AA228: Policy Search

So far, we've seen how exact solution methods can be used to solve for a policy in an offline method. We've also seen how online planning can handle large state spaces by reasoning over actions from an initial state. In this notebook, we'll discuss **policy search**, which involves searching over the space of policy parameters rather than actions. This idea will carry forward through the next few lectures.

You can find this notebook in the github repo here

Table of Contents

Motivation

Q: Given the solution methods we've seen so far, how can we solve an MDP with continuous states or actions?

Policy search helps us scale to very large or continuous state spaces.

```julia
begin
    using POMDPs
    using POMDPTools
    using Distributions
    using Parameters
    using Random
end
```

MDP Formulation: Inverted Pendulum

Let's define a running example with continuous states and actions to get started.

The inverted pendulum problem involves stabilizing a pendulum in an inverted position. Suppose we have a motor mounted at the pendulum's pivot point. The objective is to control the angle of the pendulum so that it stays as close to vertical as possible despite any disturbances that may occur, such as gusts of wind. This problem is challenging because the inverted pendulum is inherently unstable, making it difficult to maintain its balance.

Let's define the angle of the pendulum from vertical as ϕ.

We'll assume the pendulum has some fixed length l and mass m.

Q: What should we use as the MDP state? Actions?

Q: What is the transition model? (The actual equations are not important, but describe in words.)

Q: What components might our reward function have?

Great! Now let's look at how we can set define this MDP in code. We'll use the POMDPs.jl environment.

First, we'll define a struct type. This is a container for useful data about the pendulum.

PendulumMDP

```julia
@with_kw struct PendulumMDP <: MDP{Array{Float64}, Array{Float64}}
    Rstep = 1 # Reward earned on each step of the simulation
    λcost = 1 # Coefficient to the traditional OpenAIGym Reward
    max_speed::Float64 = 8.
    max_torque::Float64 = 100.
    dt::Float64 = .05
    g::Float64 = 10.
    m::Float64 = 1.
    l::Float64 = 1.
    γ::Float64 = 0.99
end
```

```julia
POMDPs.discount(mdp::PendulumMDP) = mdp.γ
```

For many problems, explicitly writing the transition model $T(s' \mid s, a)$ and reward function $R(s, a)$ can be difficult. Here, we define a **generative model** of the dynamics and reward.

```julia
function pendulum_dynamics(env, s, a; rng::AbstractRNG = Random.GLOBAL_RNG)
    θ, ω = s[1], s[2]
    dt, g, m, l = env.dt, env.g, env.m, env.l

    a = a[1]
    a = clamp(a, -env.max_torque, env.max_torque)
    costs = angle_normalize(θ)^2 + 0.1f0 * ω^2 + 0.001f0 * a^2

    ω = ω + (-3. * g / (2 * l) * sin(θ + π) + 3. * a / (m * l^2)) * dt
    θ = angle_normalize(θ + ω * dt)
    ω = clamp(ω, -env.max_speed, env.max_speed)

    sp = [θ, ω]
    r = env.Rstep - env.λcost*costs
    return sp, r
end;
```

```julia
angle_normalize(x) = mod((x+π), (2*π)) - π;
```

```julia
function POMDPs.gen(mdp::PendulumMDP, s, a, rng::AbstractRNG = Random.GLOBAL_RNG)
    sp, r = pendulum_dynamics(mdp, s, a, rng=rng)
    (sp = sp, r = r)
end
```

Define an initial state distribution

```julia
function POMDPs.initialstate(mdp::PendulumMDP)
    θ0 = Distributions.Uniform(-π/6., π/6.)
    ω0 = Distributions.Uniform(-0.1, 0.1)
    ImplicitDistribution((rng) -> [rand(rng, θ0), rand(rng, ω0)])
end
```

Rendering functions are below (hidden cells). Feel free to look under the hood if you are curious!

Policy Parameterization

Now that we've established the components of our MDP, let's start to think about how to solve it. We already discussed how offline methods and online tree-search methods might be difficult to apply.

We introduce the notion of a **parameterized policy**. We can denote the action of policy π at state s parameterized by θ as

$$a = \pi_\theta(s)$$

for deterministic policies, and

$$a \sim \pi_\theta(a \mid s)$$

for stochastic policies.

Policy space is often lower-dimensional than state space, and can often be searched more easily.

The parameters θ may be a vector or some other more complex representation. For example, we may want to represent our policy using a neural network with a particular structure. We would use θ to represent the weights in the network.

Q: How could we parameterize a policy for the inverted pendulum problem? Assume our state vector is $s = [\phi, \dot{\phi}]$

Policy Evaluation

The expected discounted return of a policy π from initial state distribution $b(s)$ is

$$U(\pi) = \sum_s b(s) U^\pi(s)$$

When we have a large or continuous state space, we often cannot compute the utility of following a policy $U(\pi)$ exactly. Instead, we rewrite $U(\pi)$ in terms of trajectories of states, actions, and rewards under the policy π.

The key idea is that we want to estimate utility based on *simulated trajectories*, τ. We call the sum of rewards for trajectory τ, $R(\tau)$ the *trajectory return*. Now, we can write the utility of following policy π as:

$$U(\pi) = \mathbb{E}[R(\tau)]$$

The expected value (or mean) return can be *approximated* by taking the mean total reward of many trajectories:

$$U(\pi) \approx \frac{1}{m} \sum_{i=1}^{m} R(\tau^{(i)})$$

This is sometimes called Monte Carlo policy evaluation.

The POMDPs.jl package provides us with a convenient way to get the sum of discounted returns from a simulation (or a 'rollout'). The `RolloutSimulator` type simulates a given policy for a fixed number of steps and returns the sum of discounted returns. Once we create a simulator like: `sim = RolloutSimulator(max_steps=max_steps)`

We can compute get the total discounted reward using `R=simulate(sim, mdp, policy)`

Let's write a function to perform Monte Carlo policy evaluation.

```julia
function mc_policy_evaluation(mdp::MDP, π::Policy; m=100, max_steps=100)
    sim = RolloutSimulator(max_steps=max_steps)
    return mean([simulate(sim, mdp, π) for _=1:m])
end;
```

Policy Search Overview

In policy search, our goal is to optimize a policy's utility with respect to its parameters. In other words, we search over the parameter space for a set of parameters that maximize our utility.

Here, we'll try out policy search for a simple 2D policy parameterization:

$$\pi_\theta(s) = \theta_1 s_1 + \theta_2 s_2 = \theta^T s$$

First, let's create our MDP

```
mdp = PendulumMDP
        Rstep: Int64 1
        λcost: Int64 1
        max_speed: Float64 8.0
        max_torque: Float64 100.0
        dt: Float64 0.05
        g: Float64 10.0
        m: Float64 1.0
        l: Float64 1.0
        γ: Float64 0.99
```

```
mdp = PendulumMDP()
```

Select θ_1 and θ_2 to maximize the utlity

theta1 : ● theta2 : ●

```
θ =  [-30.0, -30.0]
```

```
θ = [θ1, θ2]
```

```
policy = FunctionPolicy((s) -> [θ' * s]);
```

```
mean_utility = -1165.9489226242765
```

```
mean_utility = mc_policy_evaluation(mdp, policy)
```

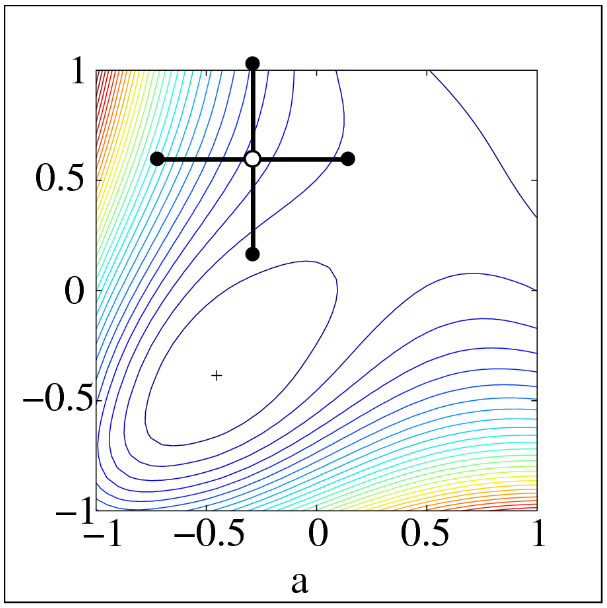
```
# begin
#   animate_pendulum(mdp, policy, "toyexample.gif")
#   LocalResource("toyexample.gif")
# end
```

Local Search (Hooke-Jeeves)

Hooke-Jeeves is an example of a local search method. We saw a local search algorithm in structure learning. In local search, we evaluate the neighbors of a current design point (θ), and move to the neighbor that most improves the value.

The Hooke-Jeeves algorithm is one example of local search. The algorithm takes a step of size ±α in each of the *coordinate directions* from the current θ. If an improvement is found, it moves to the best point. If no improvement is found, the algorithm shrinks the step size α and repeats. The algorithm terminates once α reaches some minimum value.

The algorithm is illustrated below



Now let's code up Hooke-Jeeves and apply it to the inverted pendulum! This code very closely follows the textbook. First, we'll define a special struct to store useful attributes of the algorithm.

```julia
struct HookeJeevesPolicySearch
    θ # initial parameterization
    α # step size
    c # step size reduction factor
    ϵ # termination step size
end
```

Next, let's define the optimization procedure. We'll take in the algorithm attributes, as well as a function that takes in a parameter vector θ and returns a Monte Carlo estimate of the expected utility.

```julia
function optimize(M::HookeJeevesPolicySearch, U)
    θ, θ′, α, c, ϵ = copy(M.θ), similar(M.θ), M.α, M.c, M.ϵ
    u = U(θ)
    n = length(θ)
    history = [copy(θ)]
    while α > ϵ
        copyto!(θ′, θ)
        best = (i=0, sgn=0, u=u)
        for i in 1:n
            for sgn in (-1,1)
                θ′[i] = θ[i] + sgn*α
                u′ = U(θ′)
                if u′ > best.u
                    best = (i=i, sgn=sgn, u=u′)
                end
            end
            θ′[i] = θ[i]
        end
        if best.i != 0
            θ[best.i] += best.sgn*α
            u = best.u
        else
            α *= c
        end
        push!(history, copy(θ))
    end
    return θ, history
end;
```

Now we'll define a function our function $U(\theta)$

```julia
U(θ) = mc_policy_evaluation(mdp, FunctionPolicy((s)->[θ'*s]), max_steps=500, m=10);
```

Specify the attributes of our algorithm

```julia
hookejeeves = HookeJeevesPolicySearch(rand(2), 1.0, 0.5, 1e-1);
```

Now run the optimization!

```julia
θhj, history = optimize(hookejeeves, U);
```

```
[-6.92142, -1.44199]
```

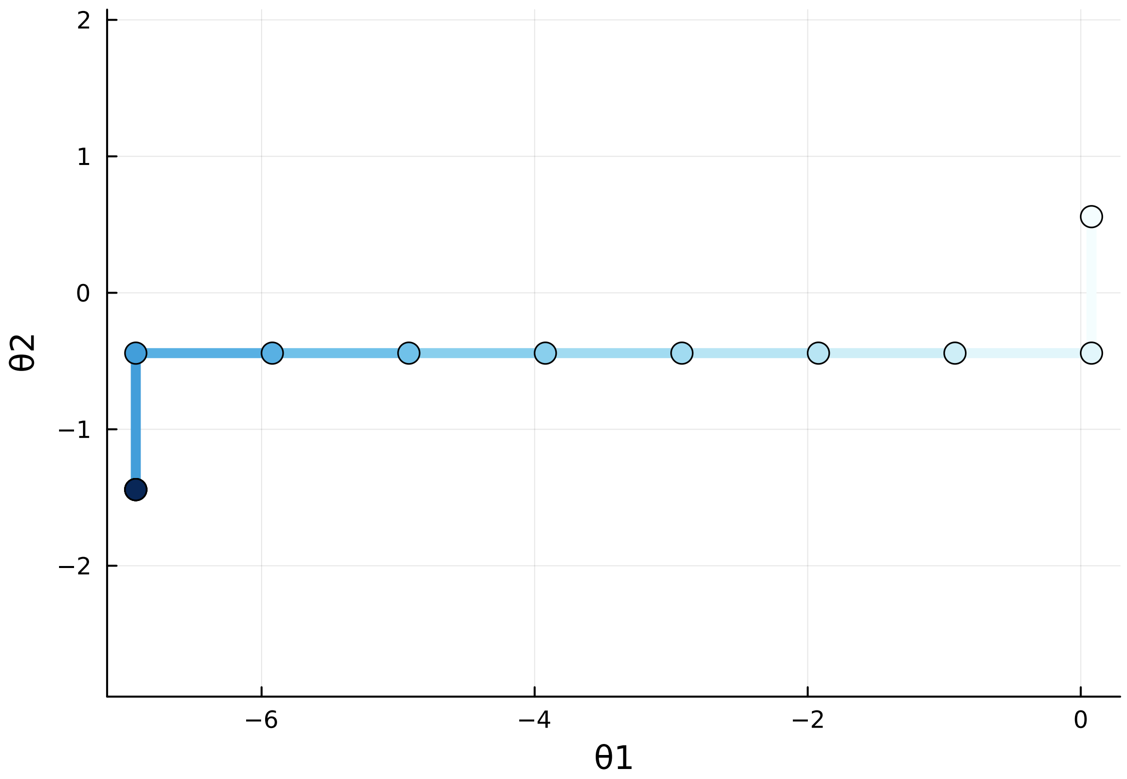
```julia
θhj
```

```
98.60627977101966
```

```julia
U(θhj)
```

```
# begin
#   πcem = FunctionPolicy((s)->[θhj'*s])
#   animate_pendulum(mdp, πcem, "hj.gif")
#   LocalResource("hj.gif")
# end
```

We can also examine the history of points evaluated by the algorithm. The algorithm starts at light blue and progresses toward darker blue.



Q: Suppose we are performing Hooke-Jeeves for policy parameterized by a 3-element vector. We are currently evaluation the point $[2, -5, 10]$. Assume the step size is 1.

What points will Hooke-Jeeves evaluate next?

Q:Hooke-Jeeves evaluates each of these points, and finds they have utilities

$$(10, 23, 16, 5, 34, 27)$$

. The current policy has a value of 35. What will Hooke-Jeeves do?

Genetic Algorithms

Local search algorithms can easily become stuck in local minima. Population-based algorithms maintain a set of points in the parameter space. By maintaining and changing the set of parameters, population-based algorithms can be less susceptible to becoming stuck. However, they are not guaranteed to converge to the global optimum.

Genetic algorithms are population-based algorithms that are inspired by biological evolution. The algorithm starts with a population of points m in parameter space, called 'individuals': $\theta^{(1)}, \ldots, \theta^{(m)}$. We compute $U(\theta)$ for each of point. The top-performing samples, are called *elite samples*. At the next iteration, some number m_{elite} of the elite samples are chosen. New points in the population are created by adding gaussian noise to the elite individuals.

```julia
struct GeneticPolicySearch
    θs # initial population
    σ # initial standard deviation
    m_elite # number of elite samples
    k_max # number of iterations
end
```

```julia
function optimize(M::GeneticPolicySearch, U)
    θs, σ = M.θs, M.σ
    n, m = length(first(θs)), length(θs)
    history = []
    for k in 1:M.k_max
        us = [U(θ) for θ in θs]
        sp = sortperm(us, rev=true)
        θ_best = θs[sp[1]]
        push!(history, (copy(θs), copy(θs[sp[1:M.m_elite]])))
        rand_elite() = θs[sp[rand(1:M.m_elite)]]
        θs = [rand_elite() + σ.*randn(n) for i in 1:(m-1)]
        push!(θs, θ_best)
    end
    return last(θs), history
end;
```

Creating an initial population

```
[[-13.7292, -14.7984], [-5.27787, 8.34339], [-13.3499, -18.6677], [8.10831, 20.2755], [24.
```

```julia
begin
    npop = 50
    θ0 = [50 .* rand(2).-25 for _=1:npop]
end
```

```julia
ga = GeneticPolicySearch(θ0, 1.0, 10, 10);
```

```julia
θga, hga = optimize(ga, U);
```

The final parameters found using the genetic algorithm are:

```
[-18.4331, -8.78312]
```

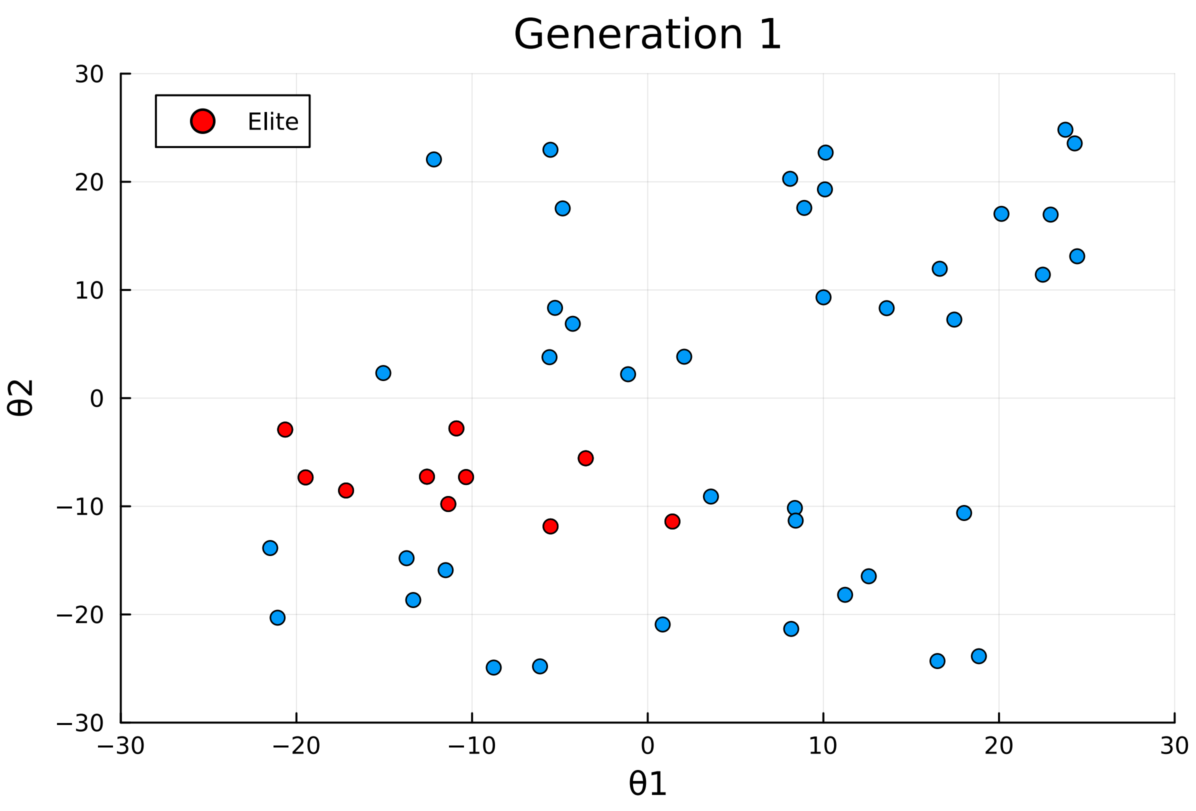
```julia
θga
```

With an expected utility of:

```
98.63754766061155
```

```julia
U(θga)
```

Let's take a look a the population over each iteration. Try playing with the initial population, standard deviation, and other parameters and see how the algorithm behaves.



Q: How will the solution found by a genetic algorithm depend on the initial population? How would it depend on the magnitude of noise added to elite samples?

Q: Is a genetic algorithm guaranteed to converge to the optimal policy?

Cross Entropy Method

The cross entropy method involves updating a search distribution over the parameters. The distribution over parameters $p(\theta \mid \psi)$ has its own parameters ψ. Typically, we use a Gaussian distribution, where ψ represents the mean and covariance matrix.

The algorithm iteratively updates the parameters ψ. At each iteration, we draw m samples from the associated distribution and then update ψ to fit a set of elite samples. We stop after a fixed number of iterations, or when the search distribution becomes very focused.

```julia
struct CrossEntropyPolicySearch
    p # initial distribution
    m # number of samples
    m_elite # number of elite samples
    k_max # number of iterations
end
```

```julia
function optimize_dist(M::CrossEntropyPolicySearch, U)
    p, m, m_elite, k_max = M.p, M.m, M.m_elite, M.k_max
    history = []
    for k in 1:k_max
        θs = rand(p, m)
        us = [U(θs[:,i]) for i in 1:m]
        θ_elite = θs[:,sortperm(us)[(m-m_elite+1):m]]
        push!(history, (p, copy(θs), copy(θ_elite)))
        p = Distributions.fit(typeof(p), θ_elite)

    end
    return p, history
end;
```

```julia
function optimize(M, U)
    d, history = optimize_dist(M, U)
    return Distributions.mode(d), history
end;
```

A key step of using the algorithm is selecting the *initial distribution*. The distribution should cover the parameter space of interest.

```
p0 = DiagNormal(
     dim: 2
     μ: [0.0, 0.0]
     Σ: [25.0 0.0; 0.0 25.0]
     )
```

```julia
p0 = MvNormal(zeros(2), [5, 5])
```

```
cem =   CrossEntropyPolicySearch(DiagNormal(            , 50, 10, 10)
                                 dim: 2
```

```julia
cem = CrossEntropyPolicySearch(p0, 50, 10, 10)
```

```julia
θcem, hcem = optimize(cem, U);
```

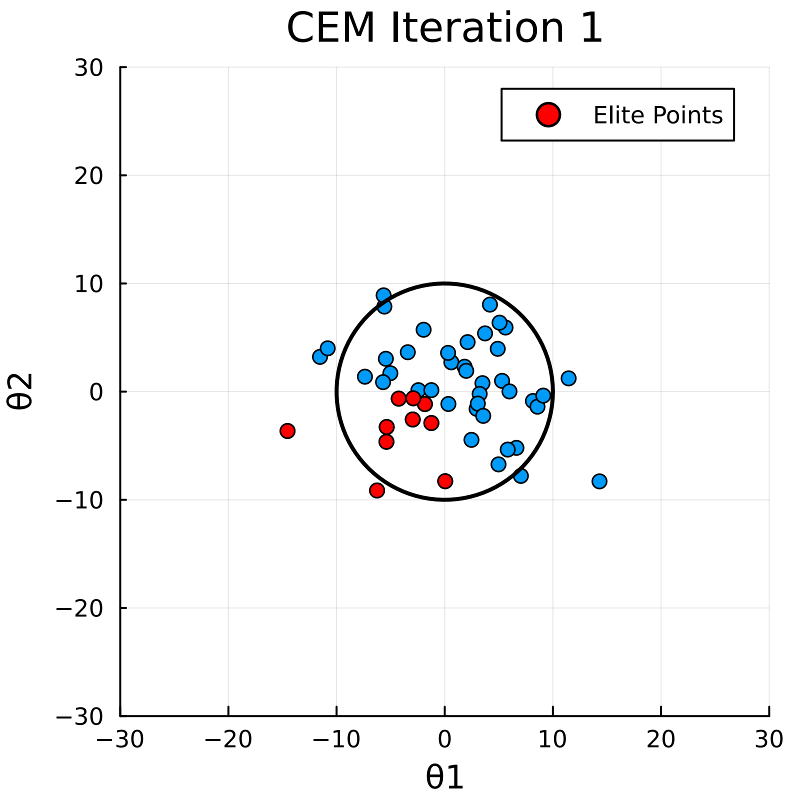
```
[-11.9783, -2.31978]
```

```julia
θcem
```

```
98.4369082219945
```

```julia
U(θcem)
```

We can also examine how the proposal distribution changes over iterations.



Q: We are performing the Cross Entropy Method on a 1D parameter space. We have elite samples at $(1, 4, 2.5, 10)$ What will be the updated parameters for our Gaussian search distribution?

Recall the maximum likelihood estimate for Gaussian's parameters with samples $o_1, \dots, o_m$:

$$\hat{\mu} = \frac{\sum_i o_i}{m}$$

$$\hat{\sigma}^2 = \frac{\sum_i (o_i - \hat{\mu})^2}{m}$$

Algorithm Comparison

Let's compare the performance of each algorithm on the inverted pendulum

```julia
begin
    @show U(θhj)
    @show U(θga)
    @show U(θcem)
end;
```

```
U(θhj) = 98.47174964444443
U(θga) = 98.76814667770871
U(θcem) = 98.61941941316118
```

They all do pretty well!

Q:What if our policy had many more parameters, say 100. Which algorithm would you pick?

Q: Suppose that we want to perform policy optimization on a problem where we know that policies far apart in parameter space can have similar high utility. What are the advantages of genetic algorithms over hooke-jeeves? What about the Cross Entropy method?

Next Steps: Gradient Information

Thus far, all of the algorithms for policy search have not used gradient information of the expected utility with respect to policy parameters. It turns out that optimizing policies with many parameters can be done much more efficiently with gradient information

How can we compute the gradient $\nabla U(\theta)$?

One option is to use *finite differences*. The idea of finite differences comes from the linear approximation of the gradient. We can estimate the gradient (or the slope) of a function in 1D by checking how much the value of $f(x)$ changes for some small change in x.

$$\frac{df}{dx} \approx \frac{f(x+\delta) - f(x)}{\delta}$$

If we extend this same idea to our utility function,

$$\nabla U(\theta) \approx \left[\frac{U(\theta + \delta \mathbf{e}^1) - U(\theta)}{\delta}, \ldots, \frac{U(\theta + \delta \mathbf{e}^n) - U(\theta)}{\delta}\right]$$

Where $\mathbf{e}^i$ is the standard basis, which is zero everywhere except the ith component.

Luckily, there are some great packages in most programming languages that do this for us!

However, recall that our Monte Carlo estimate of the expected utility is stochastic, or noisy. This means that gradients of the utility function will have noise too! If the gradients are too noisy, they will provide very poor guidance for our policy search.

A key challenge in policy gradient estimation is dealing with noisy policy gradients.

```julia
using FiniteDiff
```

```julia
function deterministic_policy_evaluation(mdp::MDP, π::Policy; m=100, max_steps=100)
    sim = RolloutSimulator(rng=MersenneTwister(42), max_steps=max_steps)
    return mean([simulate(sim, mdp, π) for _=1:m])
end;
```

```julia
Ufd(θ) = deterministic_policy_evaluation(mdp, FunctionPolicy((s)->[θ'*s]),
max_steps=100, m=5);
```

Let's try taking the policy gradient.

```
[362.1, -1606.73]
```

```julia
FiniteDiff.finite_difference_gradient(Ufd, [0.1, 0.1])
```

Now that we have an estimate of the gradient, how can we use it to improve the policy?

One of the simplest approaches is **gradient ascent**. Gradient ascent takes steps in parameter space along the gradient direction. The step size α determines how far along the gradient direction the update moves. The update for θ is

$$\theta \leftarrow \theta + \alpha \nabla U(\theta)$$

Determining the step size is a major challenge. Large steps can lead to faster progress to the optimum, but they can overshoot.

Let's try running a very simple version of gradient descent and see how it performs!

```julia
begin
    θi = rand(2)
    α = 1e-1
    niter = 500
    for k=1:niter
        gradU = FiniteDiff.finite_difference_gradient(Ufd, θi)
        θi += α .* gradU
    end
end
```

The final parameters are a little different than what was previously found. Why could that be?

```
[-14.1966, -3.95404]
```

```julia
θi
```

```
62.28524243579915
```

```julia
Ufd(θi)
```

```julia
# begin
#   πsgd = FunctionPolicy((s)->[θi'*s])
#   animate_pendulum(mdp, πsgd, "sgd.gif")
#   LocalResource("sgd.gif")
# end
```

Gradient descent is able to find a decent policy pretty quickly! However, it isn't as good as the policies we've found previously. It turns out there are **much** better ways to estimate the gradient of a policy, and much more intelligent variations on gradient ascent. We'll explore these in the coming lecture.

This wraps up our discussion on policy search. I hope it was helpful!