ECE209AS (Fall 2025)

Computational Robotics

Prof. Ankur Mehta ⟨mehtank@ucla.edu⟩

Challenge problems: Other problems to work on

Week 1: Systems and State

Mapping between continuous / discrete space system formulations

How might you map between our two system formulations? That is, given a system described as an MDP, how might you create a functional representation of its dynamics and observations? Or vice versa? What assumptions or constraints do you need in order to get to various mappings, and what can that tell you about the characteristics of the various representations?

Gridworld meta-design

Can you come up with a specification language that can represent a class of gridworld generalizations? How general can you be? Can you automatically build a simulator / visualizer from the specification?

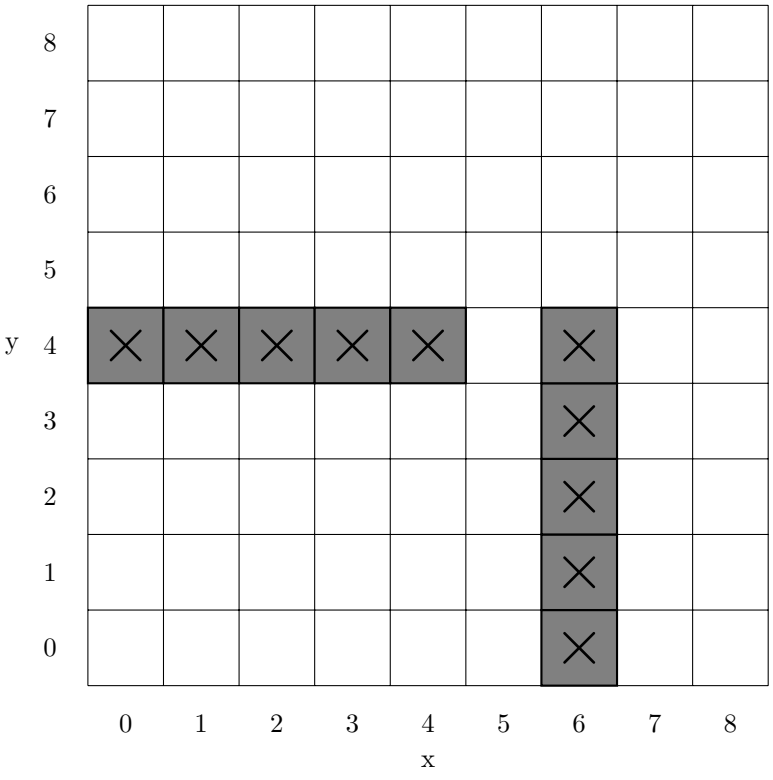
What is the complete class of systems your specification language can encode, and can you use that to explore new, unexpected 2D discrete-space systems?

See here for additional possible system dynamics:

- https://en.wikipedia.org/wiki/Fairy_chess_piece

Week 2: Planning / control on MDPs

Bottlenecked MDPs



Consider for example the gridworld seen above. It has a bottleneck in the state space that we might be able to exploit to reduce the computation required to solve an MDP problem on this space. How might we partition a problem at this bottleneck to create smaller problems, and how much computation do we save by doing so? What are the requirements on the problem that allow for such a decomposition?

Similarly, consider a problem that requires multiple steps to acquire the reward, e.g. first pick up a key, then unlock a chest. What would this state space look like, and how would you characterize its bottleneck? What if there are more than two steps? What if some of those steps can be done out of order? How do these problem scale with the number of steps in the solution?

How can we computationally identify the existence of such bottlenecks from a problem statement, and what is the complexity of that process? How "big" does the bottleneck need to be to justify this process?

Composing solved MDPs

Let's say we have two similar MDPs (differing only in their state space and associated transitions and rewards) that we know solutions for, i.e. optimal policies π_i^* and value functions V_i^* for MDPs $\{S_i, A, P_i, R_i, H, \gamma\}, i \in \{1, 2\}$. Given some new transition probabilities and rewards $\{p_{12}\}, \{r_{12}\}$ between the two original state spaces, under what conditions can we efficiently solve the combined MDP $\{S_0, A, P_0, R_0, H, \gamma\}$? We have:

$$S_0 = S_1 \cup S_2$$

$$P_0 = \{p_0(s, a, s')\}; \quad p_0(s, a, s') = \begin{cases} p_1(s, a, s') & \text{if } s, s' \in S_1 \\ p_2(s, a, s') & \text{if } s, s' \in S_2 \\ p_{12}(s, a, s') & \text{otherwise} \end{cases}$$

$$R_0 = \{r_0(s, a, s')\}; \quad r_0(s, a, s') = \begin{cases} r_1(s, a, s') & \text{if } s, s' \in S_1 \\ r_2(s, a, s') & \text{if } s, s' \in S_2 \\ r_{12}(s, a, s') & \text{otherwise} \end{cases}$$

What if we don't know P_1, R_1, P_2, R_2 at all, only $V_1^*, \pi_1^*, V_2^*, \pi_2^*$ of the original MDPs?

Week 3: Discretization and function approximation

Basis function meta-design

Given a discrete MDP parameterized by some size N (e.g. gridworld side length, number of agents, joints in a multi-DOF system, etc.), can you algorithmically determine a basis set for a function approximator that scales to larger N based on exact solutions to the problem for several smaller N?

Just-in-time (JIT) improvement on discretization planning

Can you come up with a process (algorithm) that successively refines a continuous space MDP solution? You should be able to interrupt the process at any time and return a reasonable policy; the more computation it is allowed, however, the better the resulting policy. For example, you could start by using a coarse grid to compute a no-lookahead, 1-nearest-neighbor policy. Then, given more time, you can add gridpoints to refine the resolution. Can you add on to past computations to improve on the policy?

What about if you'd like to use more nearest neighbors? Or take additional lookahead steps? Can you algorithmically determine "the best" next refinement to take given the current policy / value function and saved computation?

Week 4: Graph-search based motion planning

RRT on complex dynamical systems

Implement and evaluate an RRT planner on a continuous space system. Be sure to explore reversible dynamics and bi-directional trees, challenging obstacles and corresponding extensions to the algorithms (RC-RRT), and path-optimality (RRT*).

Converting arbitrary MDPs to graph-search based methods

1. If system dynamics include stochastic transitions, we could conceivably build a graph with probabilistic edges. What is the right notion of completeness in such a case? How might we adapt graph search approaches to best accommodate these relaxed constraints?
2. Since the state / state space is a constructed quantity, we could try to reformulate the mathematical framework of an arbitrary MDP goal to match the constraints of a graph-search based method. How? How would this approach compare (in both computational cost and resulting plan) to our past MDP solving processes under various classes of reward functions between the highly specific graph search based constraints and the completely general MDP formulation?

Week 5: Linear quadratic regulators

Towards LQR trees

When the particle-on-a-numberline is under the influence of a highly nonlinear potential field, e.g. $\phi(x) = 2\cos(x)$, the resulting nonlinear system can't be driven to a goal in any simple manner. Instead, can you build a PRM, checking connectivity between pairs of points by identifying whether one point can be reached from another using a linearized LQR?

Iterative LQR

Can you drive the trajectory of a highly nonlinear system (e.g. the underactuated wobbly rocket) to a specified state by linearizing around the setpoint, then using an LQR controller? How far away (in state space) can the system be before this approach fails to converge?

If you're too far, can you instead get to the goal state in multiple steps? Come up with a potential sequence of states that approaches the goal state from your initial state (perhaps based on naive interpolation), then iterate a linearized LQR around each successive goal state. Regularly recompute based on the current state to avoid accumulation of errors. What can you observe about this approach to planning/control?

Week 6: Bayesian filtering and POMDPs

Action / observation lookahead

From a given initial belief state of a system, comprehensively evaluate every possible belief state evolution over 4 time steps (i.e. 4 actions, each followed by an observation). How many belief states (and associated probabilities) will you therefore need to compute?

To each leaf node in this tree, assign a value of the weighted optimal value $V(Bel) = \Sigma_s Bel(s)V^*(s)$, where the optimal value V^* is given by the fully observable MDP on the same system. Use these values to compute the best action to take from the initial belief state.

How does this resulting policy compare to the optimal policy π^* of the fully observable MDP on this system when starting in a known initial state? How does this change based on the depth of the lookahead tree?

Run several example trajectories under this policy starting from a fully unknown (uniform) initial belief state from various start states. Recompute a brand new 4-step lookahead to determine the next action from the belief state at every timestep.

Forwards / backwards Bayesian filtering

A Bayesian filter computes the belief state at the current time based on the history of actions and observations. However, what was likely at a given time based on the history of information may not remain likely at that time when considering future information, i.e. observations from later times. Use a similar Bayesian approach, propagated backwards in time, to compute a *smoothed* belief state history based on the entire time series of actions and observations.

Can you come up with a metric on the belief state that captures the likelihood of that belief state given what's known about the noise in the system dynamics and measurement model? How does the backwards smoothing pass of this

filter affect that metric? Can you come up with a predictive way of potentially improving this metric without having to compute a full backwards pass each timestep?

Week 7: Kalman filtering and SLAM

Range only SLAM

Consider a holonomic rigid body (3DOF) robot on the 2D plane, with system model $\mathbf{x}[t+1] = \mathbf{x}[t] + \mathbf{B}\mathbf{u}[t]$, with constant but unknown diagonal matrix $\mathbf{B}$. Your sensor is a single range sensor that returns the distance to the wall directly in front of you. See Figures 1, 2 for an example.

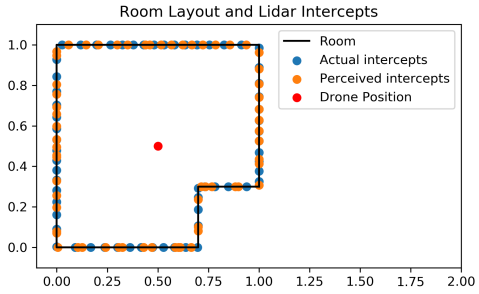


Figure 1: Example range-only slam scenario

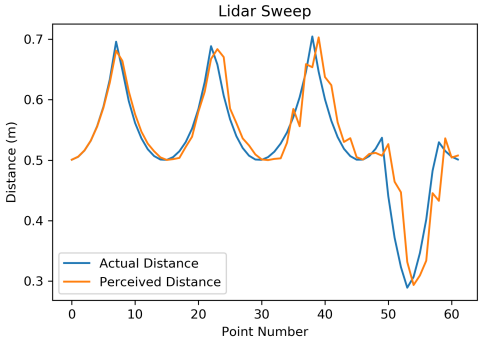


Figure 2: Time series range-only sensor data corresponding to a possible scenario

Set up and solve the SLAM problem on this system inside a closed room of varying geometries. Build a metric on the uncertainty of your map to evaluate the performance of this algorithm.

Include additive white Gaussian noise (AWGN) to the system dynamics and measurement model. How does this affect your mapping performance?

Extend this to an active SLAM problem, where you have a goal of getting to the center (centroid) of the room in minimal time.

Filtering with unknown inputs

Consider a general Kalman filter on an arbitrary linear system with additive white Gaussian noise (AWGN); the system matrices F, G, G_w, H, Q, R (or A, B, B_w, C, Q, R) are all known. However, the disturbance (process noise) w has an unknown nonzero mean. Build an estimator to compute a belief over the disturbance mean along with the state estimate.

You can extend this concept of nonzero process noise by instead considering uncertainty over the input sequence $u[t]$. Given a collection of candidate input sequences ("maneuvers") $u_j[t], j \in \{1, 2, \ldots, N\}$, evaluate the likelihood of each maneuver over time given only observations.

Week 8: Reinforcement Learning

Compare value-based and policy-based approaches to planning on a practice system.
Make sure you begin with a precise mathematical formulation of the problem. Explicitly define:

- The state space as a set
- The action/input space as a set
- The system dynamics as a probability distribution over the next state given the current state and action/input
- The sensor model as a probability distribution over the measurement (observation/output) given the current state (and action/input)
- The reward or cost defining a planning/control goal as a function of transition

For systems with adversaries (e.g. pursuit/evasion, liar's dice), define a (heuristic) policy for the adversary, then incorporate that into the system dynamics.

Build, validate, and demonstrate a complete end-to-end simulator of this system. Be sure to include parameters defining the size of the problem, all noise/uncertainty terms, and any other generalizations you would like to add. Make sure this system is modularized to allow arbitrary actions/inputs, including human-input or generated. Return both the ground truth actual state as well as the state-dependent sensor measurements. When building and evaluating the learning-based planning approaches, first use the actual ground truth state with purely deterministic dynamics, then work in noise, sensing, and noisy sensing.

Create a function approximation (select a function basis) on the Q value function, then run Q-learning with experience replay on your system. Play around with various hyperparameters, including exploration vs exploitation rates, gradient step size, choice of function bases, etc. Demonstrate the improvement in performance with increased iterations through the training loop.

Create a function approximation on a stochastic policy π, then implement, validate, and demonstrate your policy π both in isolation and connected to your system. Run the basic policy gradient approach (using the reward of the trajectory as the baseline) and demonstrate the performance improvement over training runs. Again, play around with various hyperparameters of this algorithm.

Compare computational complexity/runtime, performance, and convergence rate of the two approaches.