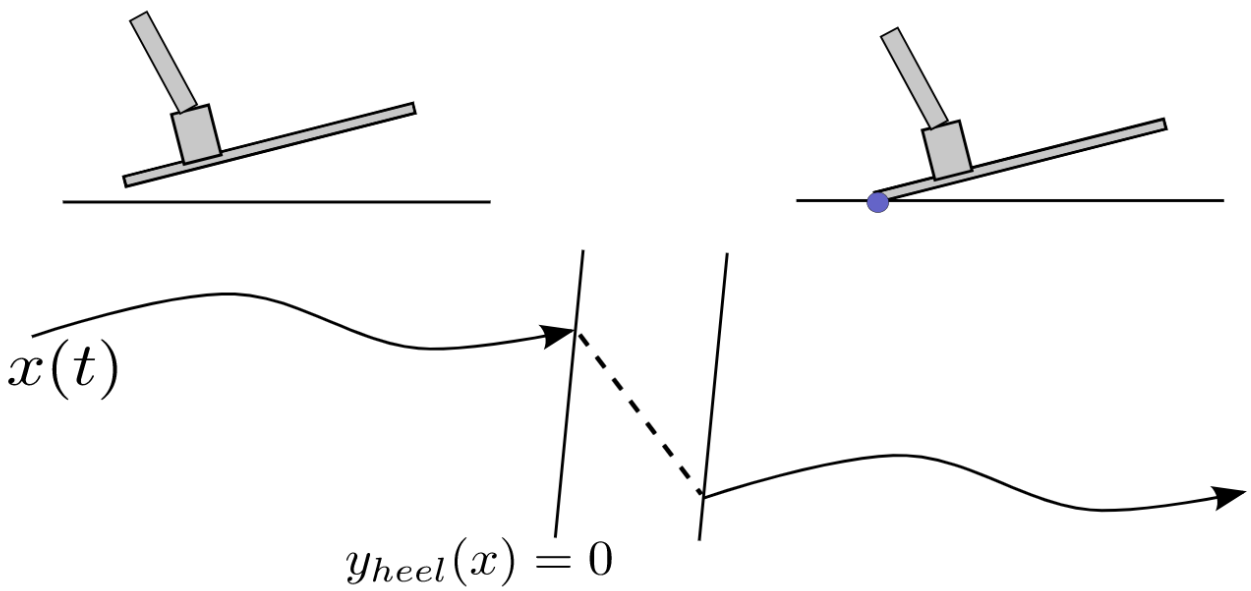
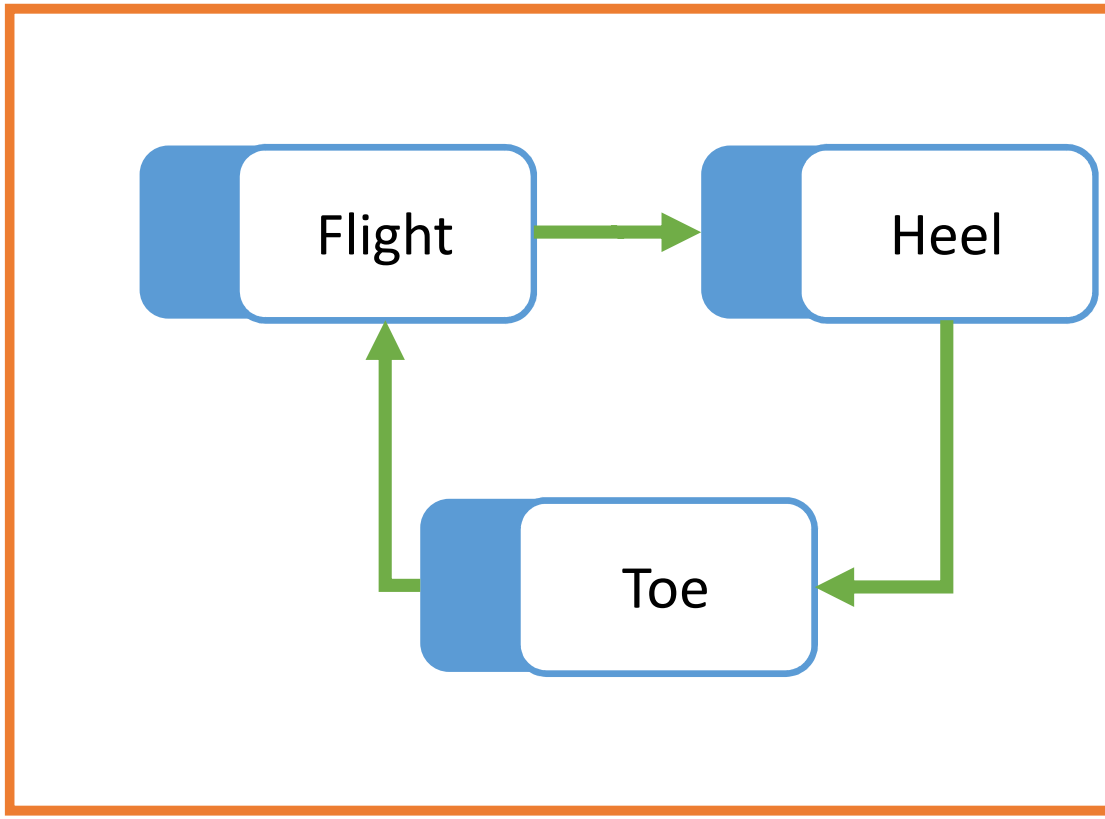


Figure 17.1 - Modeling contact as a hybrid system.

The physics equations are fundamentally different when an object makes or breaks contact

5 Practical Challenges for Trajectory Optimization: Contact



For walking these **hybrid modes** form a cyclic graph

If we **pre-specify** the mode sequence and timing we can use our algorithms as before

5 Practical Challenges for Trajectory Optimization: Contact

Manchester and Kuindersma 2017

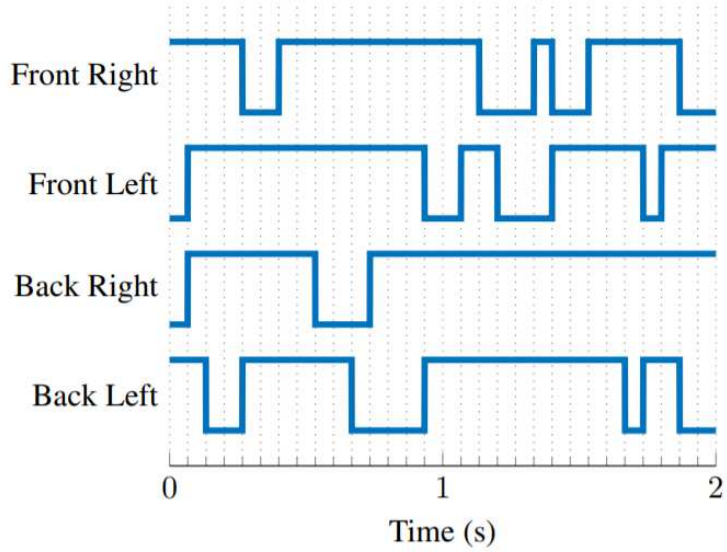
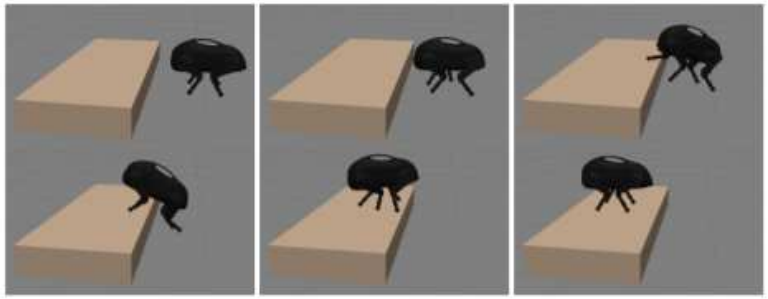


Figure 4. Contact mode sequence for each foot from LittleDog step-climbing example.



But for **complex actions** these modes start to become **hard to pre-specify**

5 Practical Challenges for Trajectory Optimization: Contact

Doshi, et. al. 2018

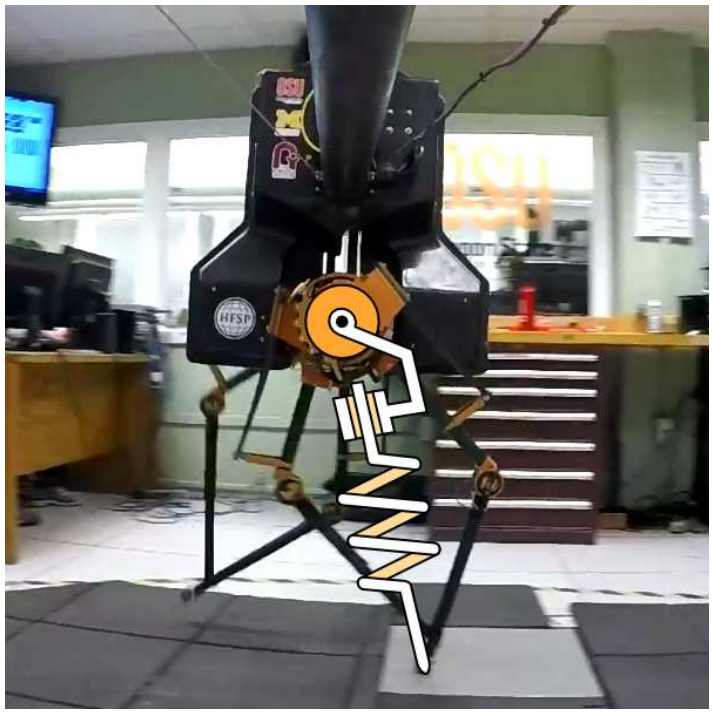
Contact-Implicit Trajectory Optimization includes the contact timings and mode transitions as state variables

2Hz Gaits
(real-time)

But these approaches are computationally very expensive (read **offline**) as the number of modes explodes combinatorically with the number of contact points (Mixed-Integer Programming)!

5 Practical Challenges for Trajectory Optimization: Contact

One approach to avoid solving these large hard problems is to solve the problem on **simpler models** of the system



5 Practical Challenges for Trajectory Optimization: Contact

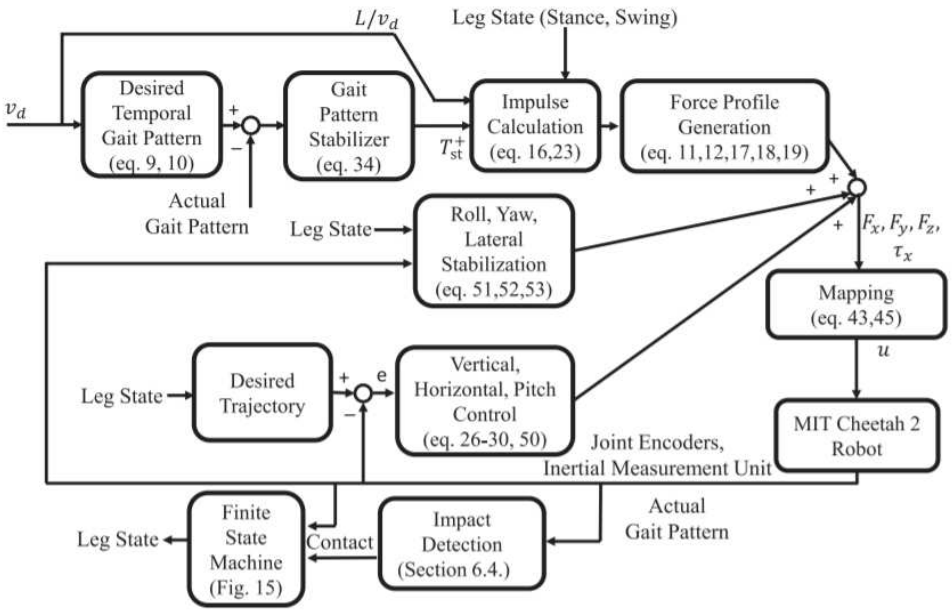
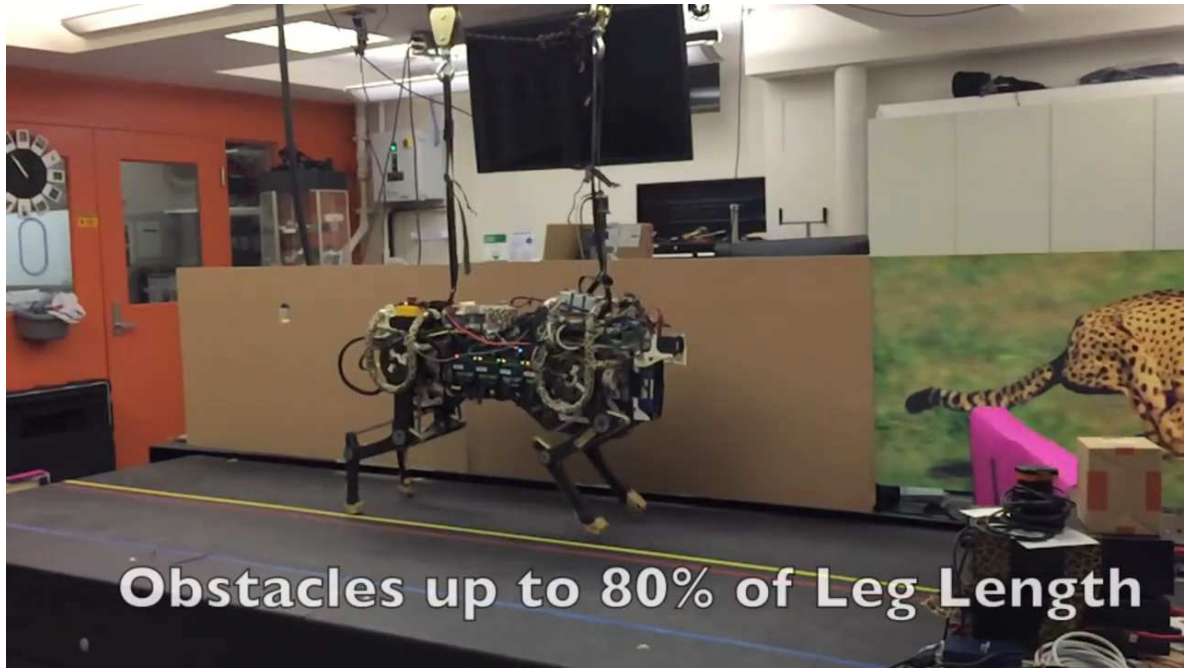


Fig. 11. Overall control system diagram.

And then **combine solutions** to these **(conservative)** simpler problems

5 Practical Challenges for Trajectory Optimization: Contact

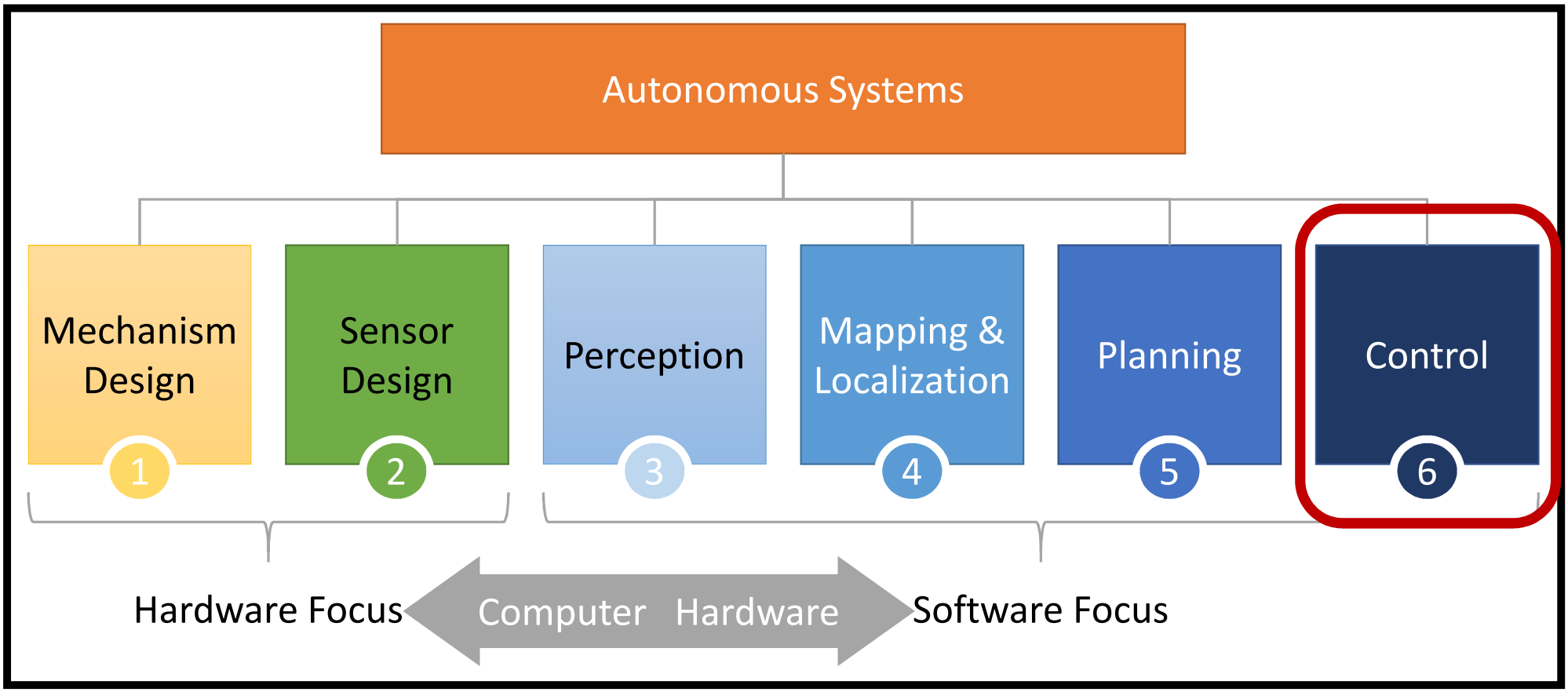


And then **combine solutions** to these **(conservative)** simpler problems

5 Key Takeaways:

1. Robot planning involves both **task and configuration spaces**
2. For many real problems, **collision checking** can be expensive
3. **Sample Based Planners** that leverage random search **(RRT/PRM)**:
   - **Probabilistically complete** (but can still be slow sometimes)
   - **Single-query (RRT) vs. Multi-query (PRM)**
   - **Probabilistically optimal** (RRT*) but generally need smoothers
4. **Trajectory Optimization** leverages local search to find **locally optimal** (generally smooth) solutions
   - Handles dynamics well but **not complete or robust**
   - Can use **off the shelf solvers** (SQP) but **generally slower** than a solver that **exploits sparsity** in the problem (DDP/iLQR)
   - **Contact is hard** and we (sometimes) use **simpler models** for tractability

Autonomous Systems / Robotics is a BIG space

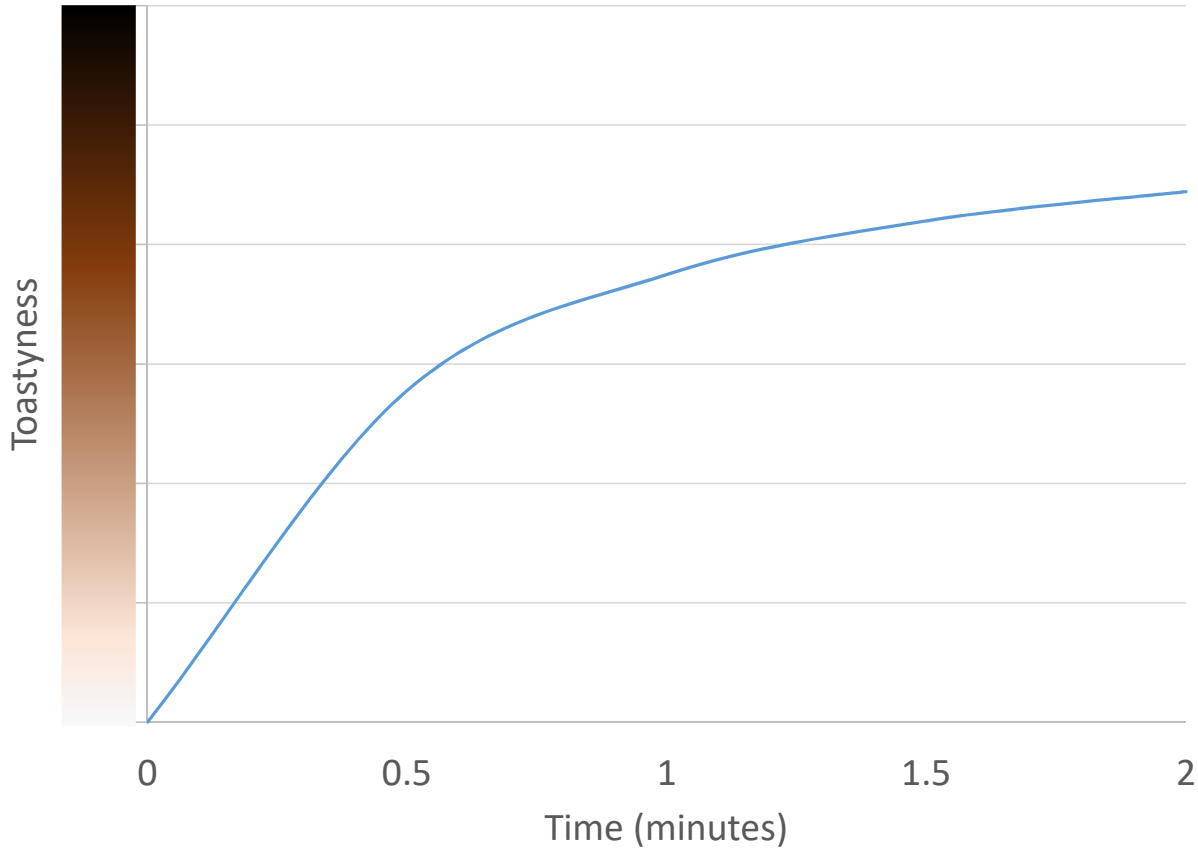


6 Control is the process of executing a plan in the real world

Well the simplest thing we could try would be to just execute the controls from our plan directly on the real system. This is called **Open-Loop Control!**

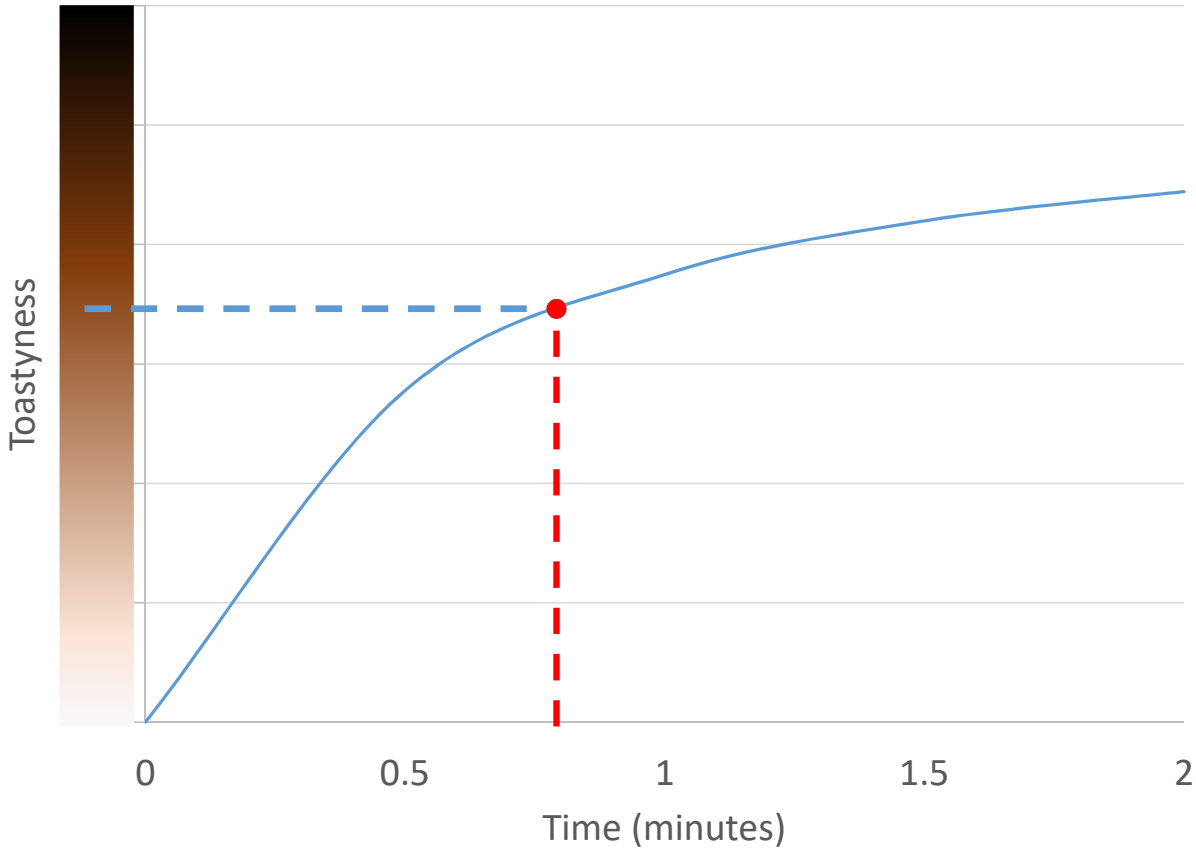
6 Open Loop Control

Adapted from MATLAB Control Toolbox



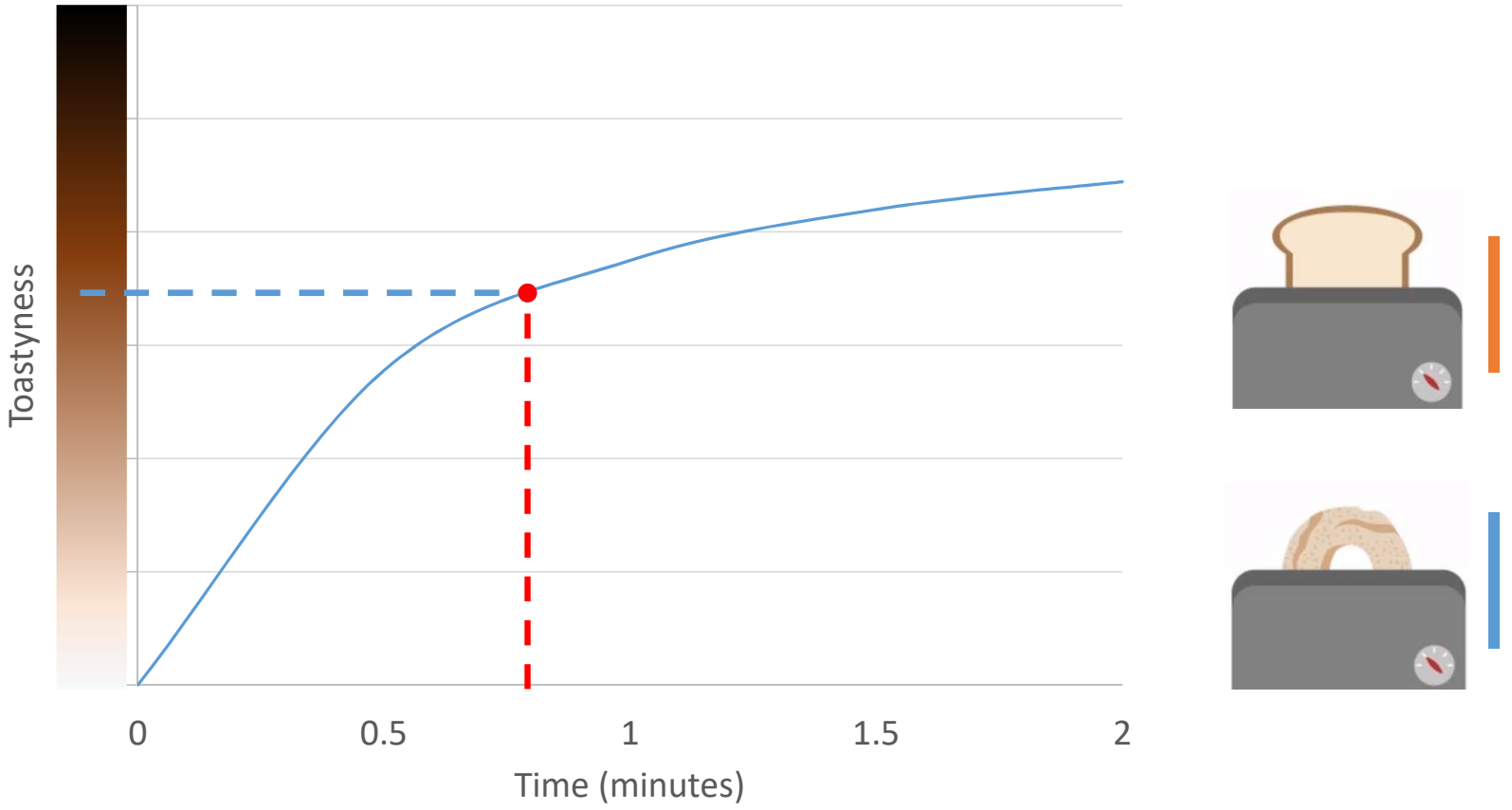
6 Open Loop Control

Adapted from MATLAB Control Toolbox



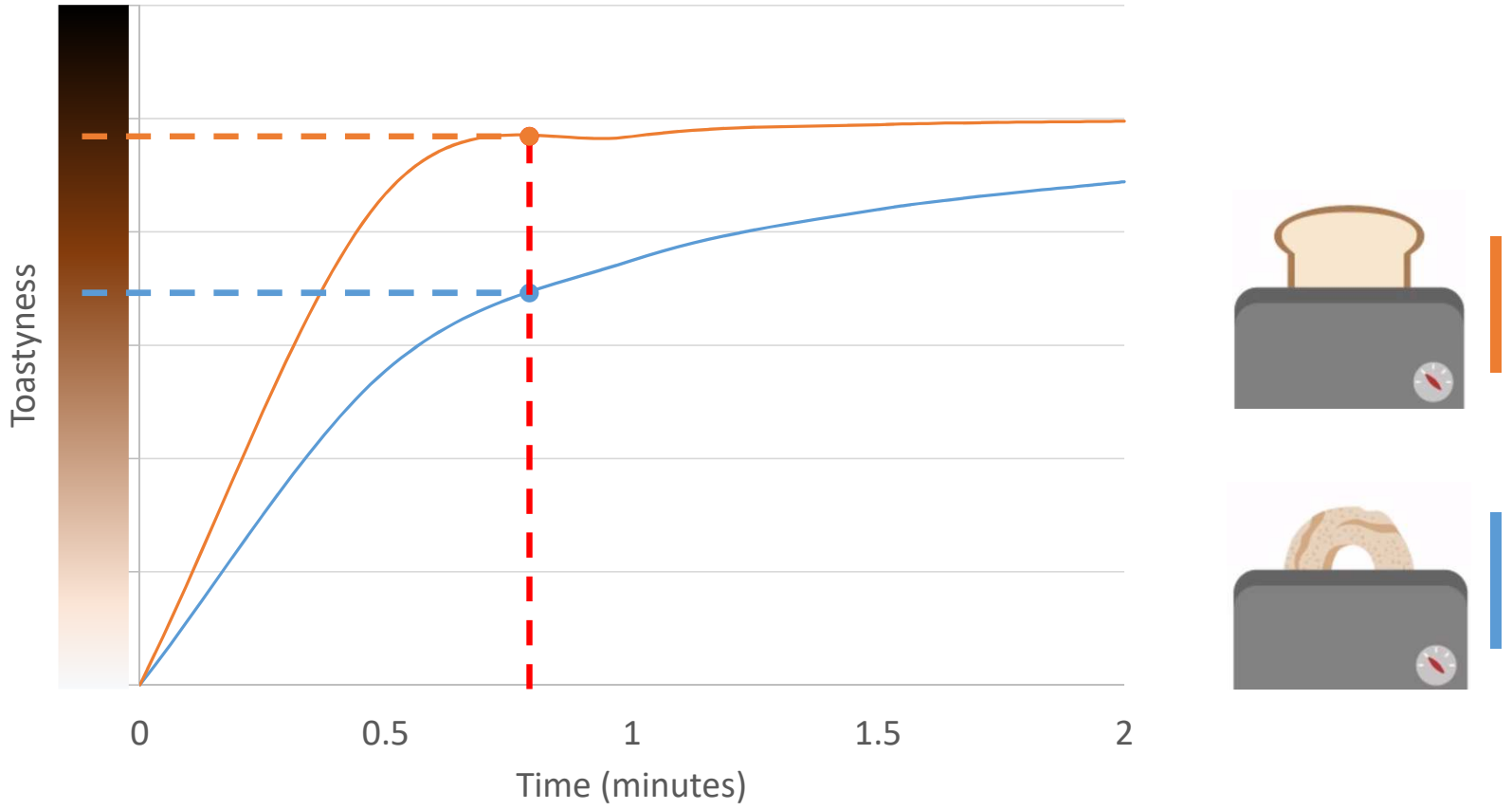
6 Open Loop Control

Adapted from MATLAB Control Toolbox



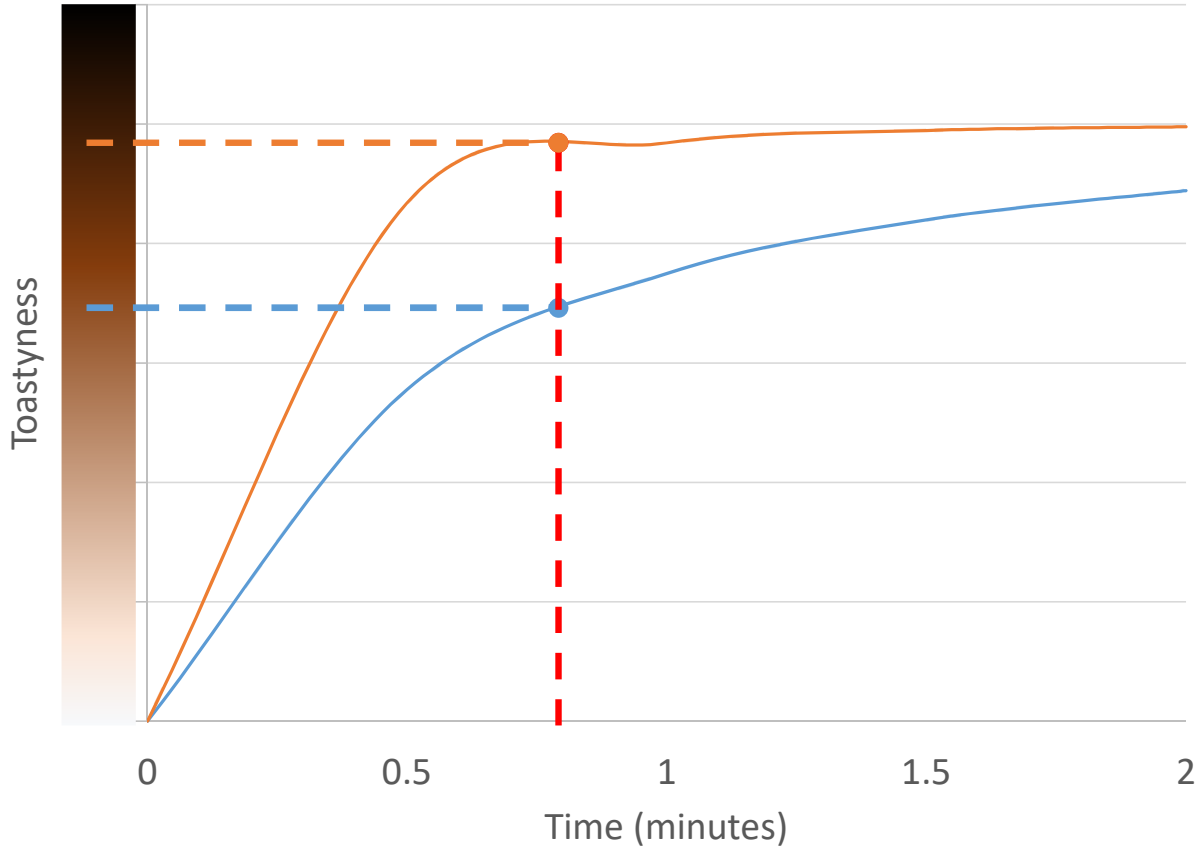
6 Open Loop Control

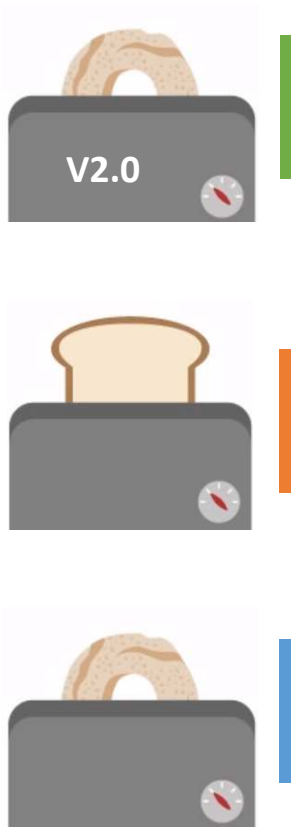
Adapted from MATLAB Control Toolbox



6 Open Loop Control

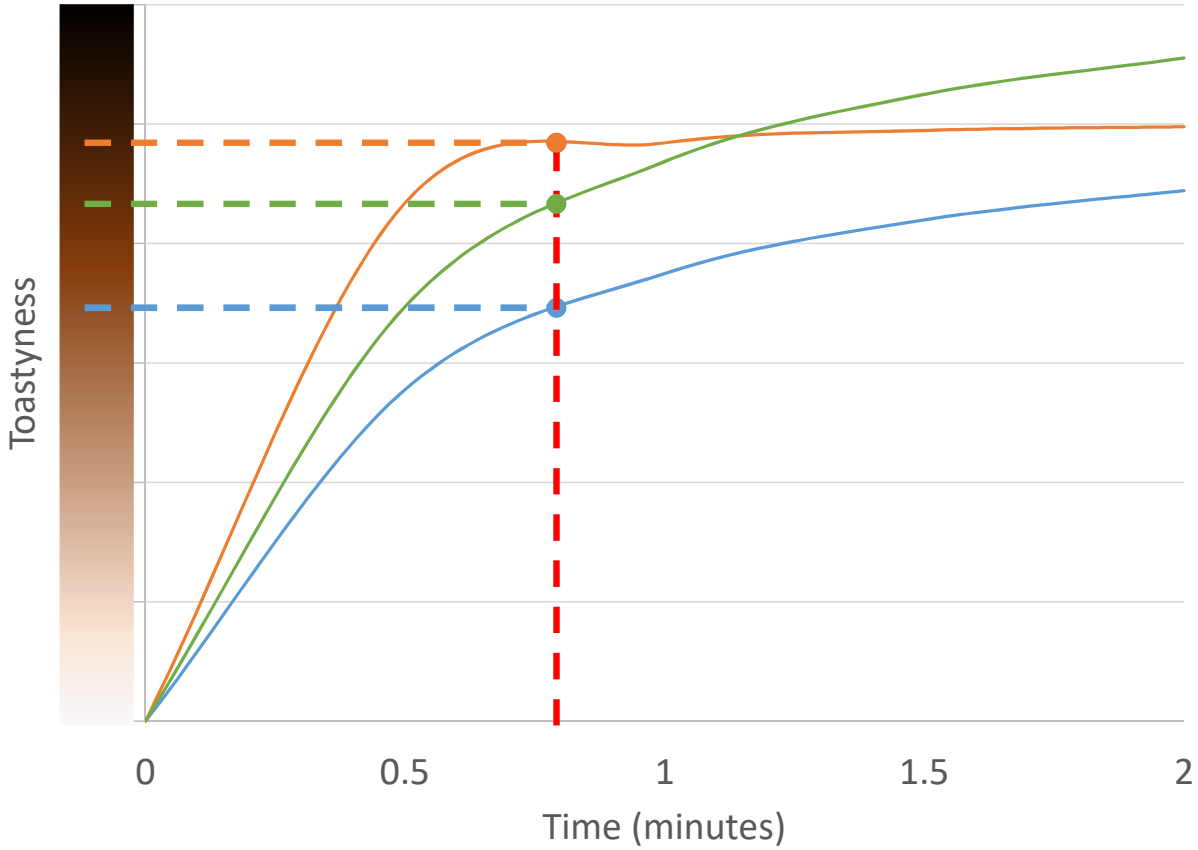
Adapted from MATLAB Control Toolbox

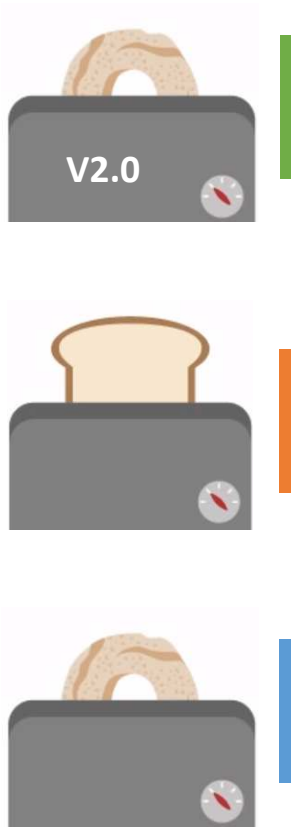




6 Open Loop Control

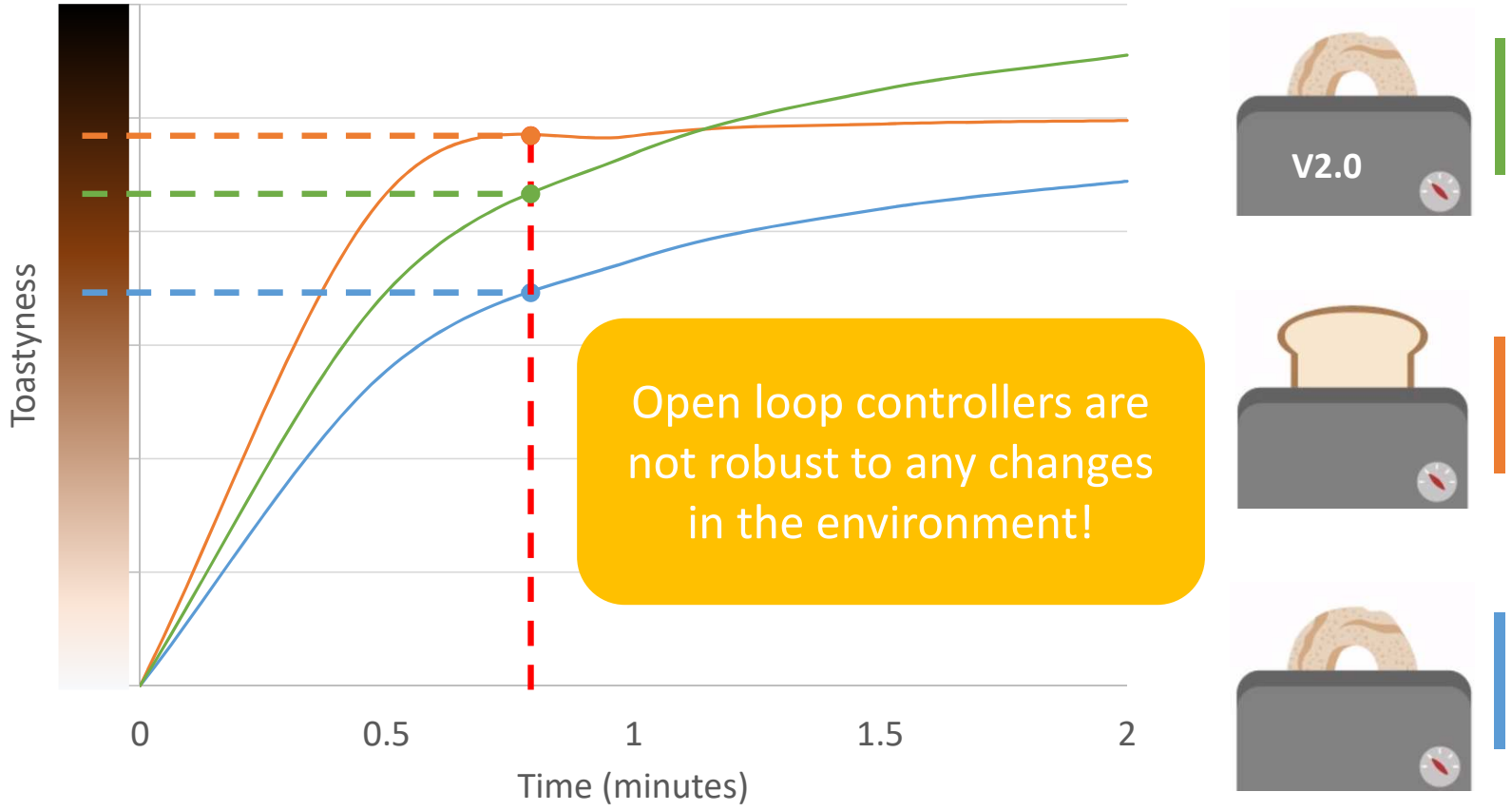
Adapted from MATLAB Control Toolbox





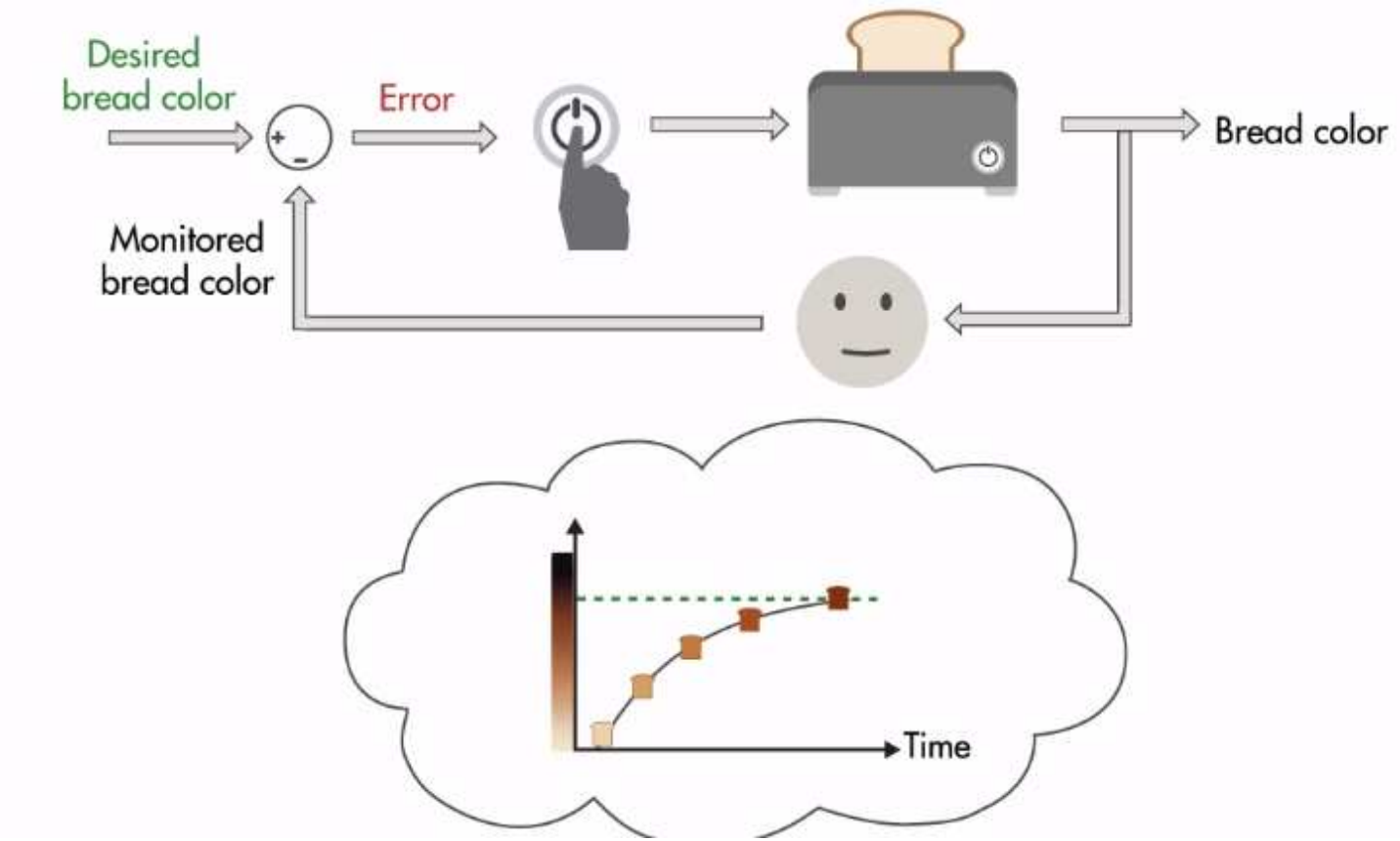
6 Open Loop Control

Adapted from MATLAB Control Toolbox



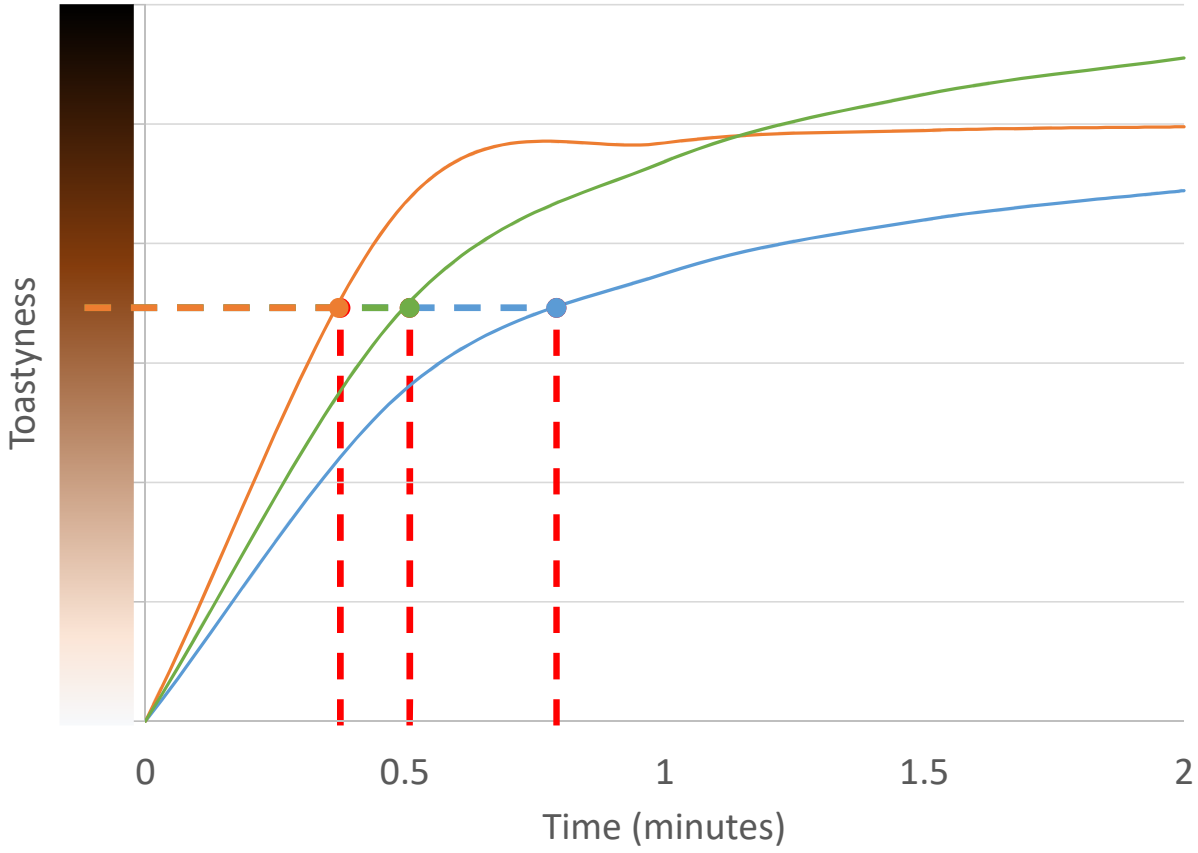
6 Feedback (Closed Loop) Control

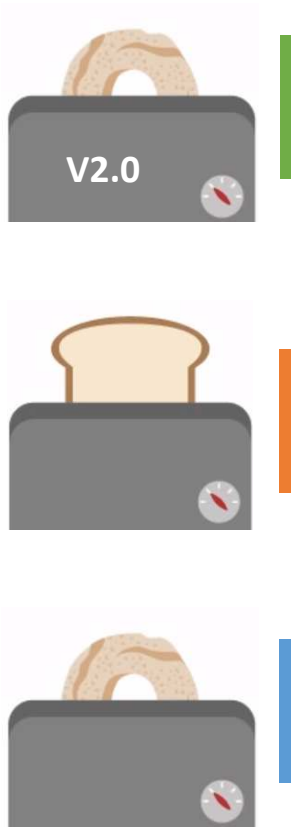
Adapted from MATLAB Control Toolbox



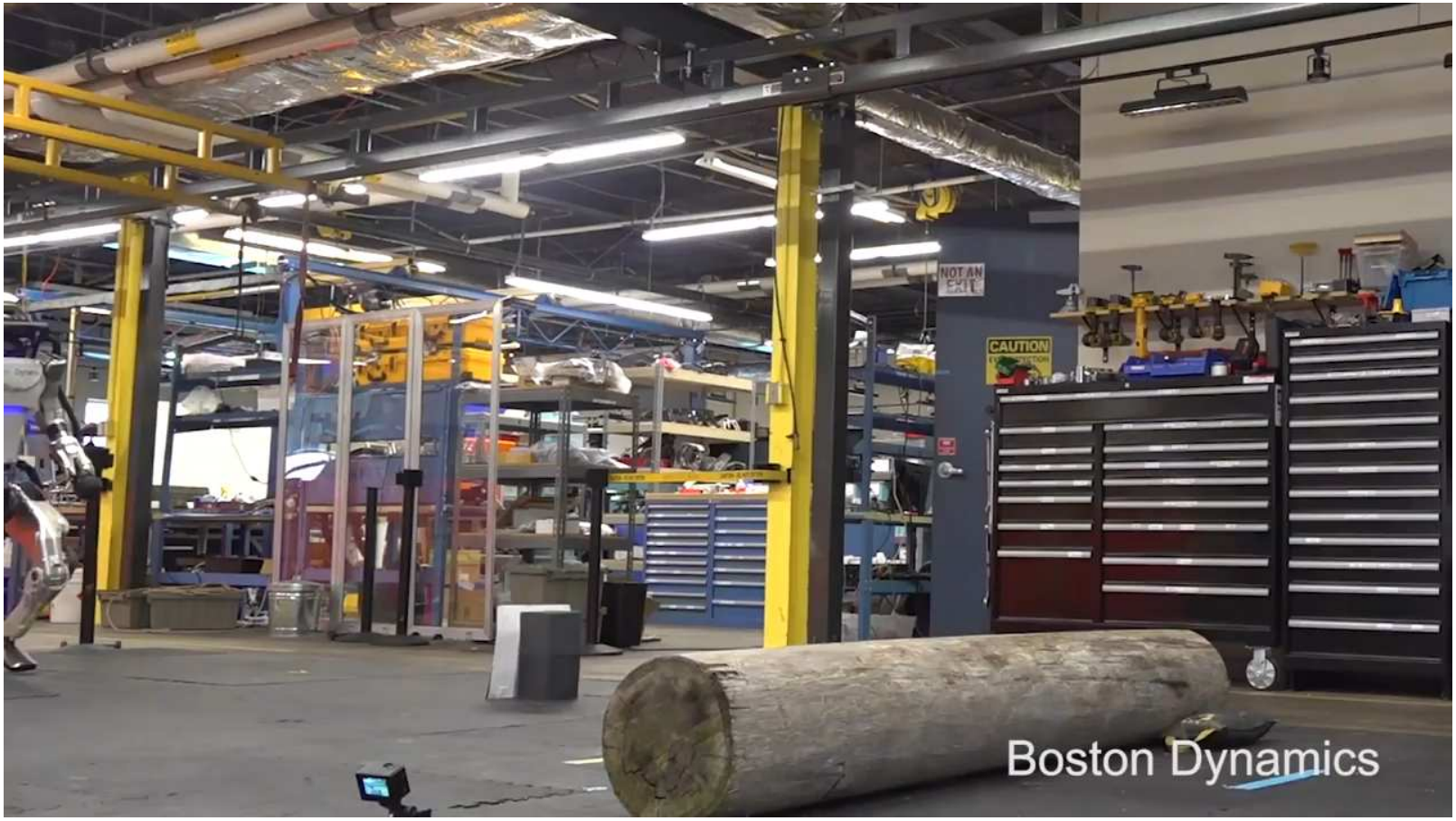
6 Feedback Control

Adapted from MATLAB Control Toolbox



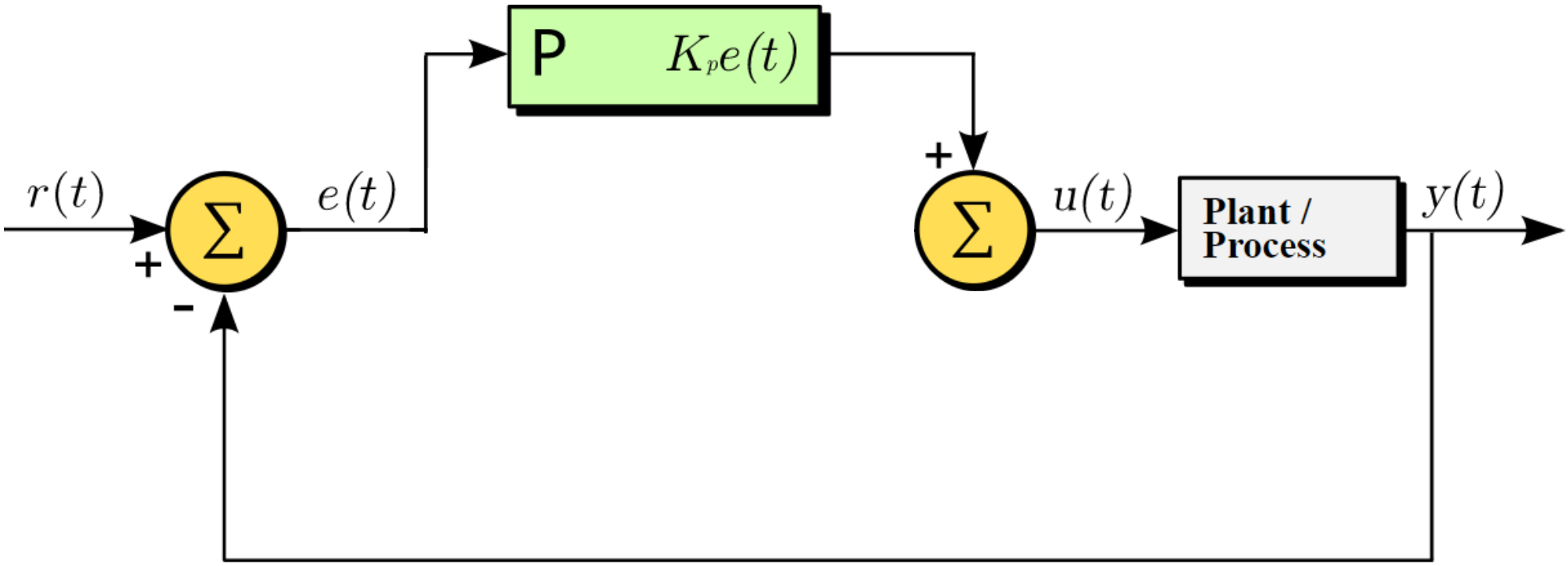


6 Feedback Control can lead to amazing performance!



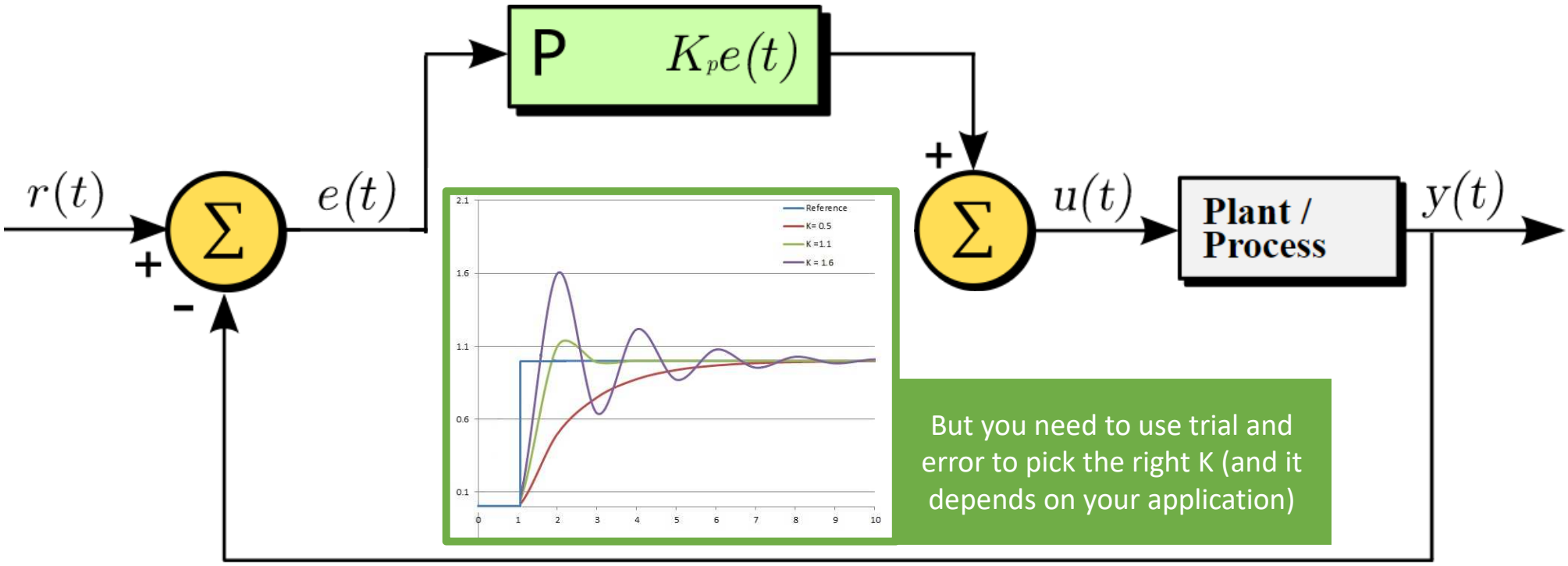
6 So how do we do Feedback Control in practice?

Adapted from Wikipedia



6 So how do we do Feedback Control in practice?

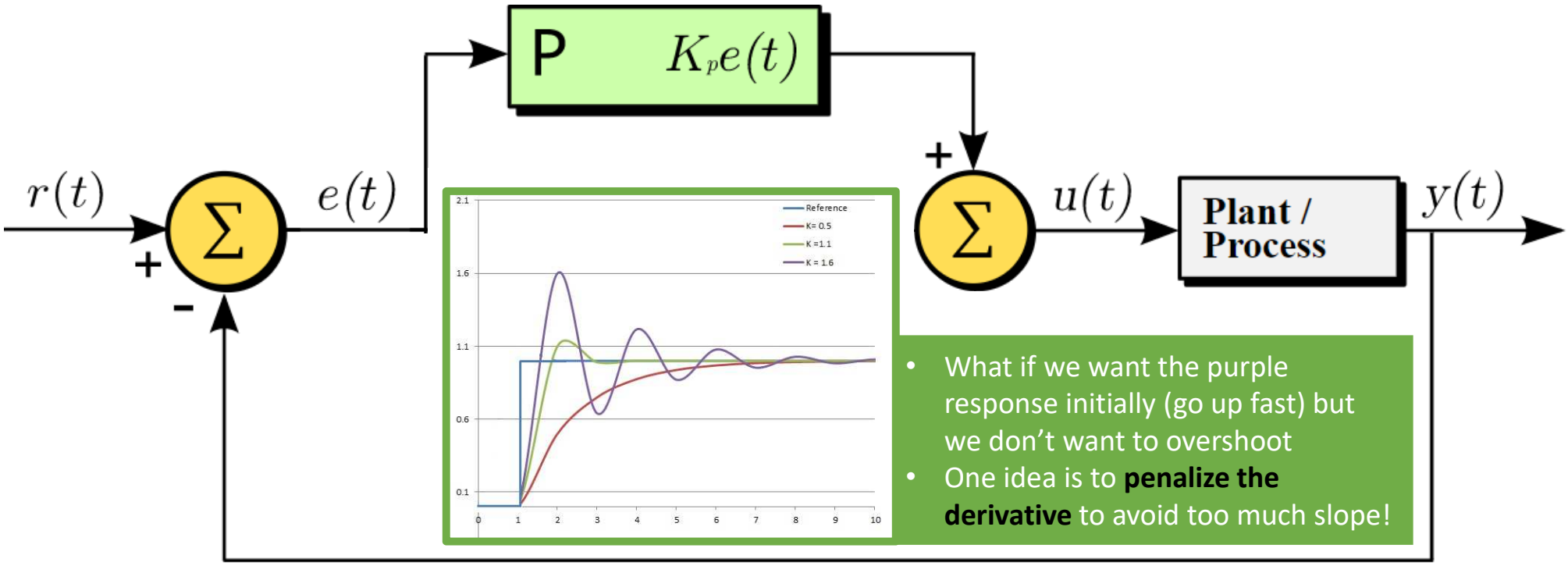
Adapted from Wikipedia



But you need to use trial and error to pick the right K (and it depends on your application)

6 So how do we do Feedback Control in practice?

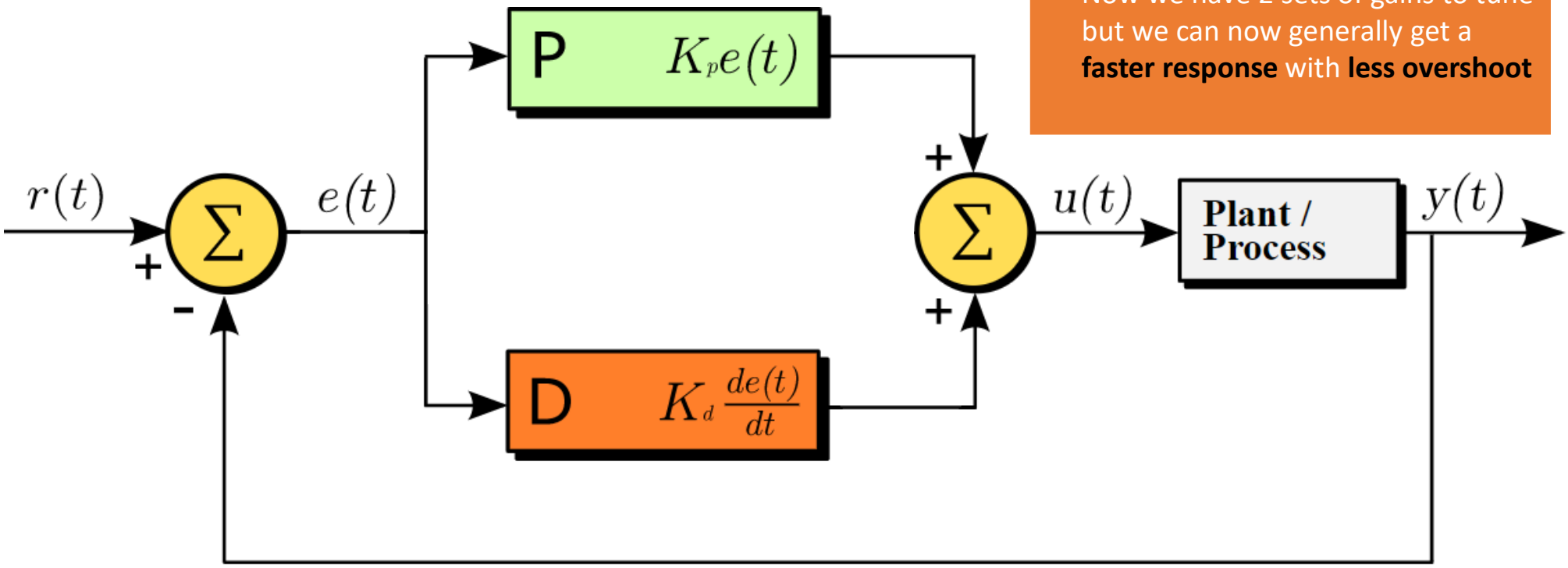
Adapted from Wikipedia



- What if we want the purple response initially (go up fast) but we don't want to overshoot
- One idea is to **penalize the derivative** to avoid too much slope!

6 So how do we do Feedback Control in practice?

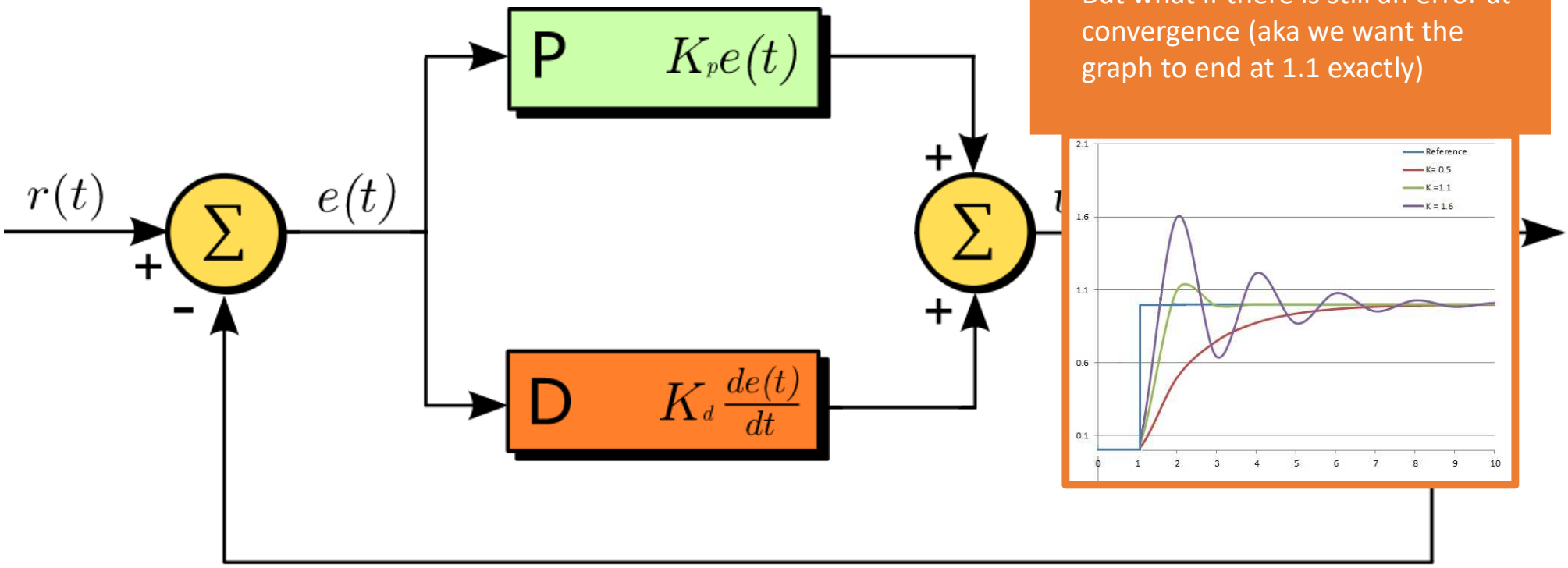
Adapted from Wikipedia



- Now we have 2 sets of gains to tune but we can now generally get a **faster response** with **less overshoot**

6 So how do we do Feedback Control in practice?

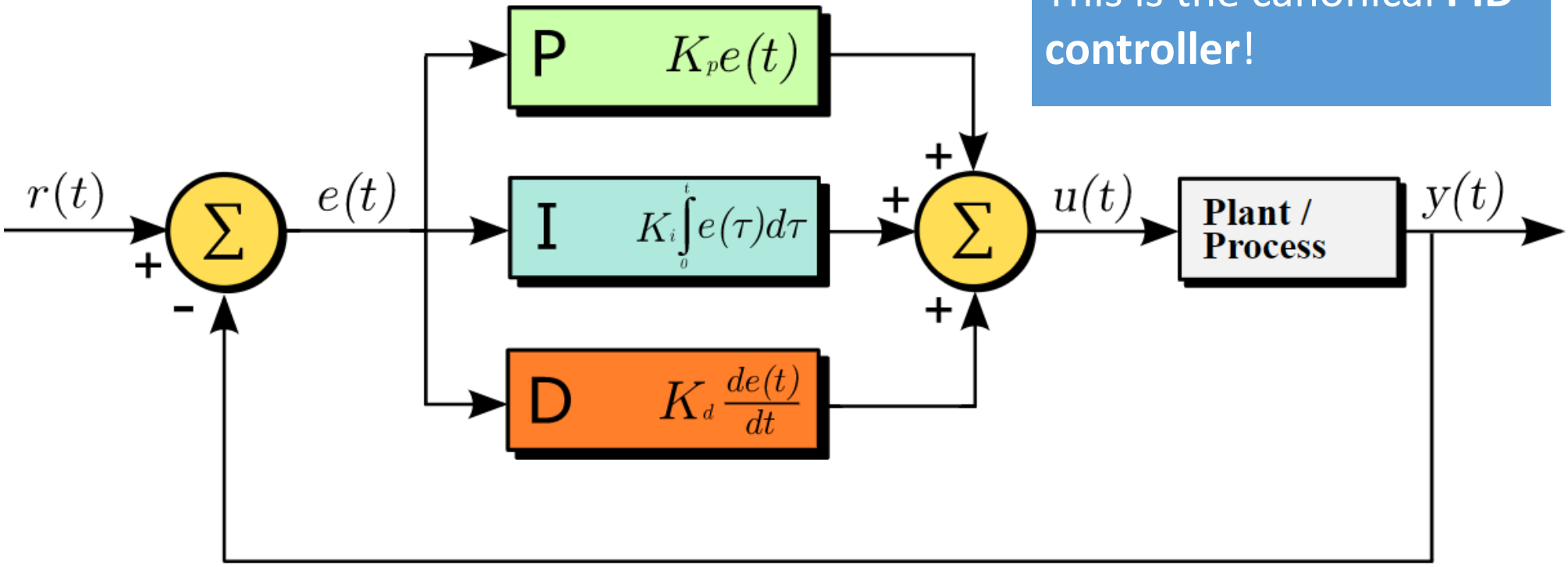
Adapted from Wikipedia



- But what if there is still an error at convergence (aka we want the graph to end at 1.1 exactly)

6 So how do we do Feedback Control in practice?

Adapted from Wikipedia



6 So how do we do Feedback Control in practice?

Adapted from Wikipedia

Ziegler–Nichols method

From Wikipedia, the free encyclopedia

Main article: PID controller

The **Ziegler–Nichols tuning method** is a heuristic method of tuning a PID controller. It was developed by John G. Ziegler and Nathaniel B. Nichols. It is performed by setting the *I* (integral) and *D* (derivative) gains to zero. The "P" (proportional) gain, K_p is then increased (from zero) until it reaches the **ultimate gain** K_u, at which the output of the control loop has stable and consistent oscillations. K_u and the oscillation period T_u are used to set the P, I, and D gains depending on the type of controller used:

Ziegler–Nichols method[1]

Control Type	K_p	T_i	T_d	K_i	K_d
P	$0.5K_u$	–	–	–	–
PI	$0.45K_u$	$T_u/1.2$	–	$0.54K_u/T_u$	–
PD	$0.8K_u$	–	$T_u/8$	–	$K_uT_u/10$
classic PID[2]	$0.6K_u$	$T_u/2$	$T_u/8$	$1.2K_u/T_u$	$3K_uT_u/40$
Pessen Integral Rule[2]	$7K_u/10$	$2T_u/5$	$3T_u/20$	$1.75K_u/T_u$	$21K_uT_u/200$
some overshoot[2]	$K_u/3$	$T_u/2$	$T_u/3$	$0.666K_u/T_u$	$K_uT_u/9$
no overshoot[2]	$K_u/5$	$T_u/2$	$T_u/3$	$(2/5)K_u/T_u$	$K_uT_u/15$

Tuning PID gains is an art and there is a whole literature on a variety of methods to get particular types of response curves!

6 PID controllers work really well in practice

6 Tuning gains is hard and non-intuitive is there a better way?

6 Tuning gains is hard and non-intuitive is there a better way?

Of course there is or I wouldn’t need the transition slide!

6 The LQR Controller

What if instead of specifying gains we can specify a **cost function** we want to achieve…

6 The LQR Controller

What if instead of specifying gains we can specify a **cost function** we want to achieve…

Maybe something like track the desired state but don’t use too much energy to do it?

6 The LQR Controller

What if instead of specifying gains we can specify a **cost function** we want to achieve…

Maybe something like track the desired state but don't use too much energy to do it?

Instead of tuning gains we can tune cost weights (Q,R) which are often more intuitive

$$L(x,u) = \underbrace{(x - x_g)^T Q (x - x_g)}_{\text{Deviation of the state from some goal state}} + \underbrace{u^T R u}_{\text{Effort (torque)}}$$

6 The LQR Controller

It turns out if we minimize this quadratic cost over time with a linear model of the dynamics

$$\min_{x,u} \sum_{k=0}^{N} (x_k - x_g)^T Q (x_k - x_g) + u_k{}^T R u_k$$

$$\text{s.t.} \quad x_{k+1} = A x_k + B u_k$$

There is a closed form solution to the optimal feedback controller! (Riccati Equation)

$$u_k = -K_k x_k$$

6 The LQR Controller

It turns out if we minimize this quadratic cost over time with a linear model of the dynamics

$$\min_{x,u} \sum_{k=0}^{N} (x_k - x_g)^T Q (x_k - x_g) + u_k{}^T R u_k$$

$$\text{s.t.} \quad x_{k+1} = A x_k + B u_k$$

This is used widely in practice!

There is a closed form solution to the optimal feedback controller! (Riccati Equation)

$$u_k = -K_k x_k$$

6

We can also use LQR in RRT as a better metric of "distance" and the feedback controller as the best "extend"

Finite-horizon, discrete-time LQR [edit]

For a discrete-time linear system described by: [1]

$$x_{k+1} = Ax_k + Bu_k$$

with a performance index defined as:

$$J = x_N^T Q x_N + \sum_{k=0}^{N-1} \left(x_k^T Q x_k + u_k^T R u_k + 2 x_k^T N u_k \right)$$

the optimal control sequence minimizing the performance index is given by:

$$u_k = -F_k x_k$$

where:

$$F_k = (R + B^T P_{k+1} B)^{-1} (B^T P_{k+1} A + N^T)$$

Feedback Controller for "Extend"

and P_k is found iteratively backwards in time by the dynamic Riccati equation:

$$P_{k-1} = A^T P_k A - (A^T P_k B + N)(R + B^T P_k B)^{-1} (B^T P_k A + N^T) + Q$$

Cost-to-Go as "Distance Metric"

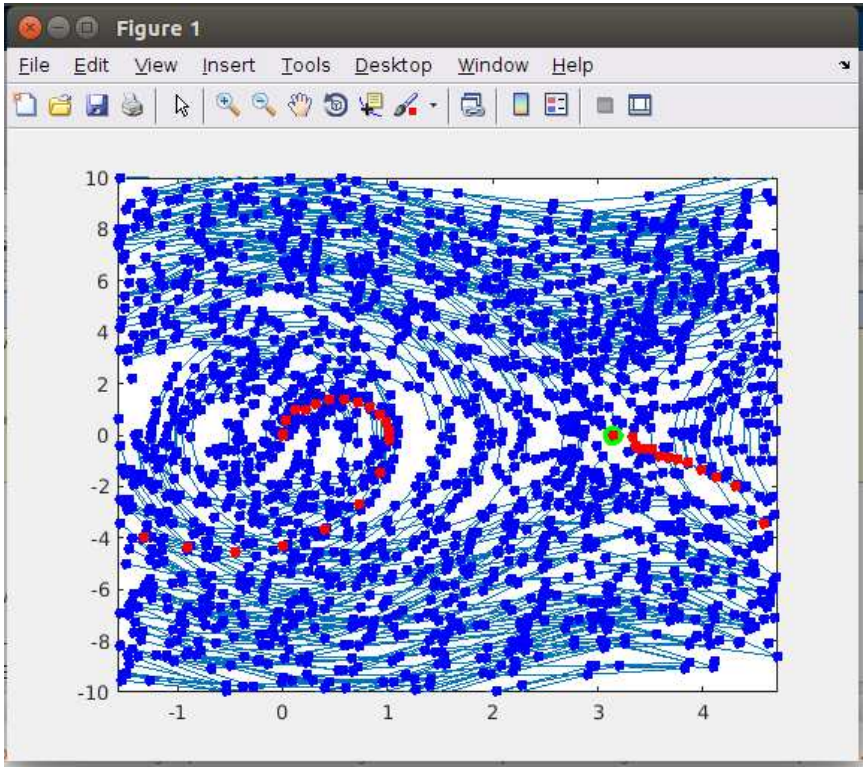
from terminal condition $P_N = Q$. Note that u_N is not defined, since x is driven to its final state x_N by $Ax_{N-1} + Bu_{N-1}$.

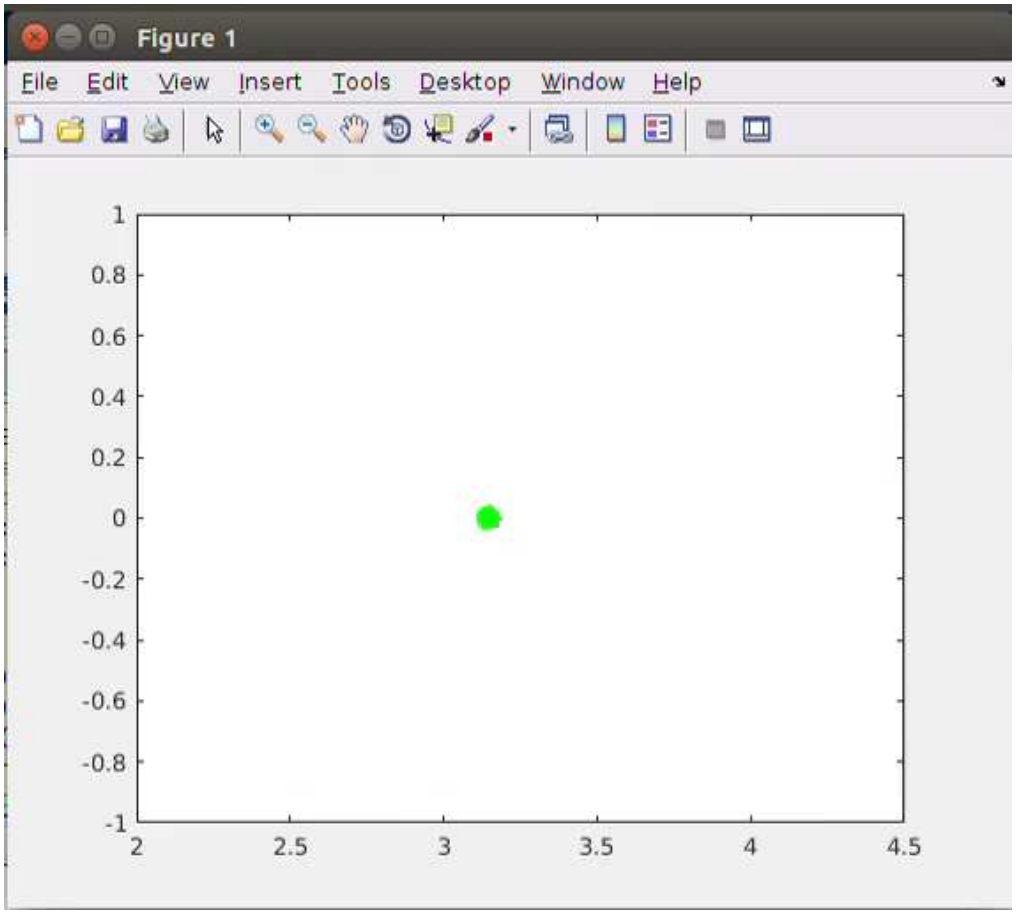
Bellman Updates

$$V_N(x_N) = c(x_N, u_N)$$

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

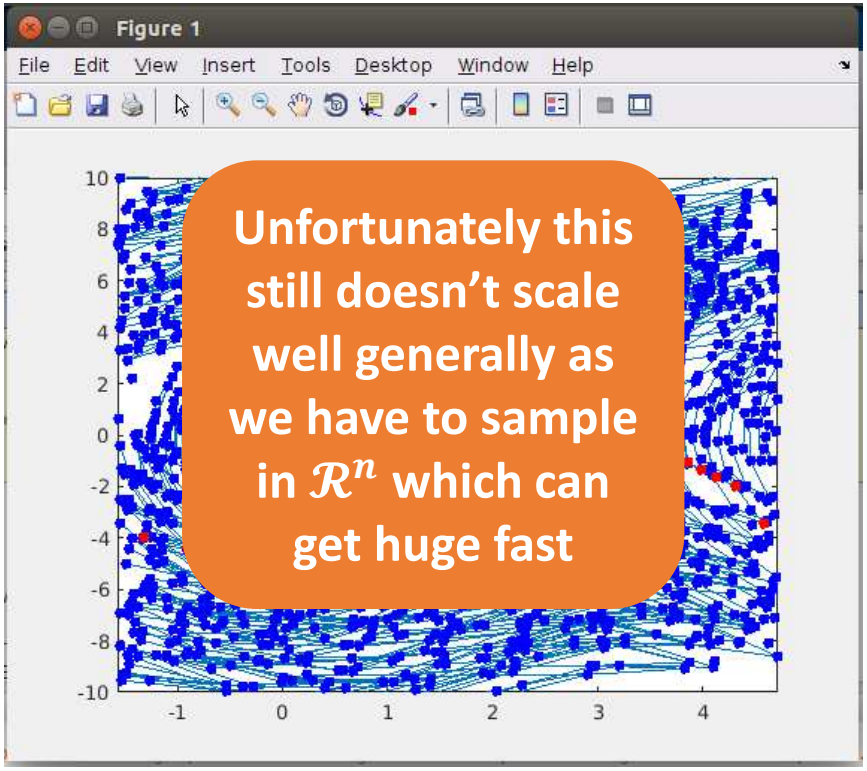
6 We can also use LQR in RRT as a better metric of “distance” and the feedback controller as the best “extend”



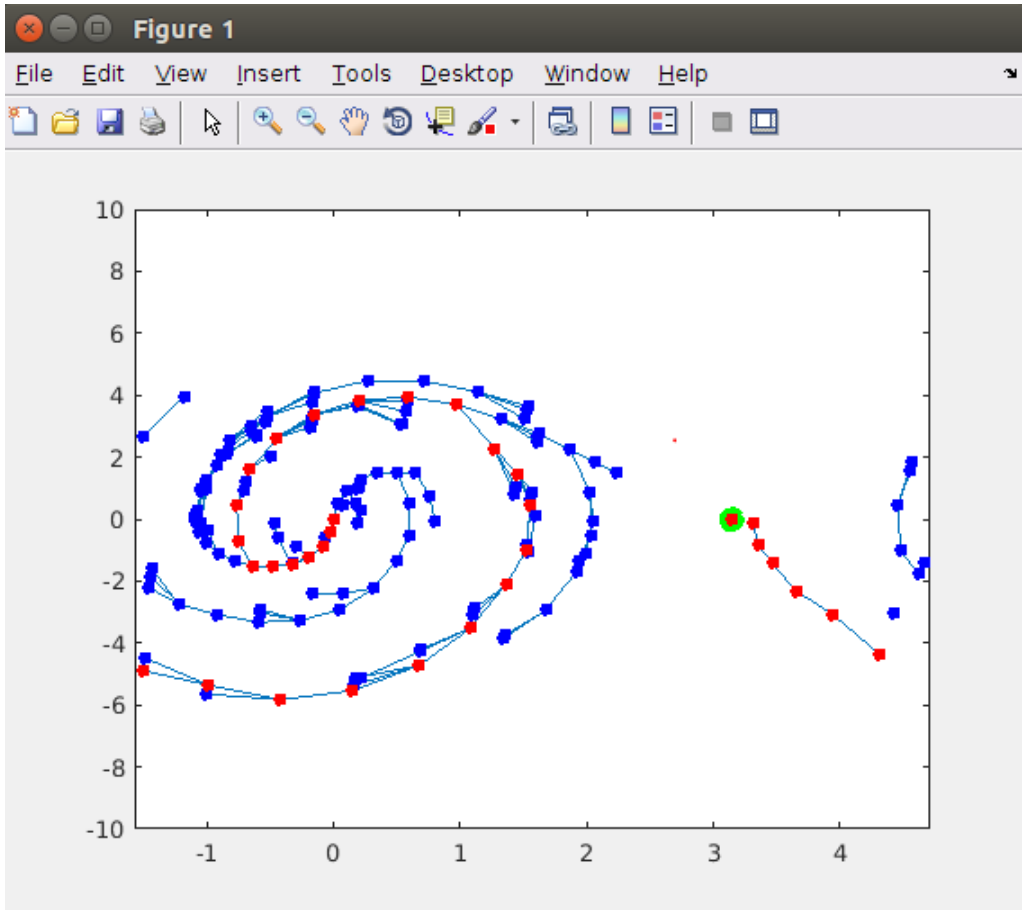


[Perez et. al. LQR-RRT*]

6 We can use LQR in RRT as a better metric of “distance” and the feedback controller as the best “extend”



Unfortunately this still doesn’t scale well generally as we have to sample in $\mathcal{R}^n$ which can get huge fast



[Perez et. al. LQR-RRT*]

6 So what have we learned so far?

1. Real world autonomous systems need to use **Feedback Control**
2. **PID** controllers are simple and effective but require **gain tuning**
3. **LQR** controllers allow for **cost function design** instead
4. PID and LQR require a plan to already exist and are simply **tracking controllers**

6 So what have we learned so far?

1. Real world autonomous systems need to use **Feedback Control**
2. **PID** controllers are simple and effective but require **gain tuning**
3. **LQR** controllers allow for **cost function design** instead
4. PID and LQR require a plan to already exist and are simply **tracking controllers**

- But what happens if we **deviate** so much from our original plan that it is **no longer valid**? How do we initiate **re-plans**?

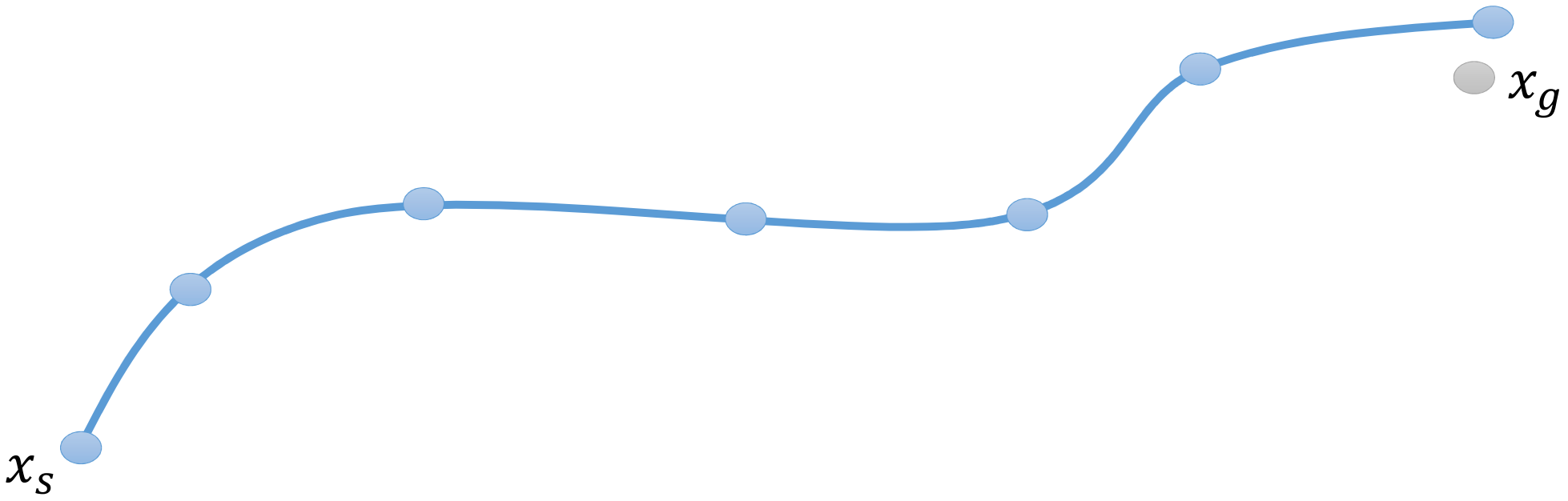
6 So what have we learned so far?

1. Real world autonomous systems need to use **Feedback Control**
2. **PID** controllers are simple and effective but require **gain tuning**
3. **LQR** controllers allow for **cost function design** instead
4. PID and LQR require a plan to already exist and are simply **tracking controllers**

- But what happens if we **deviate** so much from our original plan that it is **no longer valid**? How do we initiate **re-plans**?

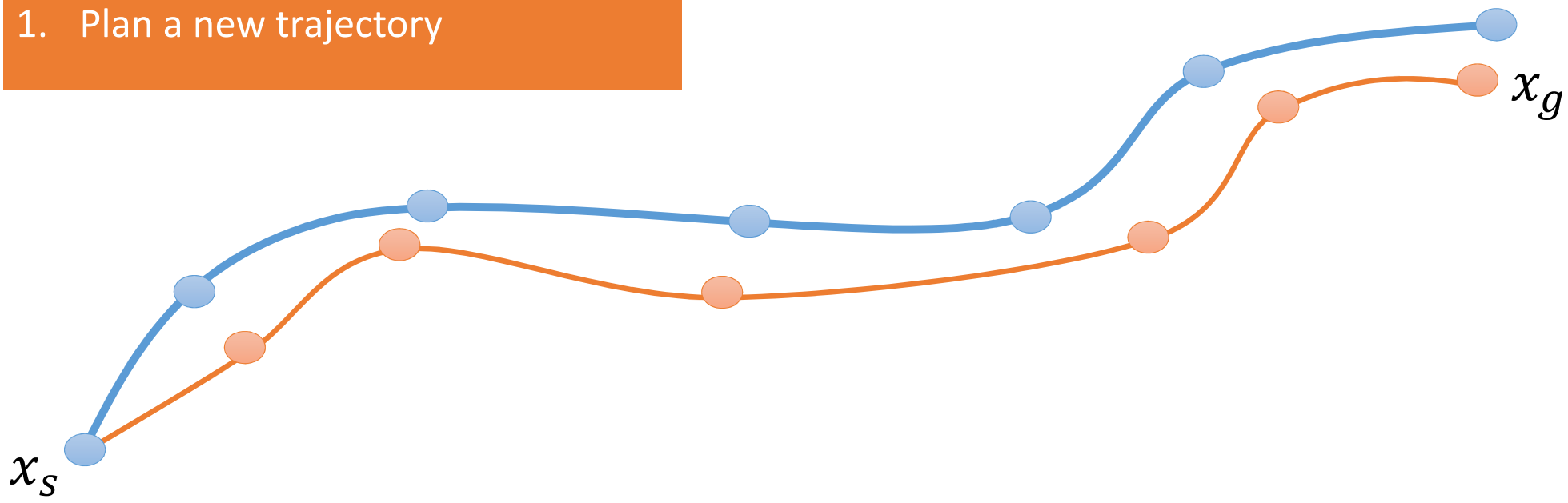
This is an open unsolved problem!

Model Predictive Control: re-planning fast enough that the plan becomes the controller!



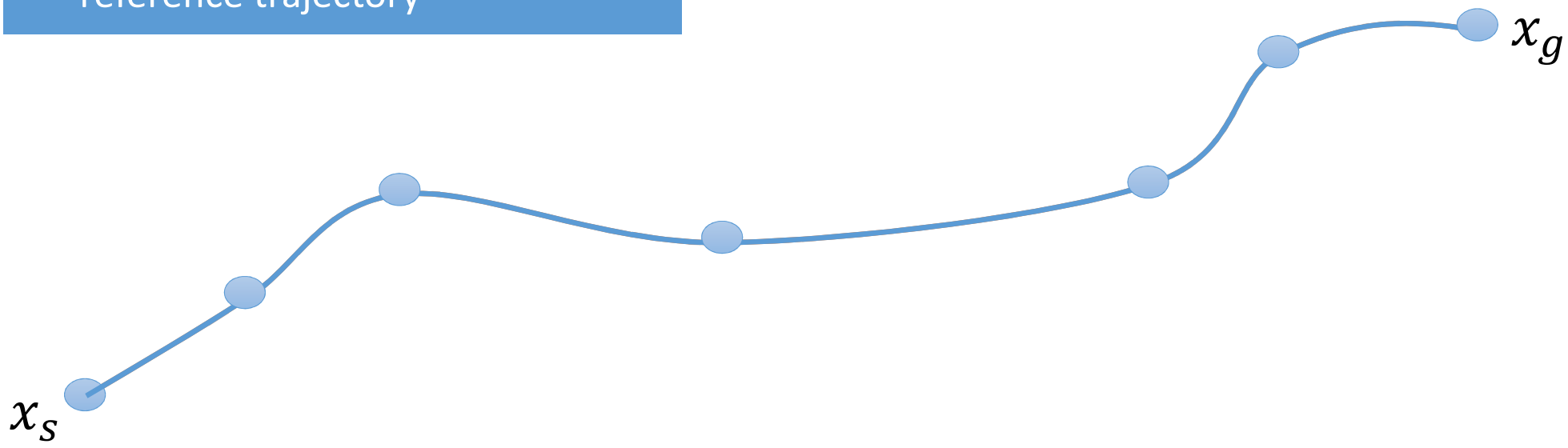
Model Predictive Control: re-planning fast enough that the plan becomes the controller!

1. Plan a new trajectory



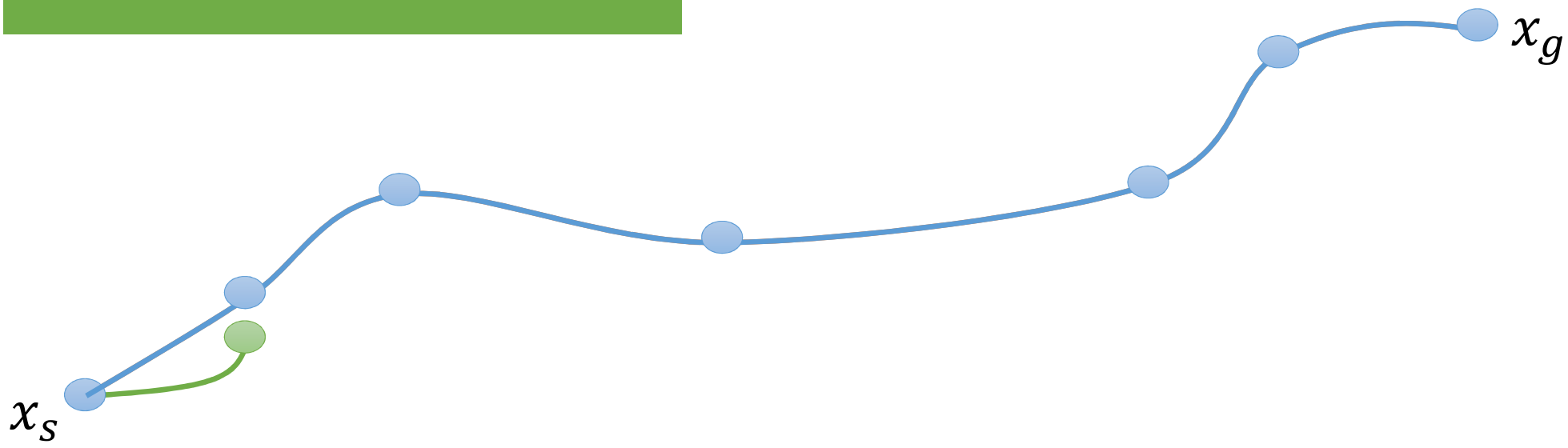
Model Predictive Control (MPC): re-planning fast enough that the plan becomes the controller!

2. The new plan becomes the reference trajectory



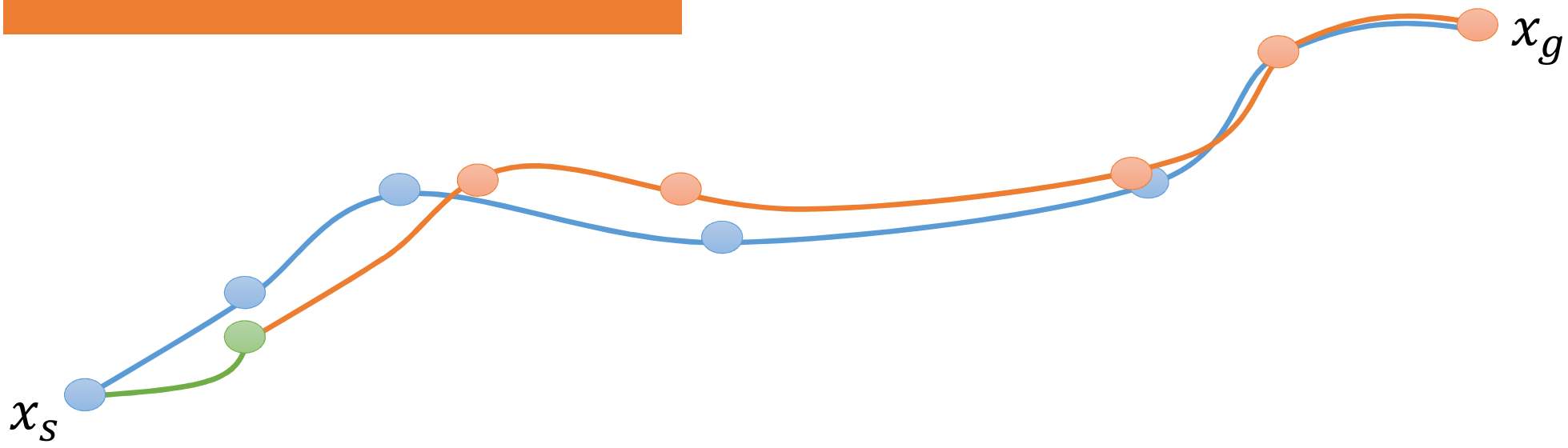
Model Predictive Control (MPC): re-planning fast enough that the plan becomes the controller!

3. Execute the first step of the plan



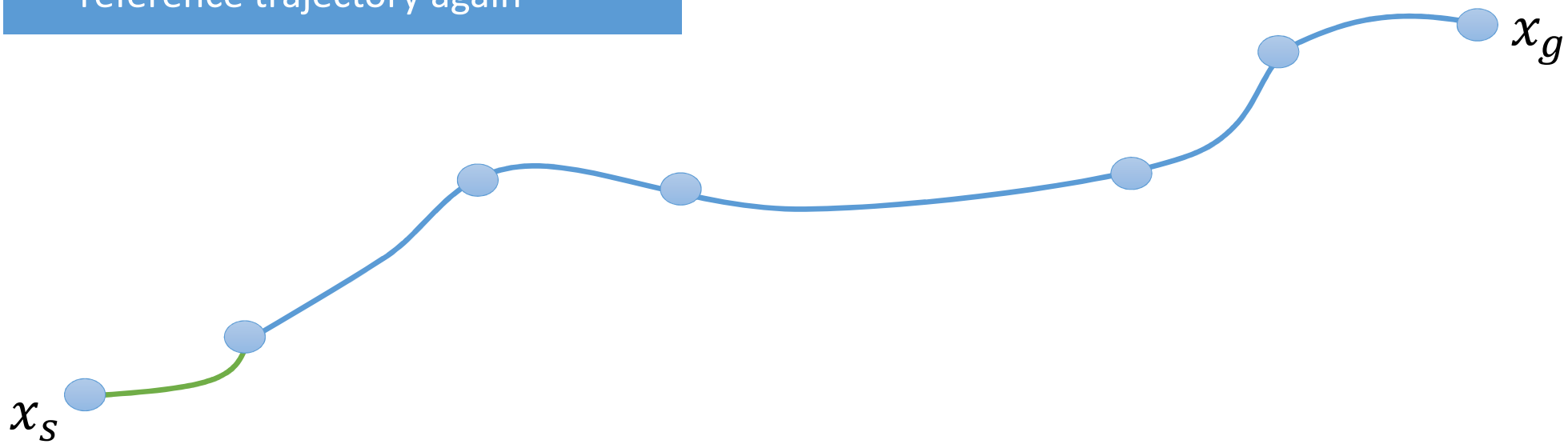
Model Predictive Control (MPC): re-planning fast enough that the plan becomes the controller!

1. Re-plan based on that step



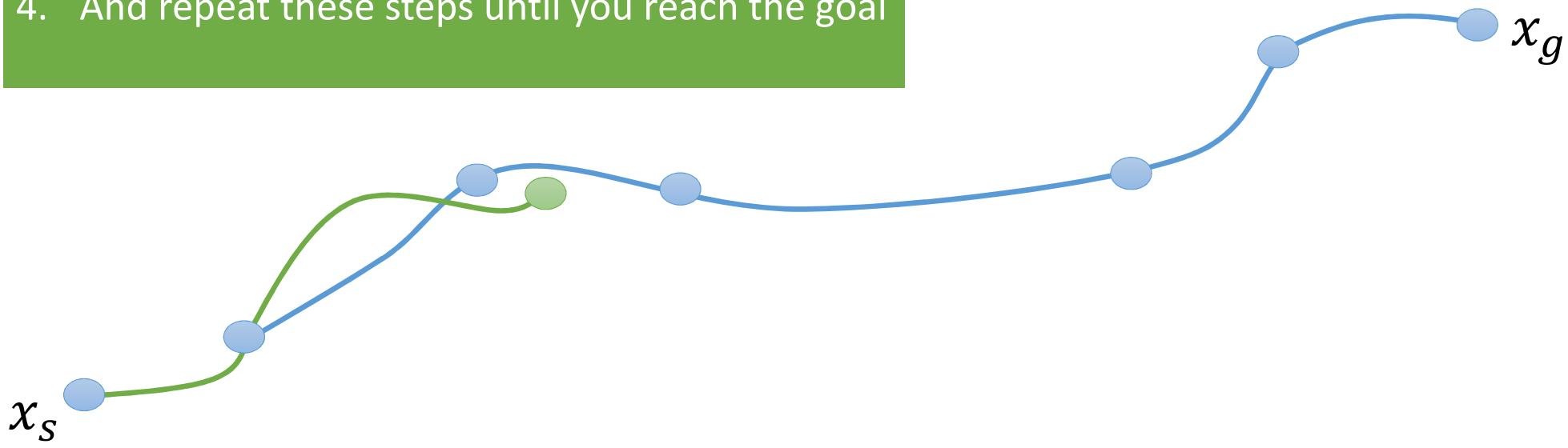
Model Predictive Control (MPC): re-planning fast enough that the plan becomes the controller!

2. The new plan becomes the reference trajectory again

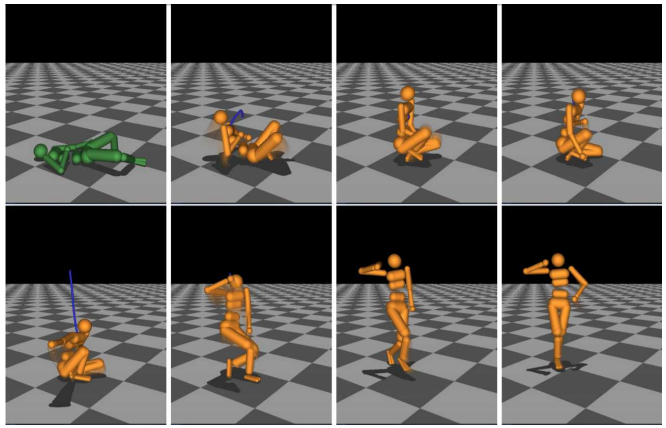


Model Predictive Control (MPC): re-planning fast enough that the plan becomes the controller!

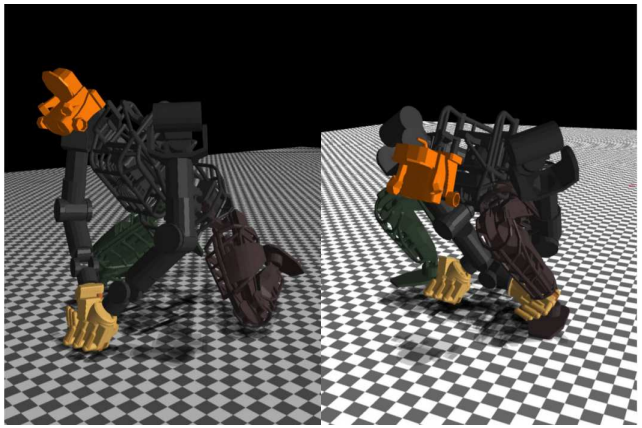
3. Execute the first step of the new plan again
4. And repeat these steps until you reach the goal



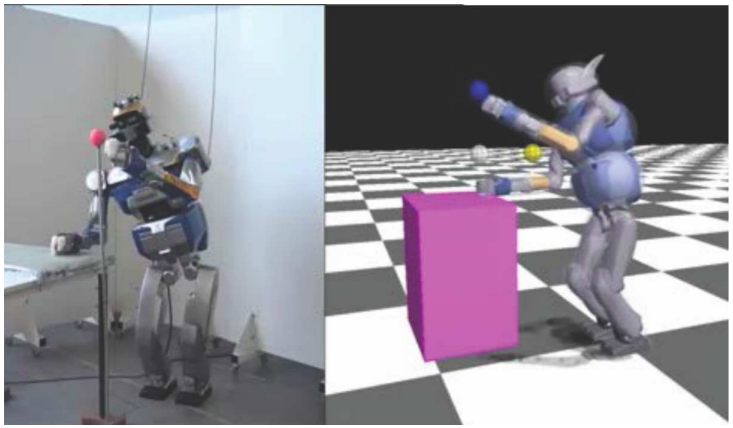
Recently MPC has been used in a variety of complex autonomous systems in simulation and on physical robots



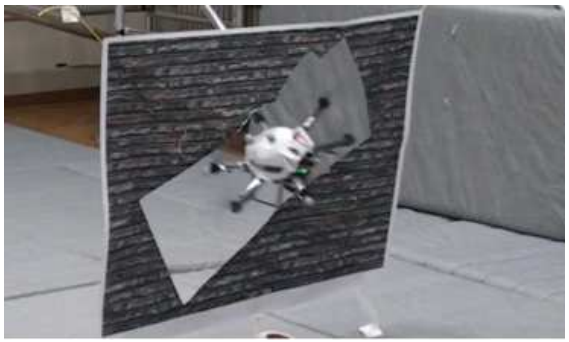
[Tassa et. al. IROS 2012]



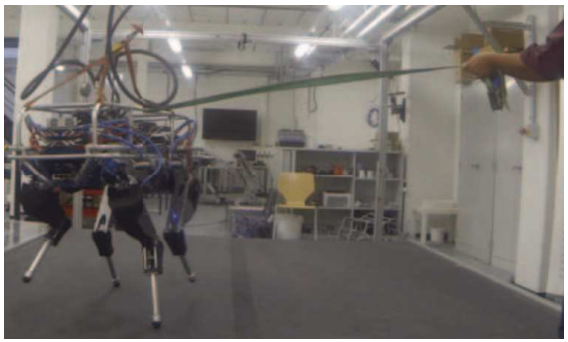
[Erez et. al. Humanoids 2013]



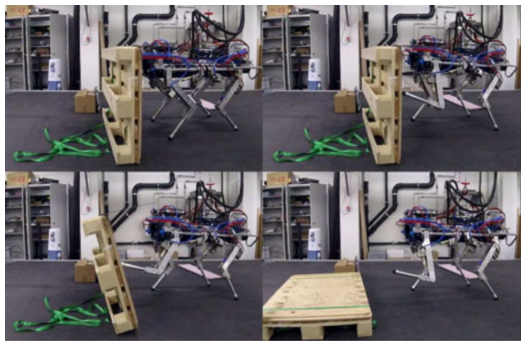
[Koenemann et. al. IROS 2015]



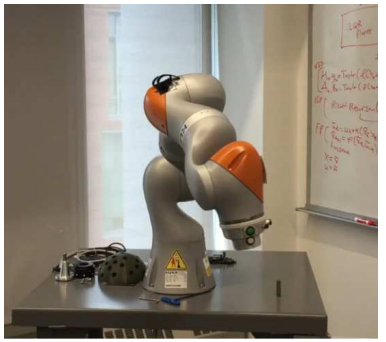
[Neunert et. al. ICRA 2016]



[Neunert et. al. Humanoids 2017]

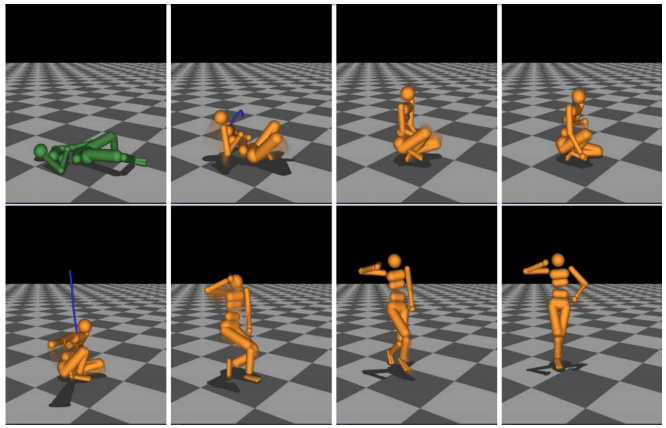


[Farshidian et. al. IEEE RAL 2017]

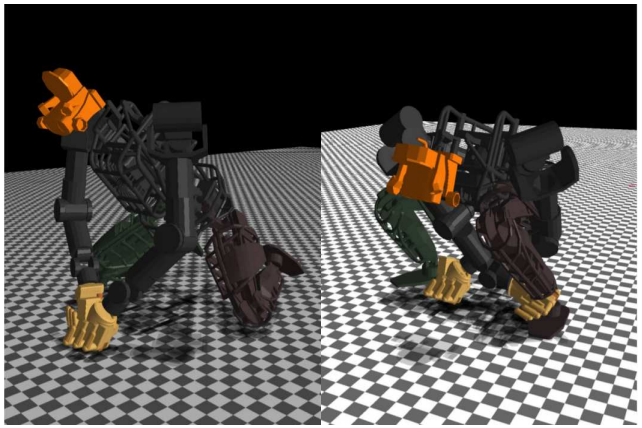


[Plancher et. al. WAFR 2018]
[Plancher et. al. ICRA 2019]

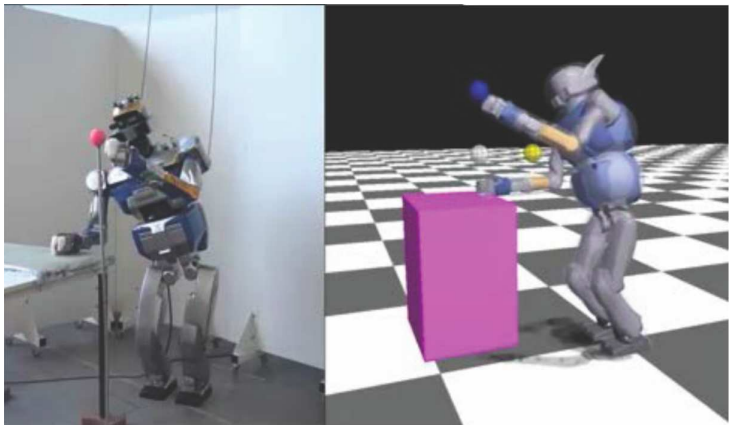
Recently MPC has been used in a variety of complex autonomous systems in simulation and on physical robots



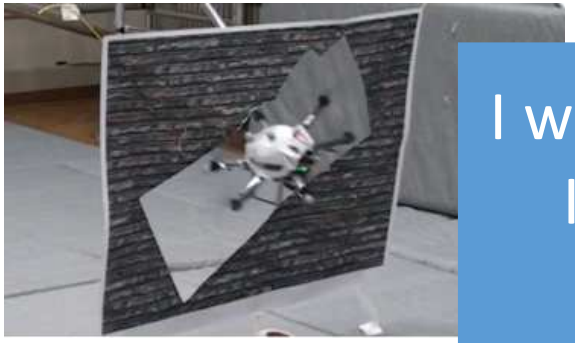
[Tassa et. al. IROS 2012]



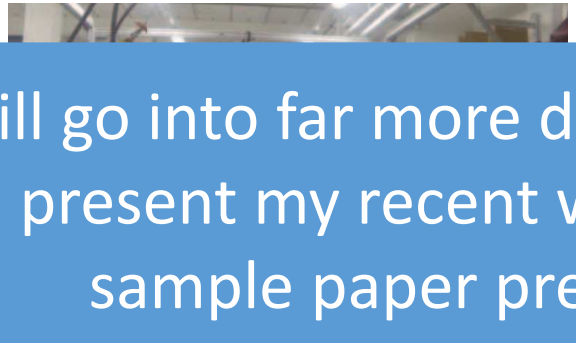
[Erez et. al. Humanoids 2013]



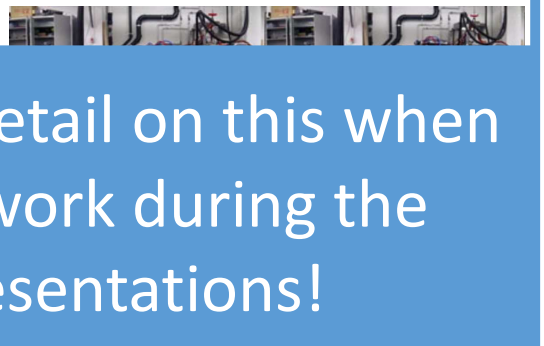
[Koenemann et. al. IROS 2015]



[Neunert et. al. ICRA 2016]

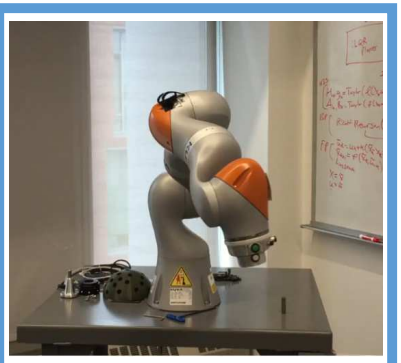


[Neunert et. al. Humanoids 2017]



[Farshidian et. al. IEEE RAL 2017]

I will go into far more detail on this when I present my recent work during the sample paper presentations!



[Plancher et. al. WAFR 2018]
[Plancher et. al. ICRA 2019]

6 Practical Challenges for Control: Contact

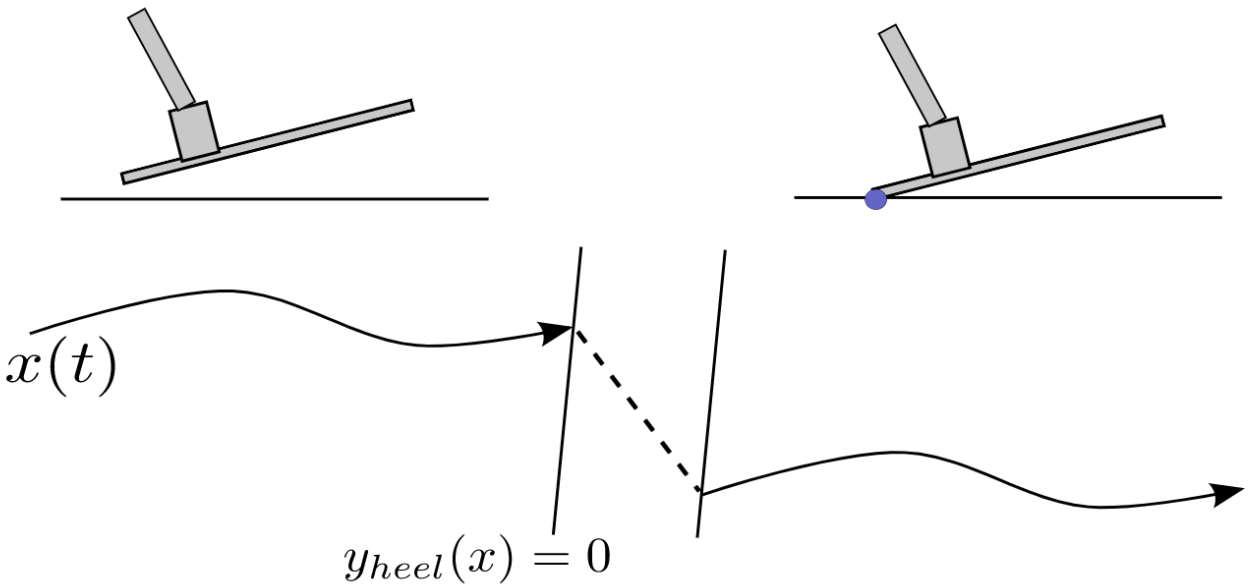


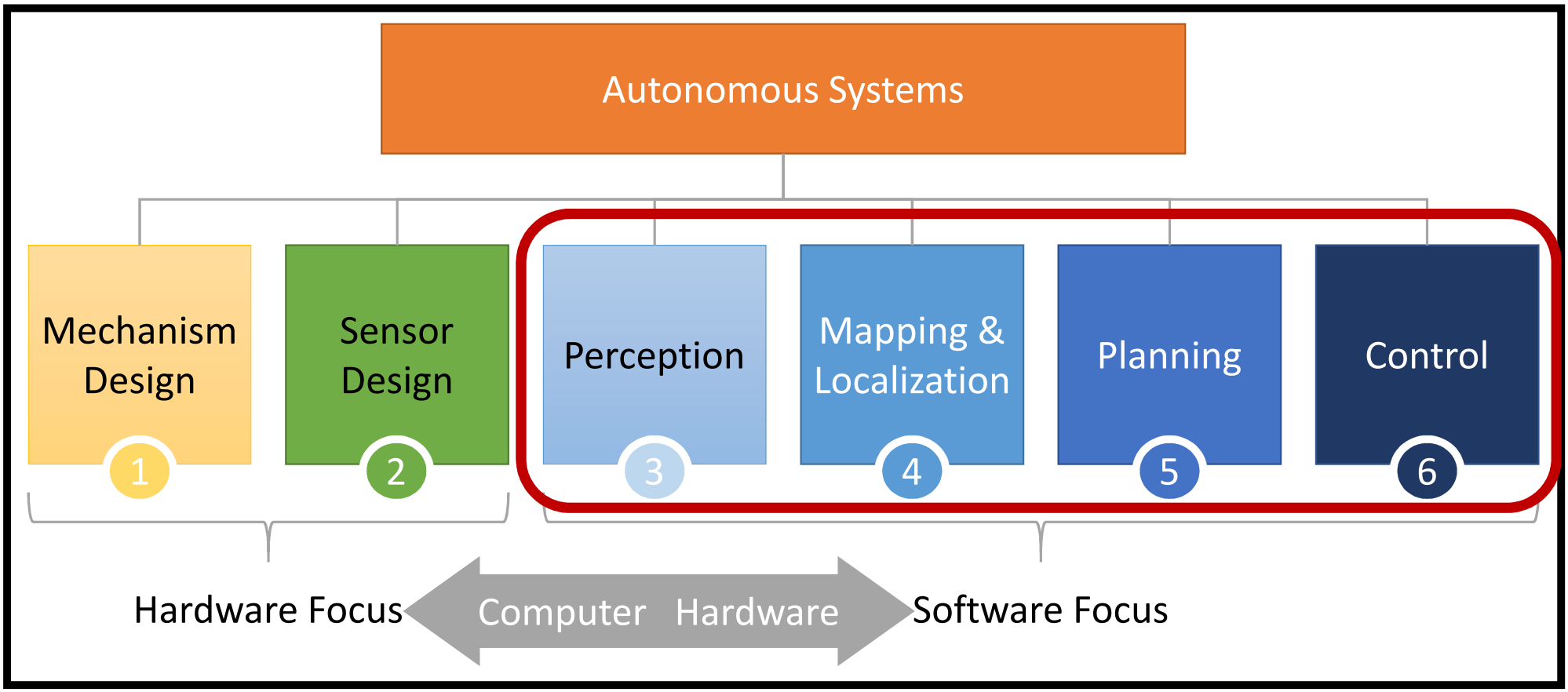
Figure 17.1 - Modeling contact as a hybrid system.

6 Practical Challenges for Control: Contact

6 Key Takeaways:

1. Real world autonomous systems need to use Feedback Control
2. Tracking controllers allow for simple control design and are quite effective in practice. Two common controllers are:
    1. PID with gain tuning
    2. LQR with cost function design
3. Using MPC allows for the planner to be the controller which enables more sophisticated control strategies
4. Contact is really hard!

Autonomous Systems / Robotics is a BIG space



Key Takeaways:

1. NNs running on accelerator chips solve most perception problems
2. The Kalman/Particle Filter uses probability to solve the localization problem but modeling and/or approximations are needed to run online
3. Mapping quickly becomes a memory storage problem
4. Stereo Depth and Visual Odometry also need acceleration to run online
5. Robot planning involves both task and configuration spaces
6. Collision checking can be expensive
7. Sample Based Planners (PRM, RRT, RRT*) leverage random search and are probabilistically complete but do not scale well to high dimensions
8. Trajectory Optimization finds locally optimal paths but is not complete or robust and (often) solved with (slow) off the shelf solvers
9. Tracking controllers (PID, LQR) work well in practice but MPC is a much more powerful (and computationally expensive) approach
10. Contact is hard and we (sometimes) use simpler models for tractability

Key Takeaways:

1. NNs running on accelerator chips solve most perception problems
2. The Kalman/Particle Filter uses probability to solve the localization problem but modeling and/or approximations are needed to run online
3. Mapping qu
4. Stereo Dept to run online
5. Robot plann s
6. Collision ch
7. Sample Bas search and are probabi dimensions
8. Trajectory Optimization finds locally optimal paths but is not complete or robust and (often) solved with (slow) off the shelf solvers
9. Tracking controllers (PID, LQR) work well in practice but MPC is a much more powerful (and computationally expensive) approach
10. Contact is hard and we (sometimes) use simpler models for tractability

There is SO much room for acceleration!!!!

And that’s everything!

http://bit.ly/CS249-Feedback-L2