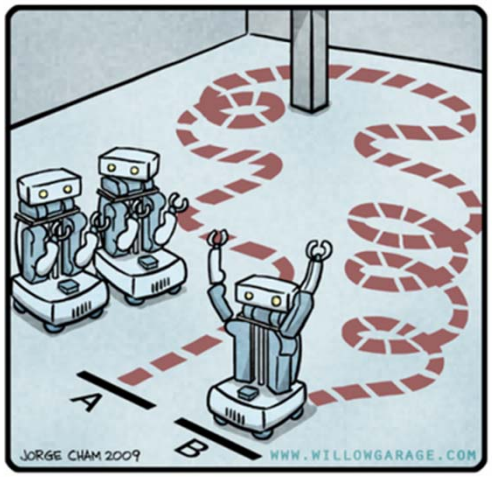
CS 189: Autonomous Robot Systems

Motion Planning and Legged Robots



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

Mark Petersen Spring 2019
Slides adapted from
Brian Plancher and Scott Kuindersma

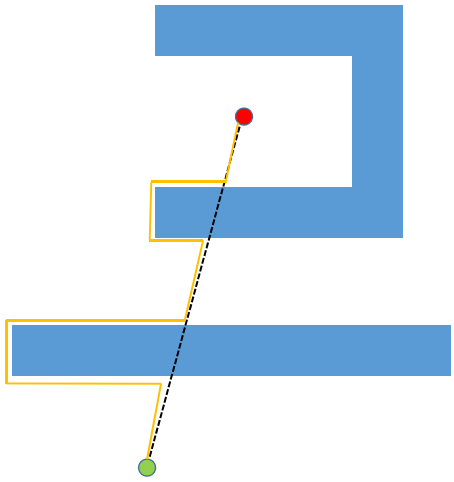
Our focus for today: Robot Motion Planning

- How do we plan motions for robots with various limitations?
- What do different planning algorithms require from the robot?
- How do we find feasible vs optimal paths?

Previously: Bug Based Path Planning

Algorithm (**input: s_0, s_{goal}**)

- Draw m-line connecting s_0 to s_{goal}
- Head towards s_{goal} along the m-line
- If obstacle is encountered follow edge of obstacle until you encounter m-line and are closer to s_{goal}
- Leave obstacle and repeat



Previously: Bug Based Path Planning

Why should you use Bug algorithms for Path Planning?

- Only assumes robot knows the goal direction
- Requires no map, only uses local information
- Does not require much memory

What if we have a map of the world and obstacles?

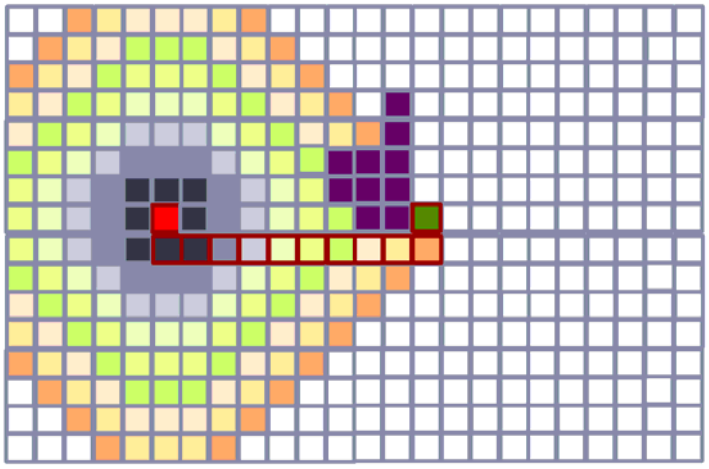
Use it to make an optimal plan!

Previously: Breadth First Search & A* Search

Algorithm (**input: s_0, s_{goal}, discrete obstacle map M**)

- Add s_0 to the tree of explored nodes
- Using M, expand the unexplored node that is closest to s_0
- Repeat until you expand s_{goal}

Note: For A* expand the node that has the lowest A* criteria



Previously: Breadth First Search & A* Search

Why should you use BFS or A* for Path Planning?

- Makes use of your map to find the optimal path
- Your planning space is already discrete

What if the planning space is continuous?

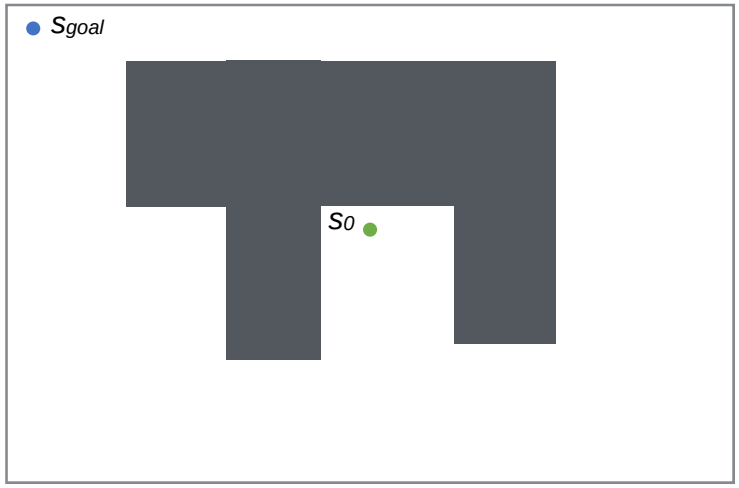
How can we handle continuous planning spaces?

- Discretize your continuous planning space
  - Works fine for low dimensional systems but you begin to suffer the curse of dimensionality
- Incrementally build up the graph in the continuous space instead of searching

Previously: Rapidly exploring Random Trees (RRT)

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

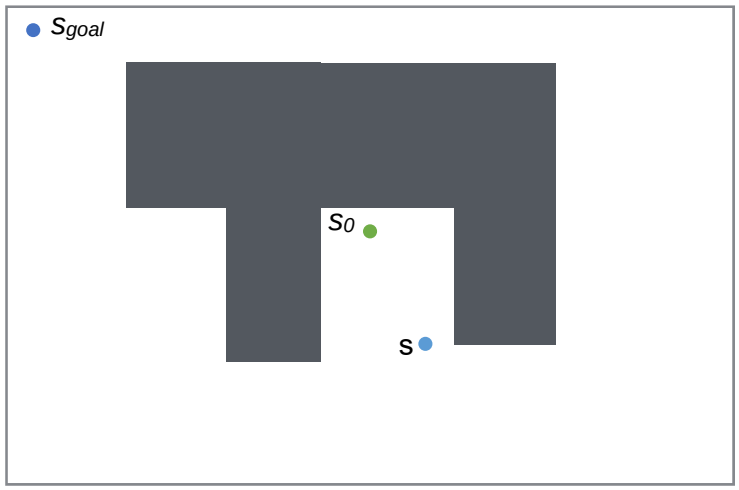
- Sample states $s \in S = R^{20}$ until s is collision-free
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



RRTs in action

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

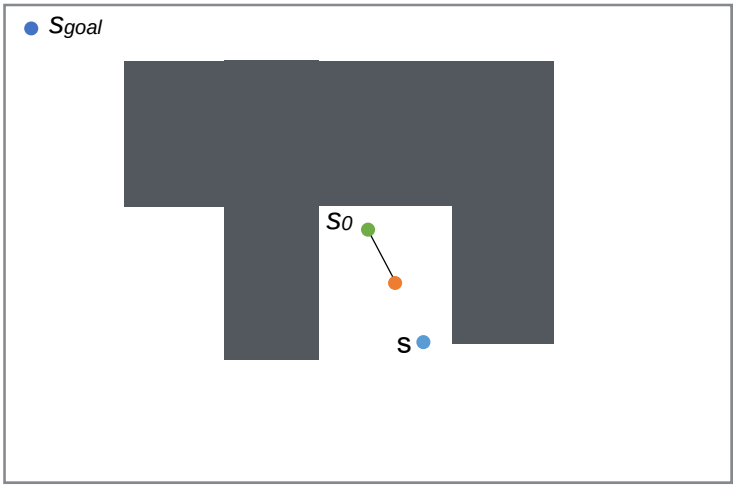
- **Sample states $s \in S = R^{20}$ until s is collision-free**
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



RRTs in action

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

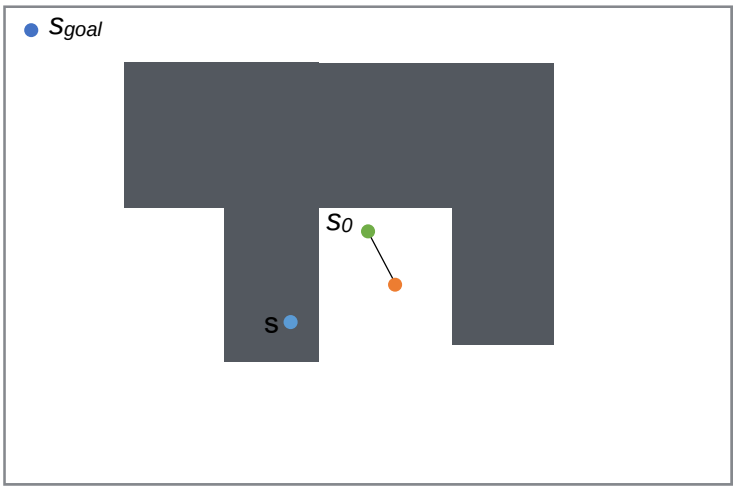
- Sample states $s \in S = R^{20}$ until s is collision-free
- **Find closest state $s_c \in T$**
- **Extend s_c toward s**
- **Add resulting state s' to T**
- Repeat until T contains a path from s_0 to s_{goal}



RRTs in action

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

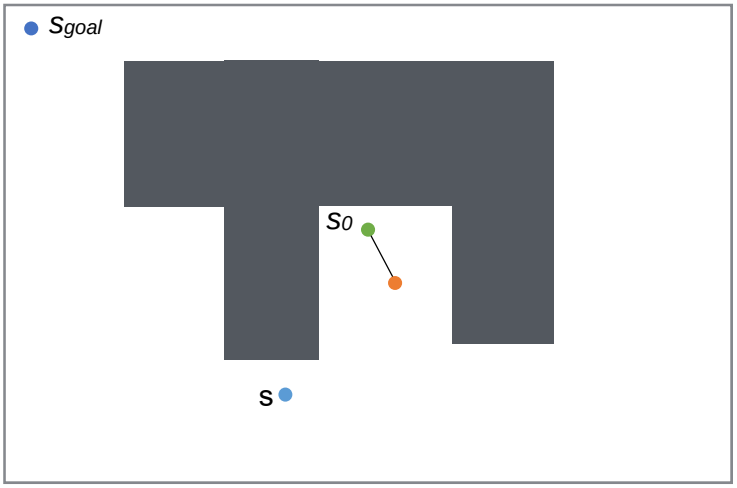
- **Sample states $s \in S = R^{20}$ until s is collision-free**
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



RRTs in action

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

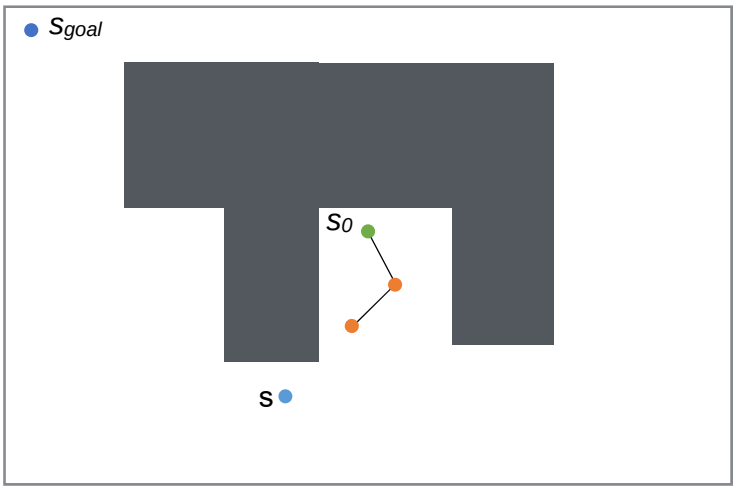
- **Sample states $s \in S = R^{20}$ until s is collision-free**
- Find closest state $s_c \in T$
- Extend s_c toward s
- Add resulting state s' to T
- Repeat until T contains a path from s_0 to s_{goal}



RRTs in action

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

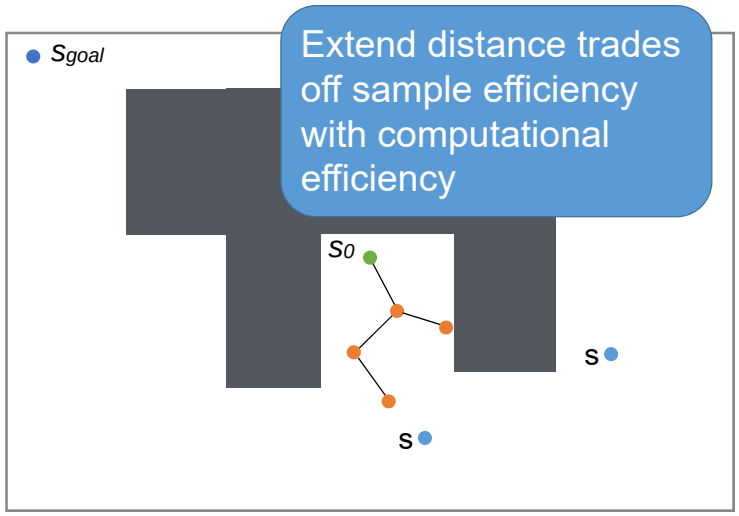
- Sample states $s \in S = R^{20}$ until s is collision-free
- **Find closest state $s_c \in T$**
- **Extend s_c toward s**
- **Add resulting state s' to T**
- Repeat until T contains a path from s_0 to s_{goal}



RRTs in action

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

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

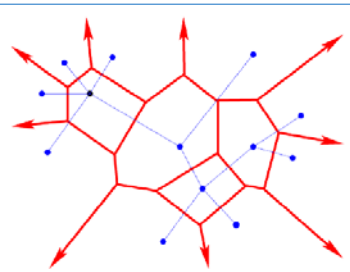


Randomness encourages exploration

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:



Properties of RRT

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

RRT is **probabilistically complete**!

- If there's a solution it will find it eventually
- Can still be slow for some problems, usually depends on how expensive collision checking is

Not optimal (cost of paths are not considered)

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

Variants of RRT

Goal Directed RRT:

- Don't always use a random sample, sample goal with some probability
- Biases the tree's exploration towards the goal

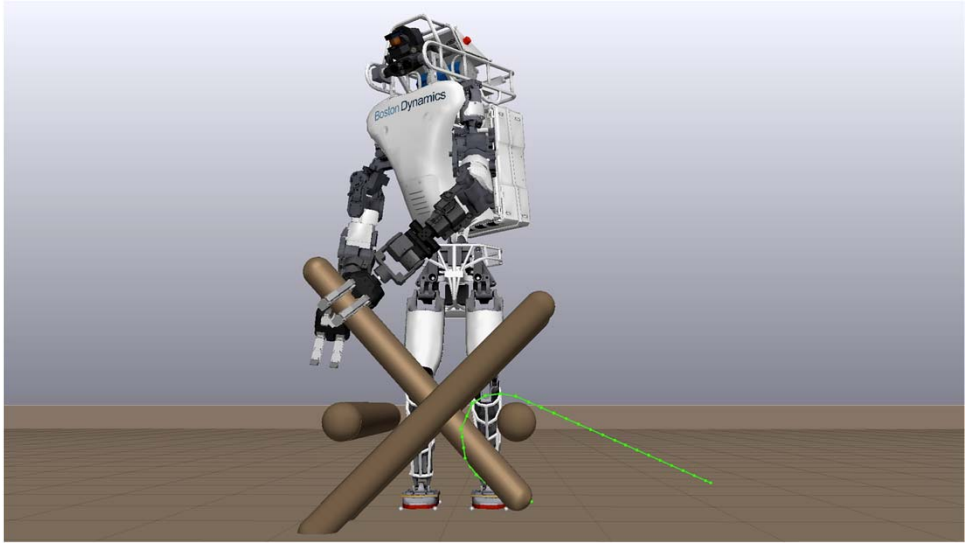
Bi-directional RRT:

- Build a tree from both the start and the goal state until they connect
- Enables the search to escape obstacle traps

RRT*:

- Perform a local rewiring of the tree after adding each node
- Allows the algorithm to find the optimal path (eventually)

RRT often works really well in practice

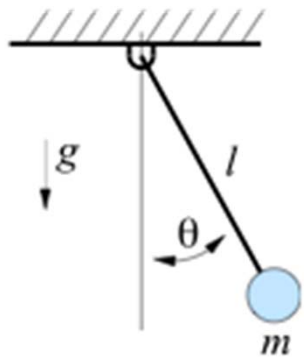


Ok so why can't robots use these awesome kinematic planning algorithms all the time and be better at life?!?

Dynamics (aka Physics)

Dynamics (aka How to break your planning algorithms)

The Simplest "Robot"

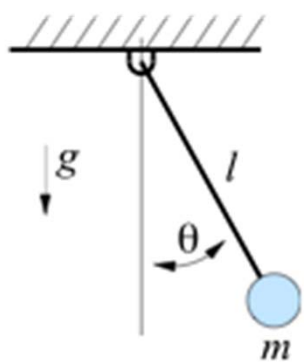


Q: Why do we need to track position and velocity?

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

Dynamics (aka How to break your planning algorithms)

The Simplest "Robot"



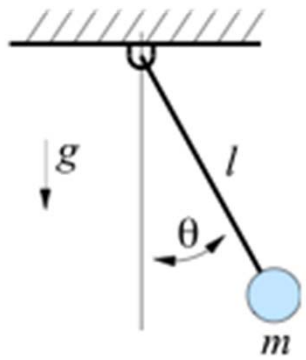
- States: $x = \{\theta, \dot{\theta}\}$ aka angle and angular velocity
- Actions: $u = \tau$ aka torque at joint
- Transitions: $\dot{x} = f(x, u)$ aka physics

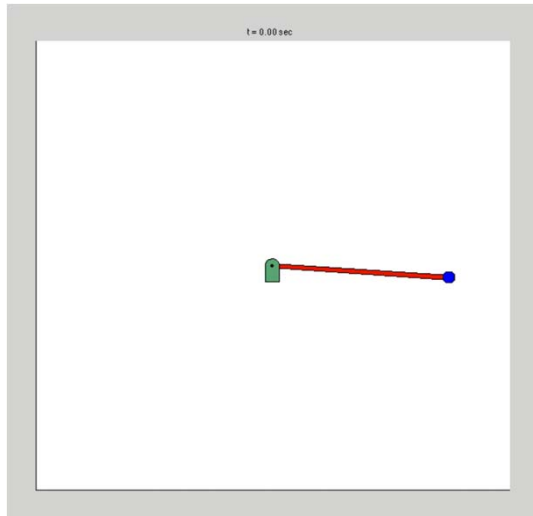
Euler Integrator

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

Dynamics (aka How to break your planning algorithms)

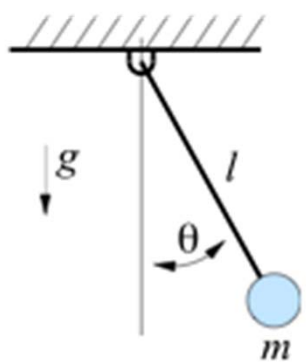
The Simplest "Robot"

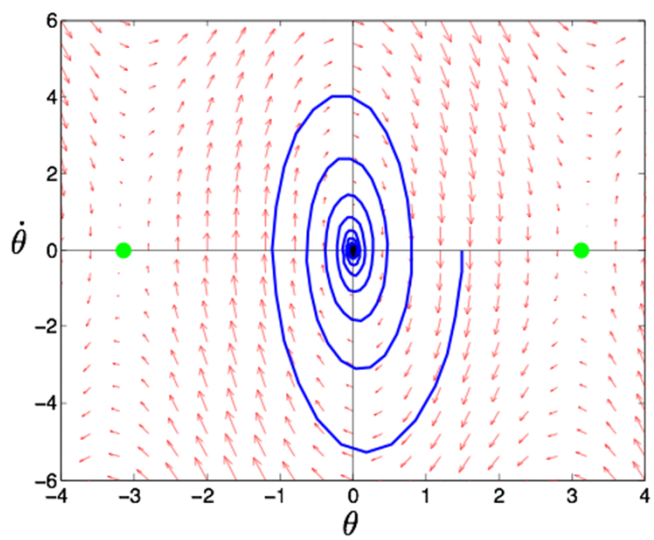




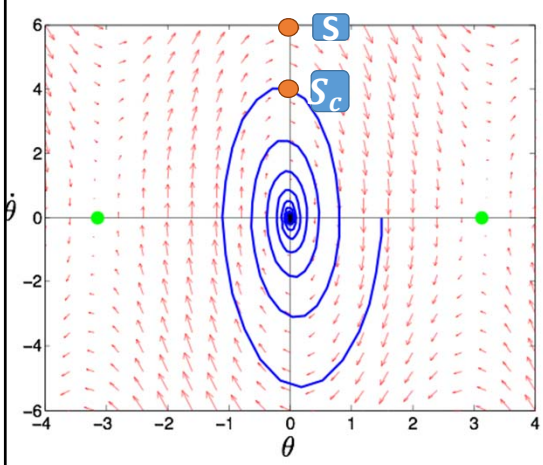
Dynamics (aka How to break your planning algorithms)

The Simplest "Robot"





Dynamics (aka How to break your planning algorithms)



Challenges for Dynamic RRTs

The "connect" operation is complex!

- We need to solve a **boundary value problem** (find a path from sc to s' such that 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** (Definitely asymmetric!)

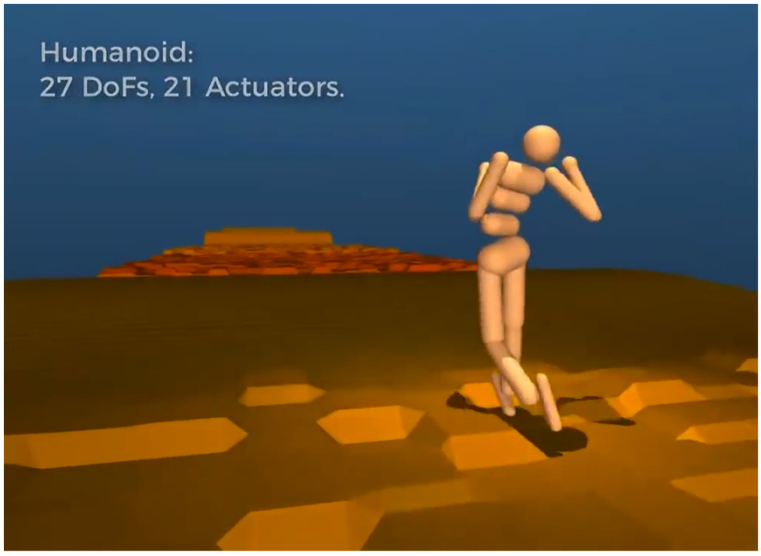
So what do we do?

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

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

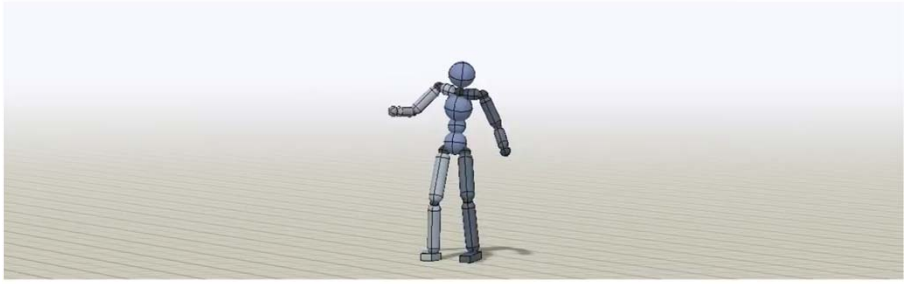
Can we make RL learn distance metrics or optimal plans?

Can Reinforcement Learning fix everything?



Can Reinforcement Learning fix everything?

DeepMimic: Example-Guided Deep Reinforcement Learning of Physics-Based Character Skills



Xue Bin Peng[1], Pieter Abbeel[1], Sergey Levine[1], Michiel van de Panne[2]

[1] University of California Berkeley

[2] University of British Columbia

Can Reinforcement Learning fix everything?

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

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

So what do we do?

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

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

Can we make RL learn distance metrics or optimal plans?

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

So what do we do?

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

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

Can we make optimal plans?

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

So what else can we do?!?

So what do we do?

Lots of math!

So what do we do?

The math isn't actually that scary and the computer will do most of it for you!

Trajectory Optimization

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

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

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

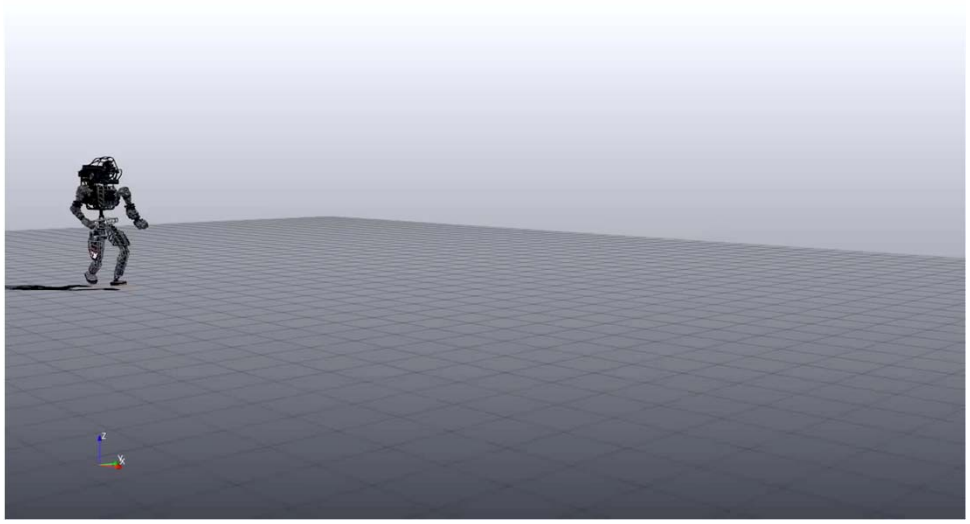
$$\text{subject to} \quad s_{k+1} = f(s_k, a_k)$$

Obey physics

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

Get to the goal

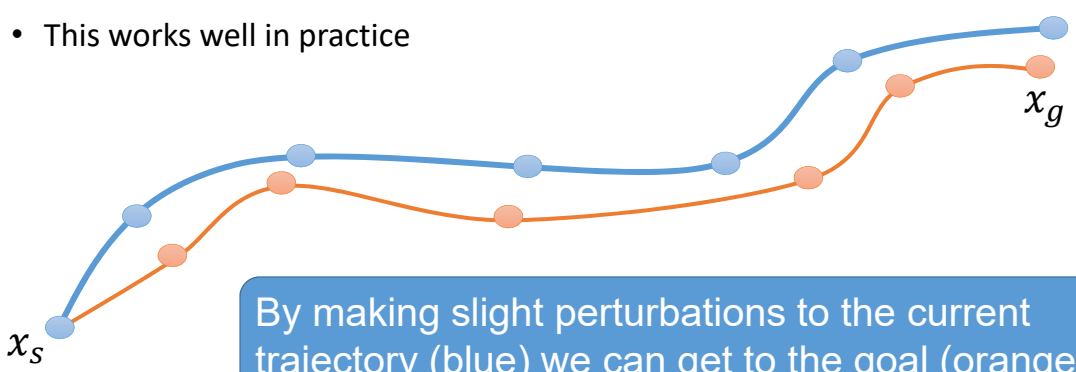
Trajectory Optimization on Atlas



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

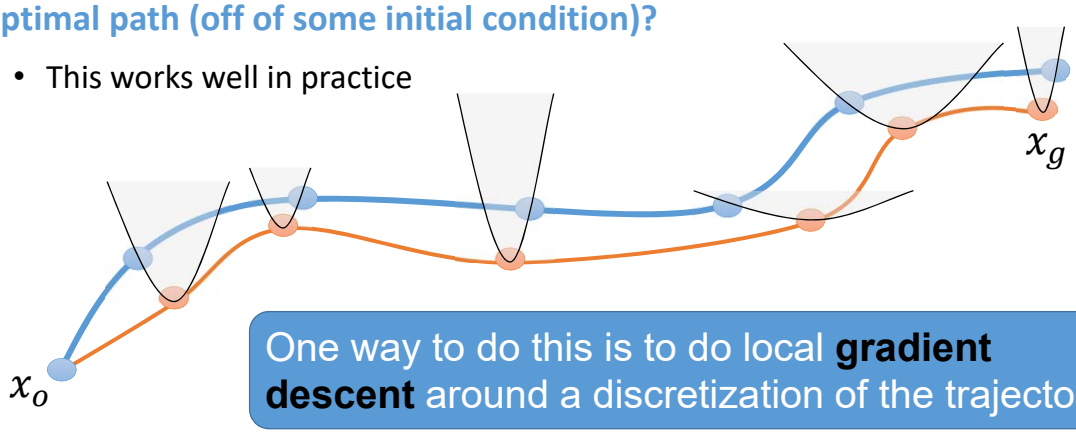


By making slight perturbations to the current trajectory (blue) we can get to the goal (orange)

Trajectory Optimization

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

- This works well in practice



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

Trajectory Optimization

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

- This works well in practice (think local search)

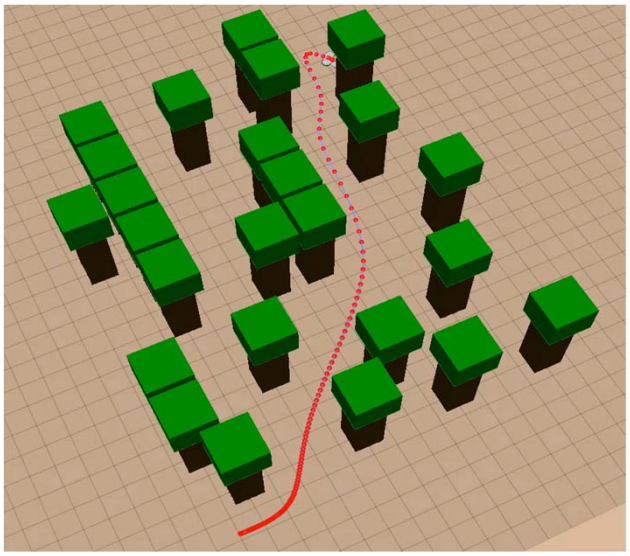
There are also a whole host of algorithms one can use to solve these problems including:

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

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

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

Quadrotor in Forest Trajectory Optimization



Trajectory Optimization in practice

How can I use trajectory optimization in practice?

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

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

Trajectory Optimization

So trajectory optimization solves everything right?

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

But….

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

Trajectory Optimization

So trajectory optimization solves everything right?

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

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**

Summary

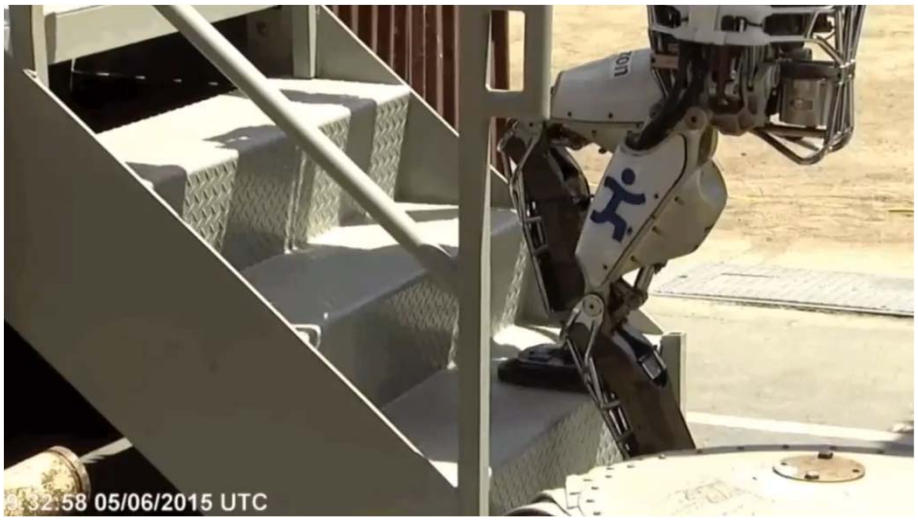
1. **Bug Algorithms**: allow you to plan a path using only local sensing information and goal direction, does not require a map a priori
2. **BFS and A***: a series of graph search algorithms that enable efficient shortest path calculations on discrete graphs
3. **RRT**: powerful sampling based motion planner that works well when distance metrics and connection operation is well defined
4. **Trajectory optimization**: a broad class of methods built on top of mathematical programming and "state of the art"

My Research

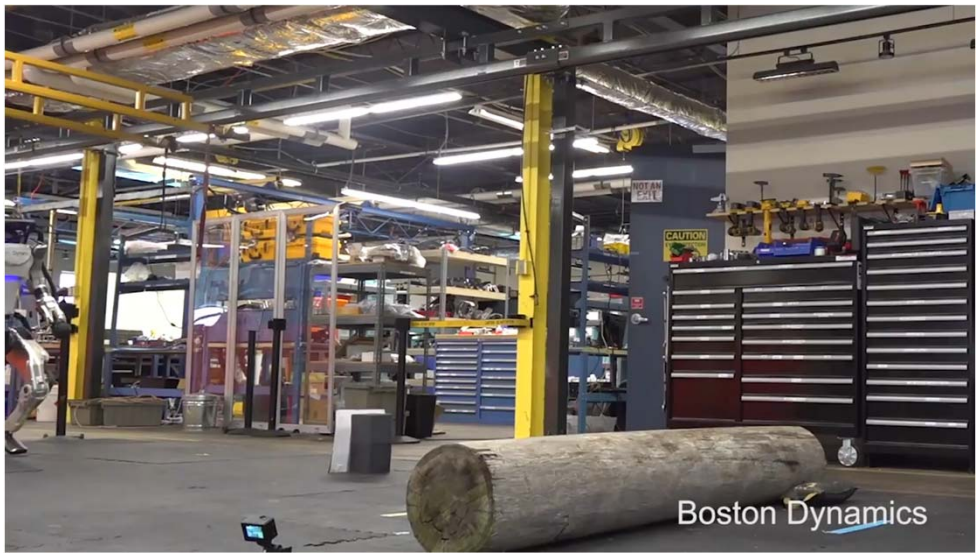
Legged Locomotion!

Incredible legged robots exist…

But controlling them is hard



Progress is being made



But there's still a technological gap

The Control Problem

- Given a legged robot
  - Has plenty of sensors
  - Strong actuators
- How do we program a computer to make it walk?
  - Copy human motion?
  - Use underlying principles of balance

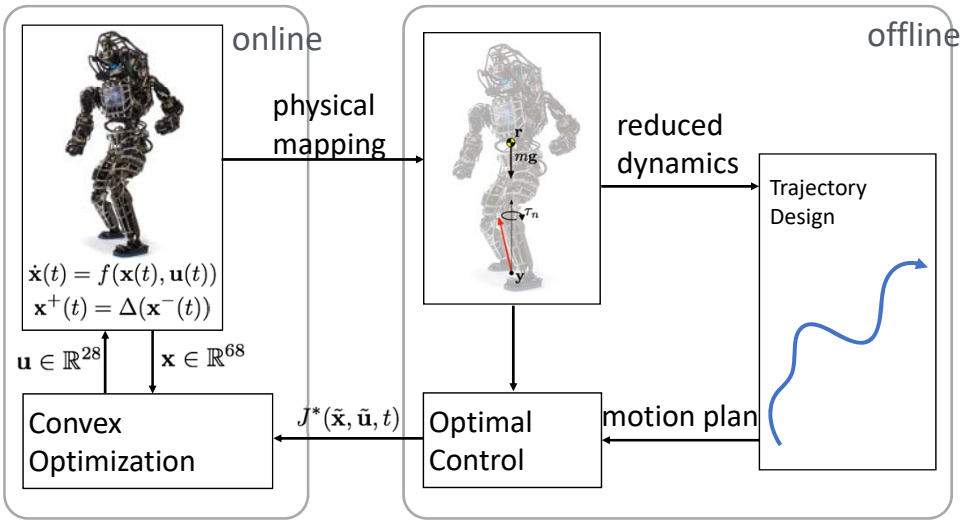
Humanoids that Balance

- Honda's Asimo
- Use physical insights to create stable walking
- Doesn't reason about all of its dynamics

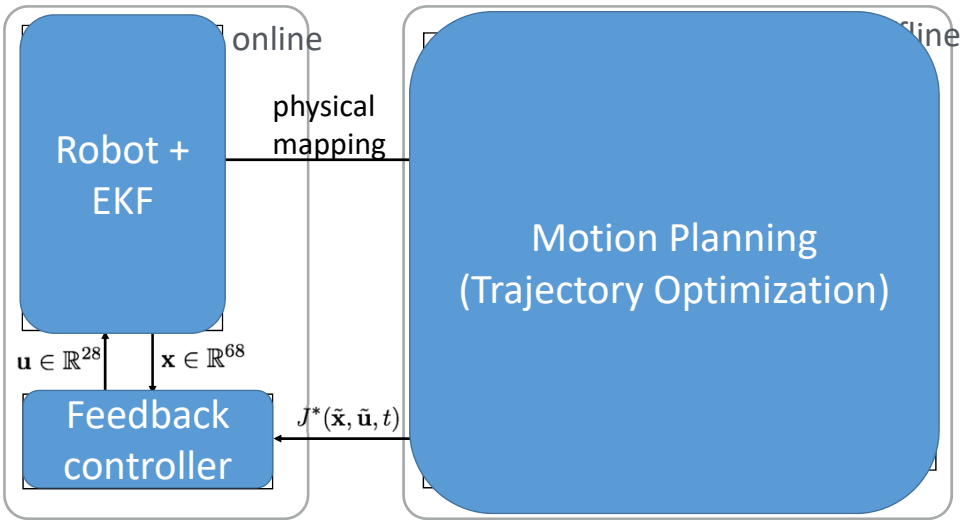
The Control Problem

- Given a legged robot
  - Has plenty of sensors
  - Strong actuators
- How do we program a computer to make it walk?
  - Copy human motion?
  - Use underlying principles of balance
  - **Optimization**: $\underset{x}{\text{maximize}}\ \text{score}(x)$

The full control loop



The full control loop



Trajectory Optimization for Legged Robots

Can we plan using the full dynamics of the robot?

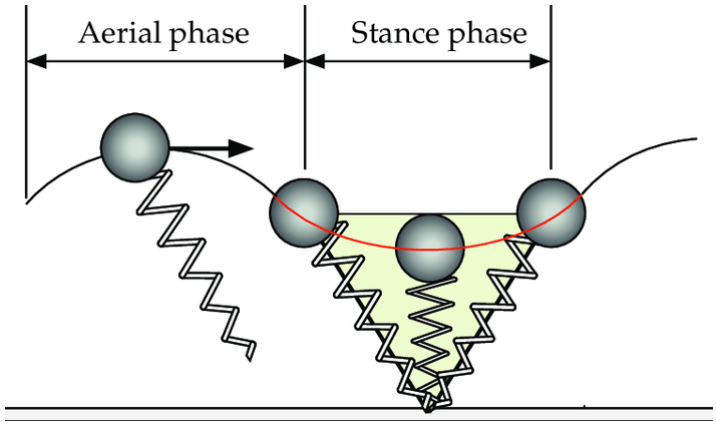
- Yes, but not in real time
  - Non-linear dynamics are made worse by contact

How can we make planning faster?

- Use simpler models
  - Lower dimensional systems will optimize faster
  - We're trying to regulate underlying principles of balance
  - Can later be tracked by the full robot

Simple Model for Walking: Spring Loaded Inverted Pendulum (SLIP)

- Model the entire mass of the robot as a point mass
- Model the leg(s) currently supporting the robot as a spring
- Resulting model only has 5 degrees of freedom
- Dynamics are much nicer (only slightly nonlinear)



Poulakakis, Ioannis. (2008). Stabilizing Momopedal Robot Running: Reduction-by-Feedback and Compliant Hybrid Zero Dynamics.

Tracking the planned path

How do we stay on the path we planned?

- Use the calculated inputs from the trajectory optimization?
  - Open loop control almost always falls off-course
- Use PID feedback?
  - Works great when the system is simple and linear
  - Won't always drive this system in the direction we want
- Do another optimization?
  - That might just work!

Tracking the planned path

The goal is to stay on the trajectory or get back to it

- At each time-step find inputs that get you as close to desired trajectory as possible

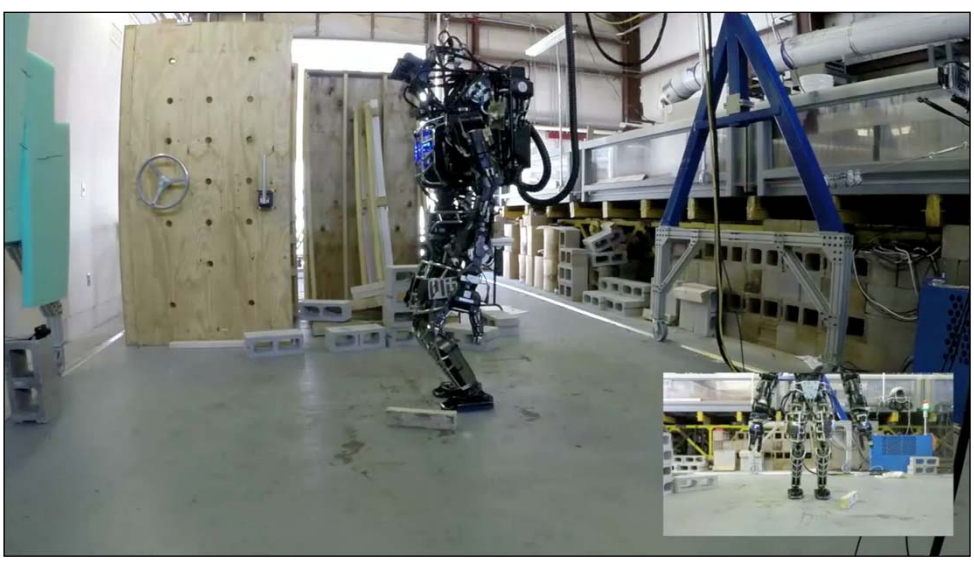
$$\underset{\ddot{q},u,\lambda}{\text{minimize}} \quad g(\ddot{q},u,\lambda) \quad \text{(one-step cost-to-go to trajectory)}$$

$$\text{subject to} \quad H(q)\ddot{q} + C(q,\dot{q}) = Bu + J(q)^T\lambda \quad \text{(dynamics)}$$

$$\lambda = \Sigma\beta_i w_i, \qquad \beta_i > 0 \quad \text{(friction)}$$

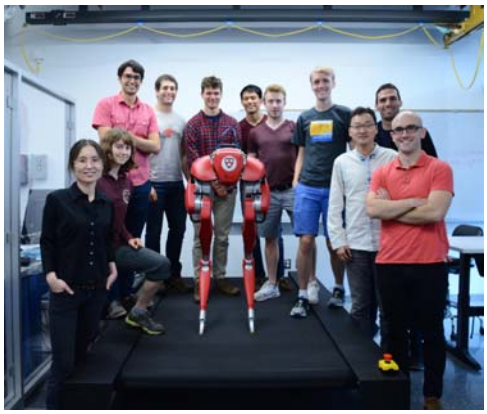
$$u_{min} < u < u_{max} \quad \text{(torque limits)}$$

Does it work when we put it all together?

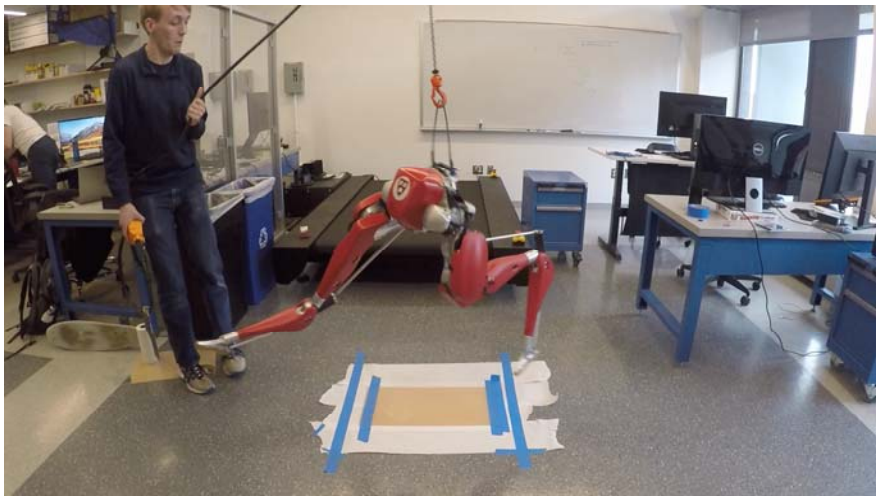


What am I doing?

Putting these algorithms onto Cassie!



The real reason this class uses Turtlebots

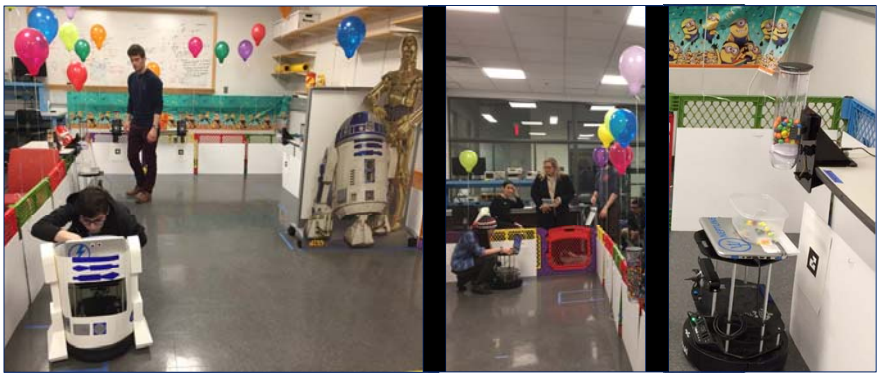


Final Project

This is You!

Inspired by Fetch Robotics!

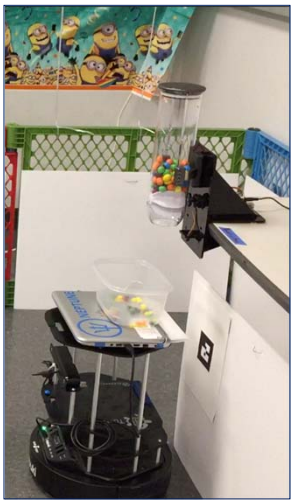
Pictures From Last Year

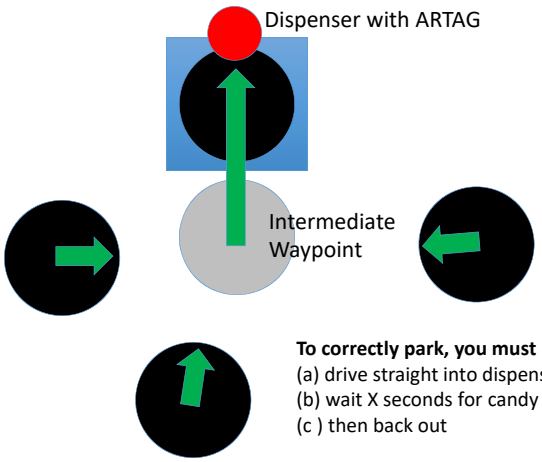


Your robot should take an order at a station, navigate to the correct dispenser, correctly park under the dispenser to trigger candy, and come back to home.

You will know ahead of time the layout of the candy factory. Home station are where you take orders, candy dispensers will dispense when you park underneath. Each home station and candy dispenser will be identified with a unique ARtag

Milestone 1: Parking (due next Friday)

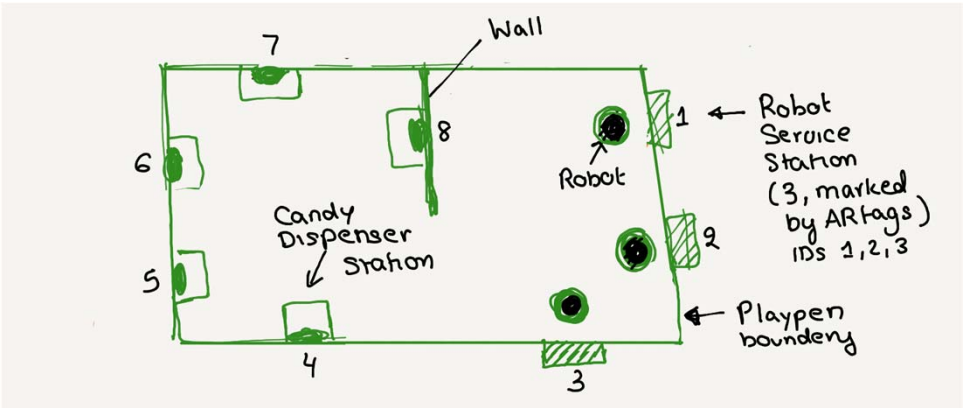




To correctly park, you must
(a) drive straight into dispenser
(b) wait X seconds for candy
(c) then back out

Use the ARTAG to determine where you are relative to the dispenser, then use path planning to drive to the intermediate waypoint and get into correct orientation before trying to park.

Future Milestones

Milestone 2: Navigating Between Base and Dispensers

Milestone 3: Multiple Robots working Together