Day 5. Thompson Sampling

NPEX Reinforcement Learning

July 30, 2021

Jaeuk Shin, Minkyu Park

CORE
Control + Optimization Research Lab

Thompson Sampling

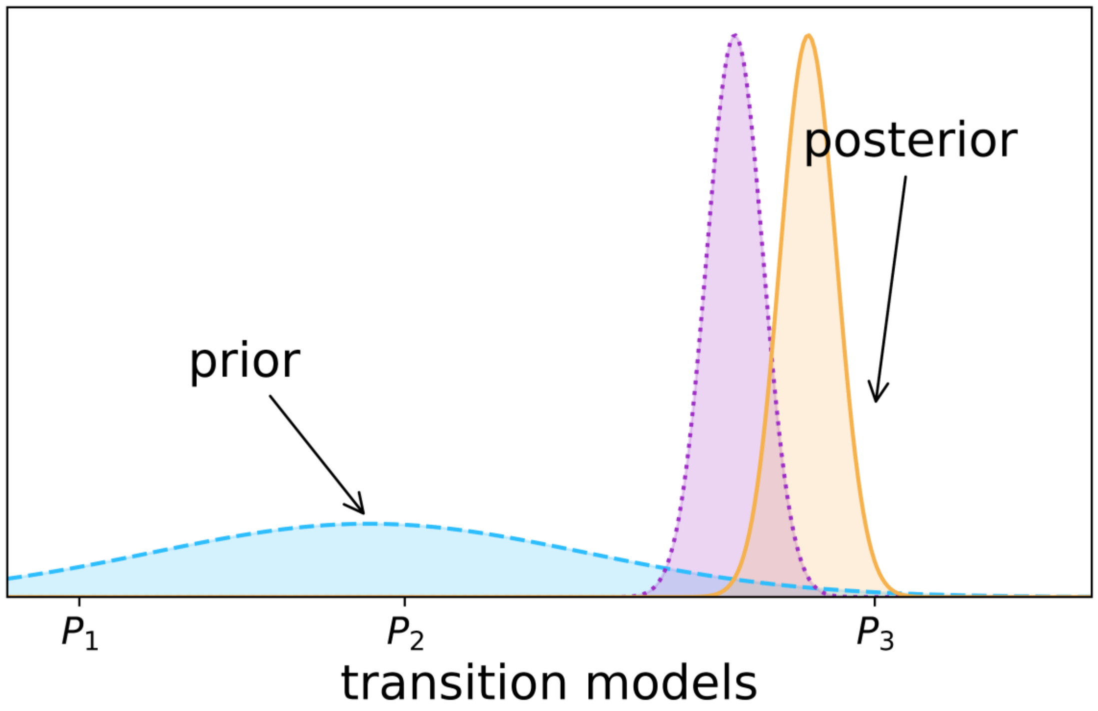
Key Idea : When the underlying dynamics is unknown, model-based approaches always repeat the following steps:

1. Act optimally according to our model of the dynamics.
   - Model Predictive Control
   - **Dynamic Programming** (Finite MDP, Linear Quadratic Control)
2. Observe the result of the action.
3. Update our model based on new information.
   - Least-square Fitting
   - **Bayesian update** $\Longleftarrow$ Spirit of Thompson Sampling

Thompson Sampling

What do we mean by **Bayesian update**?

- Using Thompson sampling, we do not update our model deterministically.
- Instead, whenver new information is given, we update the **distribution of models!**



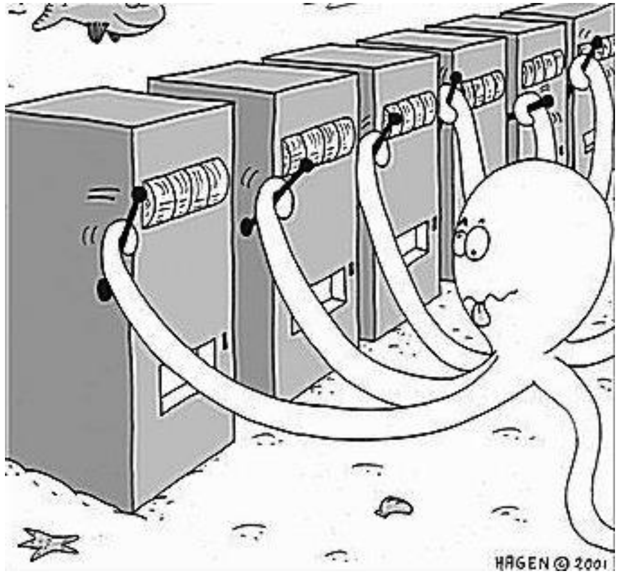
Thompson Sampling

3 applications of Thompson Sampling in RL:

1. Multi-armed Bandit Problem
2. Model-based RL via Thompson Sampling
3. Learning Linear Quadratic Control vis Thompson Sampling

Application 1. Multi-armed Bandit Problem

Multi-Armed Bandit Problem



1-step MDP problem

Framework to handle **exploration-exploitation**

With prior belief $\approx p(r|a)$, build polcy π

Take action, get reward (r, a) to update $p(r|a)$ via Bayes' rule

Regret : $r(a^*) - r(\pi)$

2 approaches

Use action to maximize $E[r|a]$ - greedy selection

Sample $r \sim p(r|a)$ and choose action - Thompsom sampling

Since we utilize *model* $p(r|a)$, it is model-based RL approach

CORE
Control + Optimization Research Lab

Multi-Armed Bandit Problem

Implementation : travel time optimization

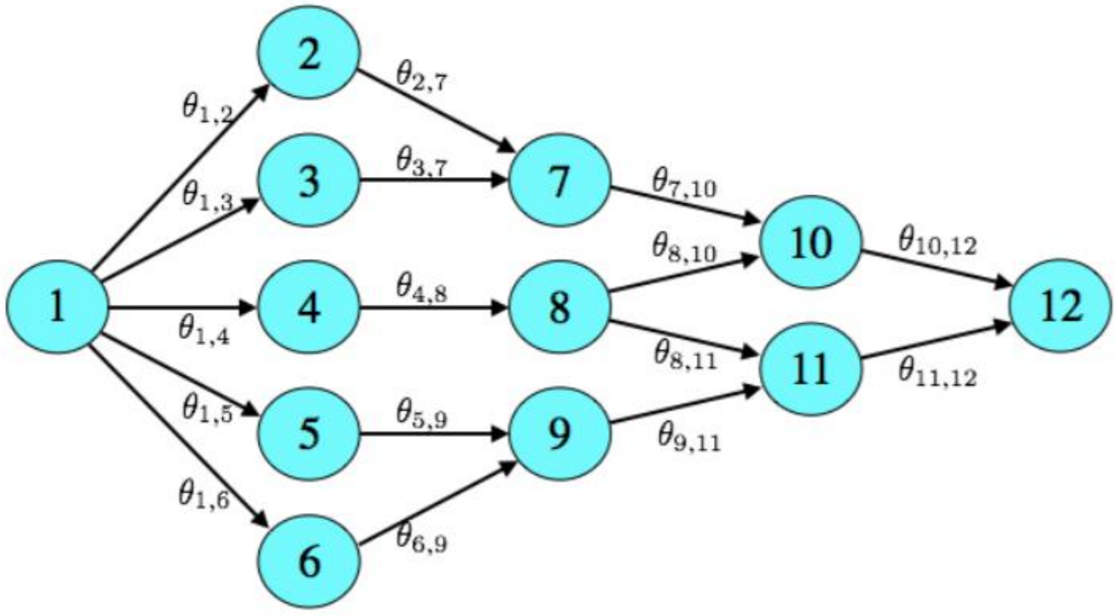


Figure 1.1: Shortest path problem.

Table for road number naming

0	5	10	14
1	6	11	15
2	7	12	
3	8	13	
4	9		

Time on road θ follows log-gaussian : $y_t \sim \mathcal{N}(\log(\theta) - \tilde{\sigma}^2/2,\ \tilde{\sigma}^2)$

There are total 16 path (16 actions) available

CORE
Control + Optimization Research Lab

Multi-Armed Bandit Problem

Implementation : travel time optimization

$\theta \sim \mathcal{N}(\mu_e, \sigma_e)$

μ_e is $d_e + w$, where d_e is distance of each path and w is random noise

Agent will use prior $\theta \approx \mathcal{N}(d_e, \hat{\sigma_e})$

When y_t is given, agent can perform Bayes' rule to update its $(\hat{\mu}_e, \hat{\sigma}_e)$

Reward will be given with $-\sum y_t$

CORE
Control + Optimization Research Lab

Multi-Armed Bandit Problem

Implementation : travel time optimization

```python
def sampling(self):
    self.theta = []
    for mu_e, sigma_e in zip(self.mu, self.sigma):
        theta = np.random.normal(mu_e, sigma_e)
        self.theta.append(np.exp(theta))

    action_score = []
    for path in self.path_set:
        time = 0
        for e_idx in path:
            # Thompson sampling samples [y_t] to choose action
            mean = np.log(self.theta[e_idx]) - 0.5*(sigma_wave**2)
            std = sigma_wave
            y_t = np.random.lognormal(mean, std)
            time += y_t
        action_score.append(-time)

    return np.argmax(action_score)
```

Each path corresponds to bandit action

Sample $r \sim p(r|a)$ and choose action - Thompsom sampling

CORE

Control + Optimization Research Lab

Multi-Armed Bandit Problem

Implementation : travel time optimization

```python
def greedy(self, eps=0):
    self.theta = []
    for mu_e, sigma_e in zip(self.mu, self.sigma):
        theta = mu_e
        self.theta.append(np.exp(theta))

    action_score = []
    for path in self.path_set:
        time = 0
        for e_idx in path:
            # Greedy poilcy use E[y_t|theta] to choose action
            # TODO : calculate E[y_t|theta]
            # Hint : for lognormal distribution ( log(m)-0.5*(n**2), n ), m = E[y_t|theta]
            # Hint : find y_t at sampling function!
            y_t = self.theta[e_idx]
            time += y_t
        action_score.append(-time)

    r = np.random.uniform(0,1)
    if eps < r:
        return np.random.randint(0,len(self.path_set))
    else:
        return np.argmax(action_score)
```

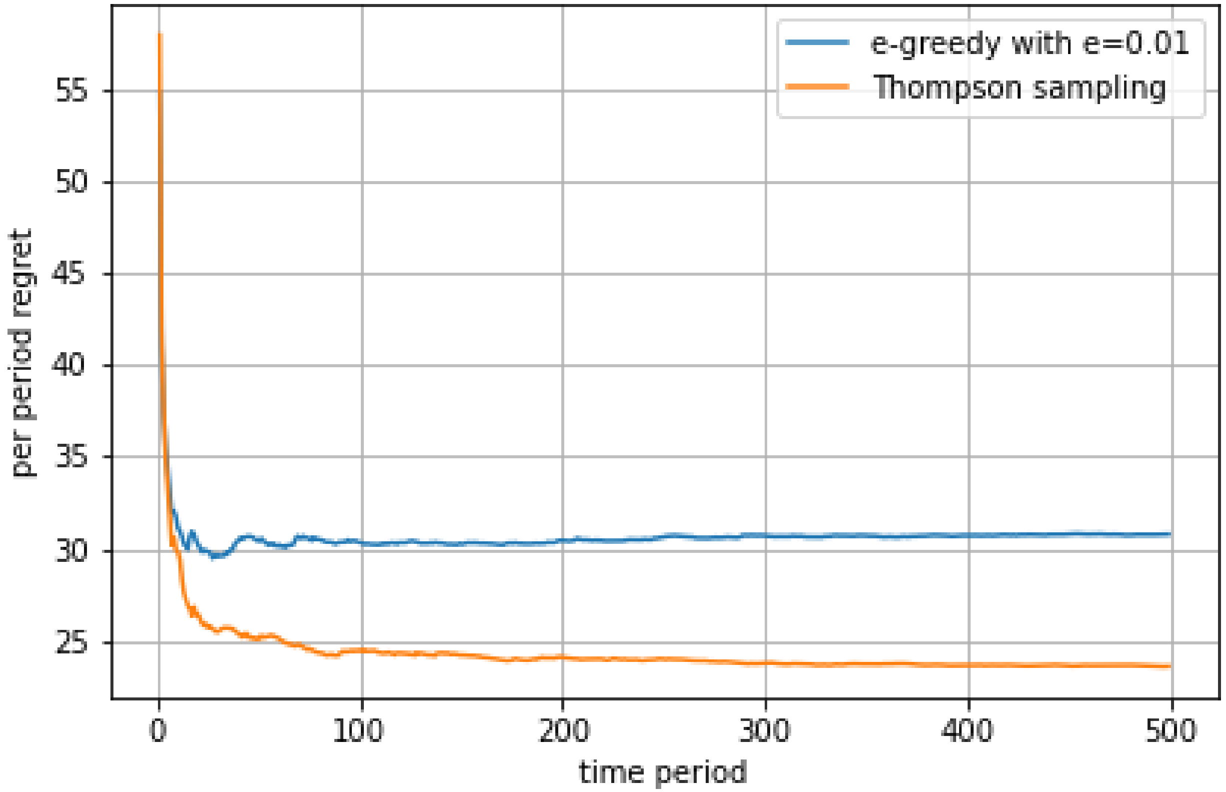
Use mean value of prior distribution, rather than sampling

ϵ-greedy

CORE
Control + Optimization Research Lab

Multi-Armed Bandit Problem

Using Thompson sampling reduce regret much larger than greedy policy



Application 2. Model-based Reinforcement Learning via Thompson Sampling

Model-based RL via Thompson Sampling : Implementation

In this application, we assume

- MDP has n states & m actions
- transition probability P : **unknown**
- reward matrix R : known

Recall : transition probability : matrix of the form

$$P = \begin{pmatrix} p(s_1|s_1, a_1) & \cdots & p(s_n|s_1, a_1) \\ \vdots & \vdots & \vdots \\ p(s_1|s_n, a_m) & \cdots & p(s_n|s_n, a_m) \end{pmatrix}$$

$\Longrightarrow$ each row : probability vector of transitions from a single (s, a) pair

Model-based RL via Thompson Sampling : Implementation

Our strategy is to repeat the followings:

1. Sample P from the prior/posterior distribution (make a guess)
   - Sample each row $(p_1, \cdots, p_n)$ from **Dirichlet distribution**
     - Why? $\Longrightarrow$ Dirichlet : conjugate prior of Categorical
2. Act optimally according to P
   - Use **relative value iteration** (for average reward infinite-horizon MDP)
3. Update distribution using **Bayes rule**
   - Update parameters of each Dirichlet distributions ($n \times m$ distributions in total)

CORE

Control + Optimization Research Lab

Model-based RL via Thompson Sampling : Implementation

```python
class TSRLAgent:
    def __init__(self, num_states, num_actions, alpha0, R):
        self.num_states = num_states
        self.num_actions = num_actions
        self.alpha = alpha0
        self.R = R
        self.policy = None

    def reset(self):
        P = self.infer_model()
        self.policy = value_iteration(P=P, R=self.R)[1]

    def act(self, state):
        return self.policy[state]

    def update(self, state, action, next_state):
        self.alpha[self.num_states * action + state, next_state] += 1

    def infer_model(self):
        nrows = self.num_states * self.num_actions
        P = np.zeros((nrows, self.num_states))
        for i in range(nrows):
            P[i] = np.random.dirichlet(self.alpha[i])
        return P
```

prior distribution

Sample new model & compute optimal policy via DP

Posterior update : if (s^t, a^t, s^{t+1}) is observed, then adjust $\alpha(s^t, a^t)$.

Sample new model!

$p^{(t)}(\cdot|s,a) \sim \textbf{Dirichlet}\left(\alpha^{(t)}(s,a)\right)$ for each (s,a)

CORE

Control + Optimization Research Lab

Model-based RL via Thompson Sampling : Implementation

```python
def TSRL(env):
    alpha0 = .1 * np.ones((num_states * num_actions, num_states))
    agent = TSRLAgent(num_states=num_states, num_actions=num_actions, alpha0=alpha0, R=env.R_true)
    t = 0
    t_init = 0
    ep_len = 1
    visit_count = np.zeros((num_states, num_actions), dtype=int)
    state = env.reset()
    while True:
        agent.reset()
        visit_count_init = visit_count
        while t <= t_init + ep_len:
            action = agent.act(state)
            next_state, reward = env.step(action)
            visit_count[state, action] += 1
            agent.update(state, action, next_state)
            state = next_state
            if visit_count[state, action] > 2 * visit_count_init[state, action]:
                break
            t += 1
        ep_len = t - t_init
        t_init = t
```

Forget about t_init or ep_len...

Sample new model & compute optimal policy via DP

Act as if the sampled model is correct

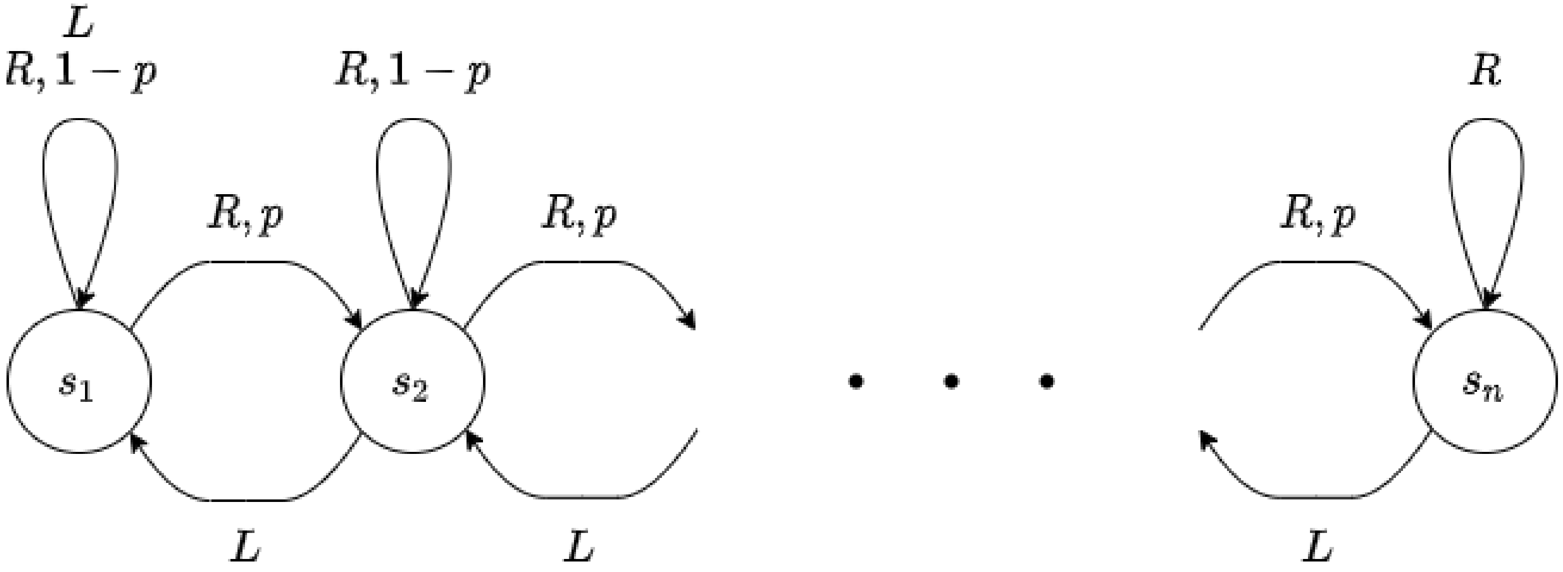
Bayesian update

resampling criteria

Model-based RL via Thompson Sampling : Implementation

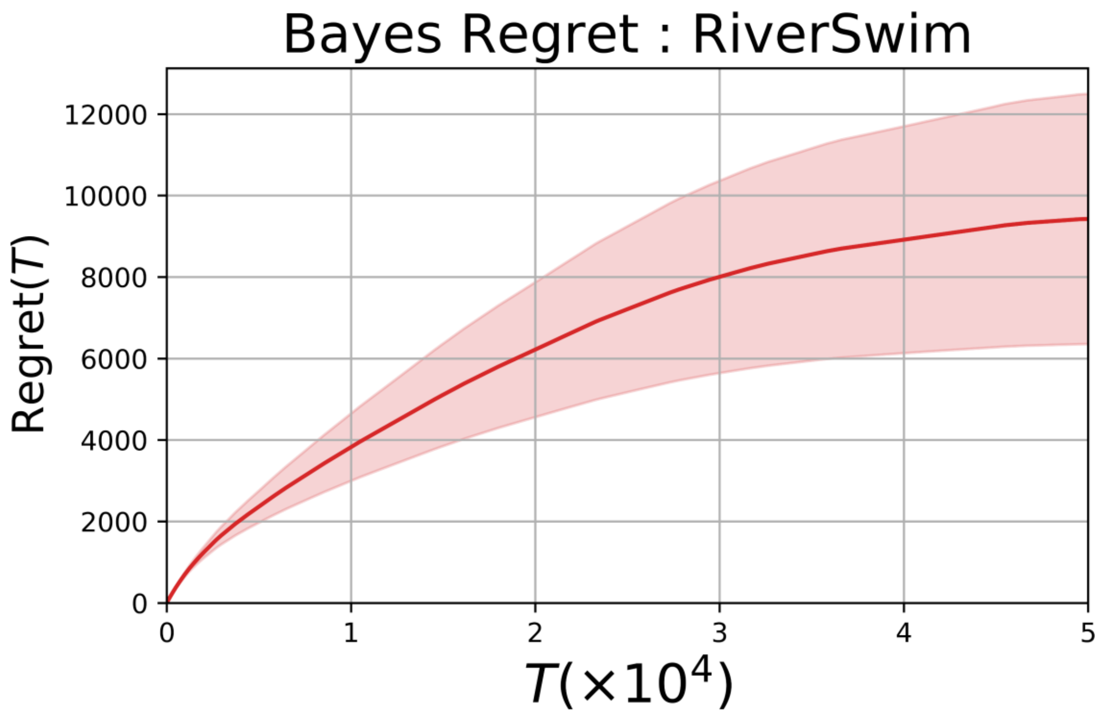
RiverSwim Example

- reward : -0.8 if LEFT at s_1, 0 if RIGHT at s_n, -1 everywhere else
- LEFT always succeeds, but RIGHT succeeds with probability p



Model-based RL via Thompson Sampling : Implementation

10 states, $p = 0.5$



$$\text{Regret}(T) = O(\sqrt{T})$$

Application 3. Learning Linear Quadratic Control via Thompson Sampling

Learning Linear Quadratic Control via Thompson Sampling

Consider the following linear system with stochastic noise:

$$x_{t+1} = Ax_t + Bu_t + w_t, \quad w_t \sim \mathcal{N}(0, I).$$

with a quadratic cost function

$$c(x_t, u_t) = x_t^\top Q x_t + u_t^\top R u_t.$$

Here we assume

- system matrices A & B : **Unknown**
- cost function $c(x, u)$: known

Why linear system? $\implies$ Many interesting systems can be approximated as a linear one!

Learning Linear Quadratic Control via Thompson Sampling

The approach is similar to the previous one: we simply repeat

1. Sample $\Theta = \begin{bmatrix} A \\ B \end{bmatrix} \in \mathbb{R}^{(n+m)\times n}$ from the prior/posterior distribution (make a guess)
   - Sample each column from **Gaussian distribution**
2. Act optimally according to Θ
   - apply theory of **Algebraic Riccati equation**
3. Update distribution using **Bayes rule**
   - Update parameters of each Gaussian distributions (n distributions in total)

Learning Linear Quadratic Control via Thompson Sampling

```python
class Distribution:
    def __init__(self, system: LinearSystem, Q, R, mean_prior, cov_prior):
        ...

    def sample(self, delta):
        L = np.linalg.cholesky(self.cov)
        d, n = self.mean.shape
        accept = False
        while not accept:
            theta = self.mean + L @ np.random.randn(d, n)
            G = compute_gain(theta, self.Q, self.R)
            accept = self.system.is_stable(G, delta)
        return theta

    def update(self, z, x_next):
        cov_times_z = self.cov @ z
        denom = 1 + np.dot(z, cov_times_z)
        self.mean += np.outer(cov_times_z, x_next - z @ self.mean) / denom
        self.cov -= np.outer(cov_times_z, cov_times_z) / denom
        return
```

sample $\Theta = \begin{bmatrix} A \\ B \end{bmatrix}$ from Gaussian prior/posterior

Given $(\underbrace{x_t, u_t}_{z_t}, x_{t+1})$, update the posterior

CORE
Control + Optimization Research Lab

Learning Linear Quadratic Control via Thompson Sampling

```
def PSRL_LQ(system: LinearSystem, Q, R, mean_prior, cov_prior, delta=0.999):
    distribution = Distribution(system, Q, R, mean_prior, cov_prior)
    t = 0
    t_init = 0
    ep_len = 1
    x = system.reset()
    while True:
        theta = distribution.sample(delta=delta)
        G = compute_gain(theta, Q, R)
        init_cov_sz = distribution.cov_sz
        current_cov_sz = init_cov_sz
        while (t <= t_init + ep_len) and (current_cov_sz >= 0.5 * init_cov_sz):
            u = G @ x
            state_arr.append(x)
            cost += np.dot(x, Q @ x) + np.dot(u, R @ u)
            cost_arr.append(cost)
            z = np.concatenate([x, u])
            x_next = system.step(u)
            distribution.update(z=z, x_next=x_next)
            x = x_next
            t += 1
        ep_len = t - t_init
        t_init = t
```

sample $\Theta = \begin{bmatrix} A \\ B \end{bmatrix}$ from Gaussian prior/posterior

obtain optimal policy $u_t = G(\Theta)x_t$ by solving Algebraic Ricatti eqn.

resampling criteria

Given $(\underbrace{x_t, u_t}_{z_t}, x_{t+1})$, update the posterior

Learning Linear Quadratic Control via Thompson Sampling

Path tracking example

Goal : control the car to follow the laser point of constant velocity

linearized dynamics with sampling time $h = 0.1\text{sec}$:

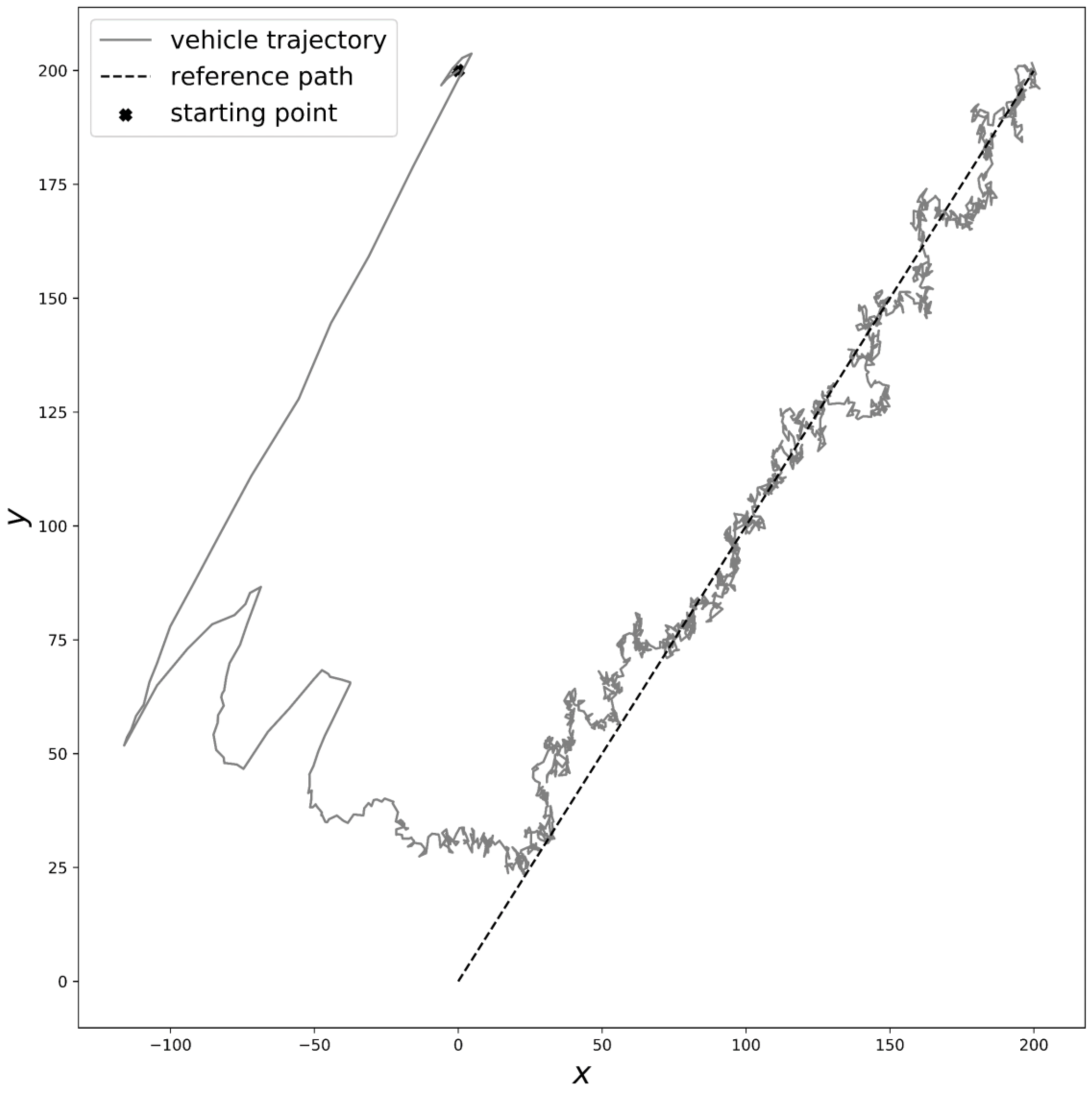
$$A = \begin{pmatrix} 1 & 0 & -h^2 \\ 0 & 1 & h^2 \\ 0 & 0 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} h/\sqrt{2} & 0 \\ h/\sqrt{2} & 0 \\ 0 & 1 \end{pmatrix}$$

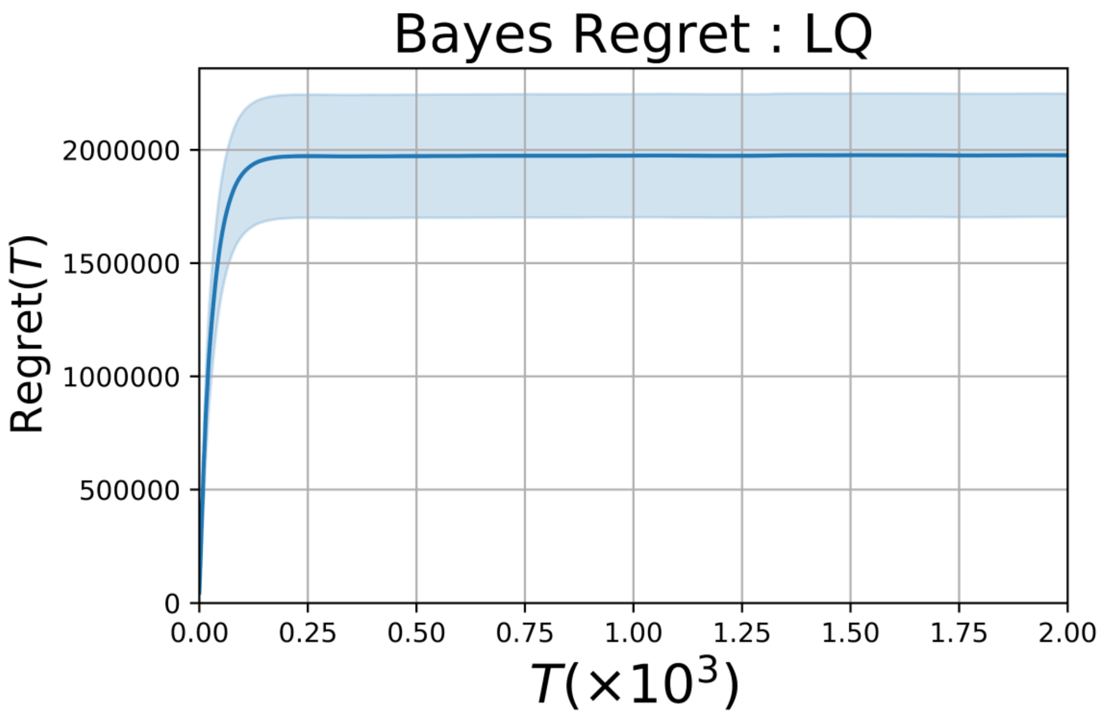
with state vector $x_t = x_{\text{car}} - x_{\text{laser}}$ where $x_{\text{car}} = (x_{\text{car},t}, y_{\text{car},t}, \theta_{\text{car},t})$

CORE

Control + Optimization Research Lab

Learning Linear Quadratic Control via Thompson Sampling





CORE
Control + Optimization Research Lab