> *Exercise 2.1* In ϵ-greedy action selection, for the case of two actions and $\epsilon = 0.5$, what is the probability that the greedy action is selected?

The probability that the greedy action is selected is a combination of $P(u > \epsilon)$ where u is a sample from $U = uniform(0, 1)$ and the probability of selecting the greedy action at random. Since both cases are independent, the probabilities can be summed.

$$P(a = a_{greedy}) = P_1 + P_2$$

$$P_1 = P(u > \epsilon) = (1 - \epsilon) = 0.5$$

$$P_2 = P(a_{rand} = a_{greedy})P(u < \epsilon) = \frac{\text{num greedy actions}}{\text{num total actions}} \times \epsilon = 0.5 \times 0.5 = 0.25$$

$$P(a = a_{greedy}) = 0.5 + 0.25 = 0.75$$

2.3 The 10-armed Testbed

The following code recreates the 10-armed Testbed from section 2.3

```
PlotlyBackend()
```

```
begin
    using Random ✓
    using Base.Threads
    using Plots ✓
    using BenchmarkTools ✓
    using PlutoUI ✓
    using Profile ✓
    using JLD2 ✓
    plotly()
end
```

```
create_bandit (generic function with 1 method)
```

```
function create_bandit(k::Integer; offset::T = 0.0) where T<:AbstractFloat
    qs = randn(T, k) .+ offset #generate mean rewards for each arm of the bandit
end
```

```
sample_bandit (generic function with 1 method)
```

```
function sample_bandit(a::Integer, qs::Vector{T}) where T<:AbstractFloat
    randn(T) + qs[a] #generate a reward with mean q[a] and variance 1
end
```

```
simple_algorithm (generic function with 1 method)
```

```
function simple_algorithm(qs::Vector{Float64}, k::Integer, ϵ::AbstractFloat; steps = 1000, Qinit = 0.0, α = 0.0, c = 0.0)
    bandit(a) = sample_bandit(a, qs)
    N = zeros(k)
    Q = ones(k) .* Qinit
    accum_reward_ideal = 0.0
    accum_reward = 0.0
    cum_reward_ideal = zeros(steps)
    step_reward_ideal = zeros(steps)
    cum_reward = zeros(steps)
    step_reward = zeros(steps)
    bestaction = argmax(qs)
    optimalstep = fill(false, steps)
    optimalcount = 0
    optimalaction_pct = zeros(steps)
    actions = collect(1:k)
    for i = 1:steps
        shuffle!(actions) #so that ties are broken randomly with argmax
        a = if rand() < ϵ
            rand(actions)
        elseif c == 0.0
            actions[argmax(view(Q, actions))]
        else
            actions[argmax(view(Q, actions) .+ (c .* sqrt.(log(i) ./ view(N, actions))))]
        end
        if a == bestaction
            optimalstep[i] = true
            optimalcount += 1
        end
        step_reward[i] = bandit(a)
        step_reward_ideal[i] = bandit(bestaction)
        accum_reward_ideal += step_reward_ideal[i]
        cum_reward_ideal[i] = accum_reward_ideal
        accum_reward += step_reward[i]
        cum_reward[i] = accum_reward
        optimalaction_pct[i] = optimalcount / i
        N[a] += 1.0
        if α == 0.0
            Q[a] += (1.0/N[a])*(step_reward[i] - Q[a])
        else
            Q[a] += α*(step_reward[i] - Q[a])
        end
    end
    return (;Q, step_reward, step_reward_ideal, cum_reward, cum_reward_ideal, optimalstep, optimalaction_pct)
end
```

```
average_simple_runs (generic function with 1 method)
```

```
k = 10
```

```
k = 10
```

```
(Q = [-0.6844, 2.22401, -0.556754, 1.52197, 0.252321, -0.0467577, 0.342918, 1.50853, -1.35
```

```
simple_algorithm(create_bandit(k), k, 0.1)
```

```
ϵ_list = [0.0, 0.01, 0.1]
```

```
ϵ_list = [0.0, 0.01, 0.1]
```

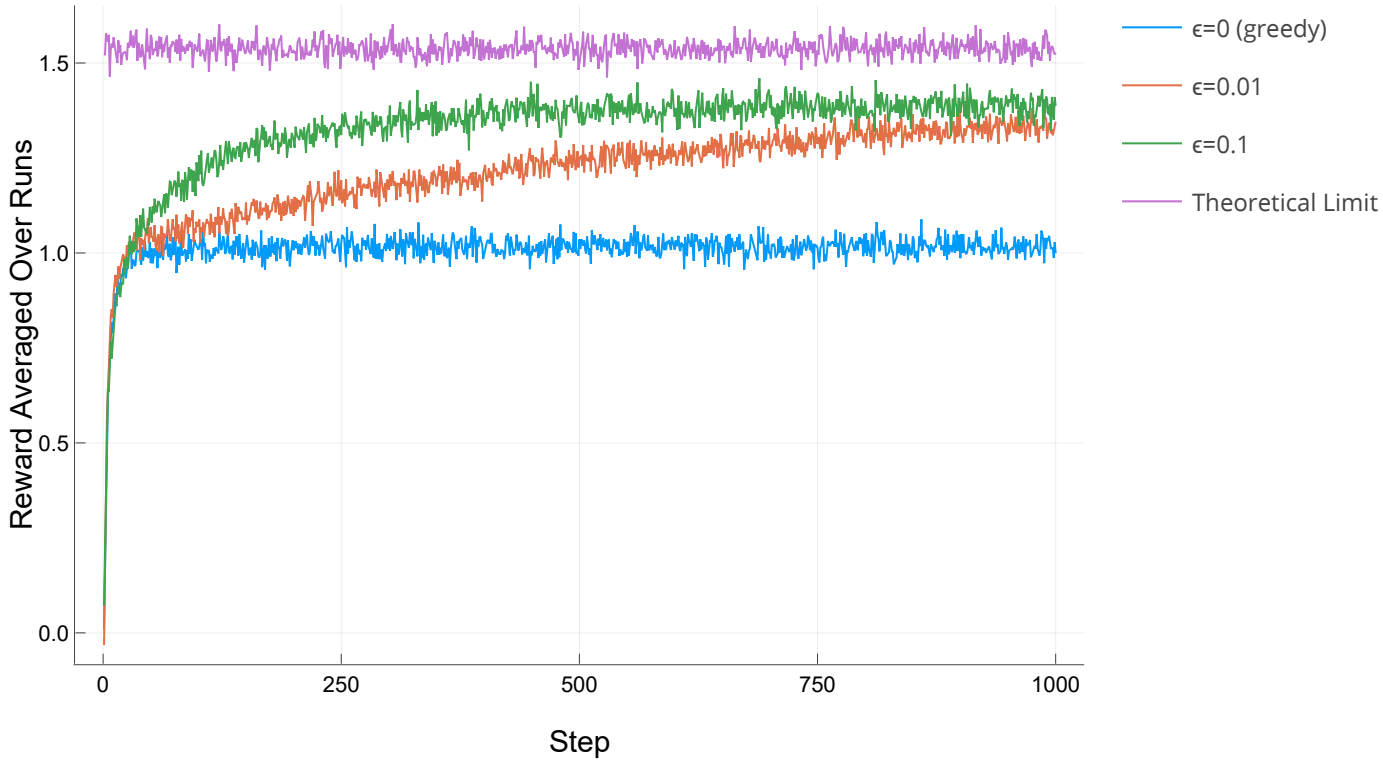
```
qs =
[0.982431, 0.589796, -0.951355, -1.24553, -2.30556, -2.12996, 0.0712543, -0.184582, 1.8807
```

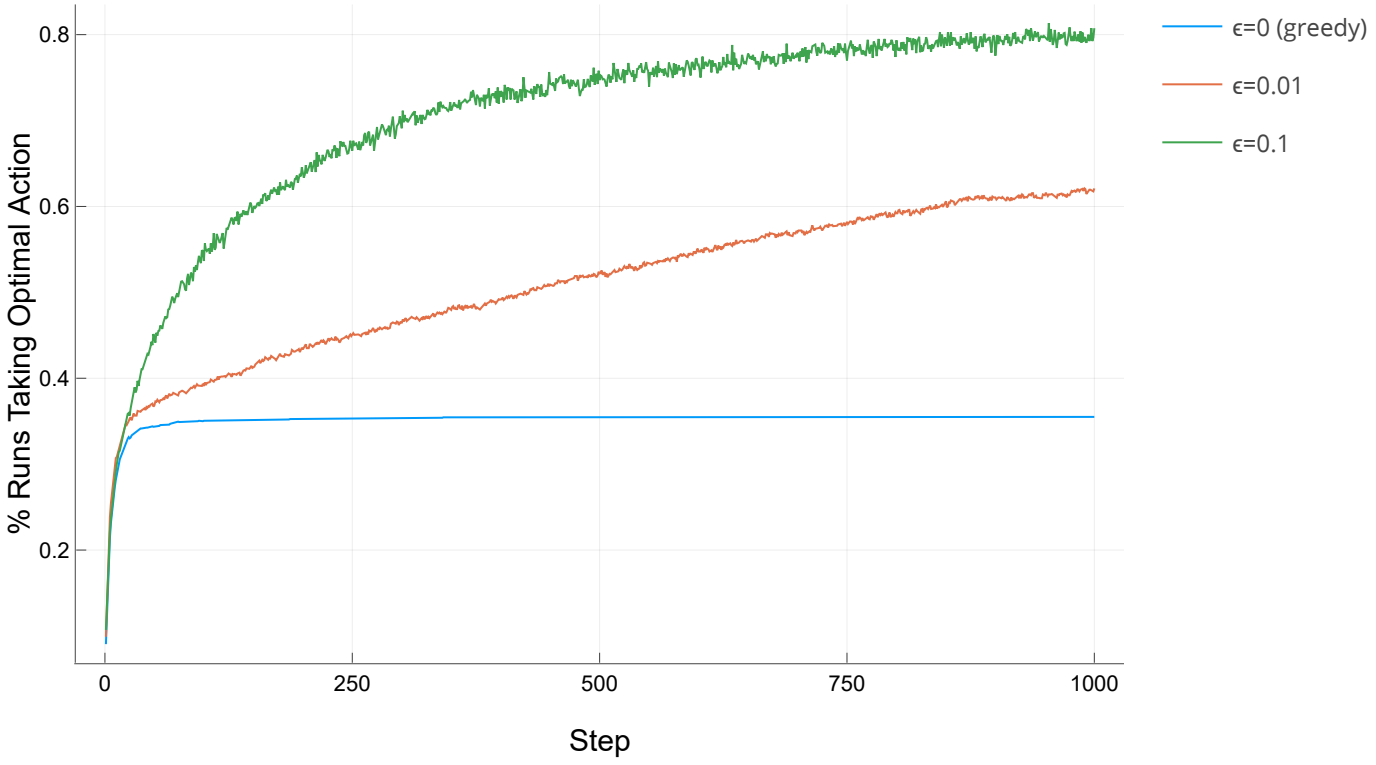
```
qs = create_bandit(k)
```

```
runs =
[([0.0100909, 0.215003, 0.435372, 0.483309, 0.613388, ⋯ more ,0.999531], [1.52093, 1.5228
```

```
runs = [average_simple_runs(k, ϵ) for ϵ in ϵ_list]
```

The following two plots recreate Figure 2.2




% Runs Taking Optimal Action
0.8
0.6
0.4
0.2
0
250
500
750
1000
Step
ϵ=0 (greedy)
ϵ=0.01
ϵ=0.1

Exercise 2.2: Bandit example Consider a k-armed bandit problem with k = 4 actions, denoted 1, 2, 3, and 4. Consider applying to this problem a bandit algorithm using ϵ-greedy action selection, sample-average action-value estimates, and initial estimates of $Q_1(a) = 0$, for all a. Suppose the initial sequence of actions and rewards is $A_1 = 1$, $R_1 = -1$, $A_2 = 2$, $R_2 = 1$, $A_3 = 2$, $R_3 = -2$, $A_4 = 2$, $R_4 = 2$, $A_5 = 3$, $R_5 = 0$. On some of these time steps the ϵ case may have occurred, causing an action to be selected at random. On which time steps did this definitely occur? On which time steps could this possibly have occurred?

The table below summarizes the actions taken leading into every step and the Q estimate for each action at the end of each step. So step 0 shows the initial Q estimates of 0 for every action and the selected action 1 that generates the reward on step 1. For the row in step 1 it shows the Q estimates after receiving the reward on step 1 and thus what actions are demanded by a greedy choice leading into the next step. If the action selected is not equal to or in the set of greedy actions, then a random action **must** have occured. Since a random action choice can also select one of the greedy actions, such a random choice is possible at every step and not necessary to list. Note that the answer in row 0 corresponds to action A_1, row 1 -> A_2 etc...

Step	Q_1	Q_2	Q_3	Action Selected	Greedy Action	Surely Random
0	0	0	0	1	1-3	no
1	-1	0	0	2	2-3	no
2	-1	1	0	2	2	no
3	-1	-.5	0	2	3	yes
4	-1	0.5	0	3	2	yes
5	-1	0.5	0	n/a	n/a	n/a

> *Exercise* 2.3 In the comparison shown in Figure 2.2, which method will perform best in the long run in terms of cumulative reward and probability of selecting the best action? How much better will it be? Express your answer quantitatively.

In the long run both the $\epsilon = 0.1$ and $\epsilon = 0.01$ methods will have Q value estimates that converge to the true mean value of the reward distribution. Since both methods will necessarily take random actions 10% and 1% of the time respectively, we'd expect each method to take the optimal action with probability $(1 - \epsilon) + \epsilon \times \frac{1}{10} = \frac{10 - 9 \times \epsilon}{10}$. So for each value of ϵ we have.

$$Pr(a = a_{best} | \epsilon = 0.1) = 0.91$$

$$Pr(a = a_{best} | \epsilon = 0.01) = 0.991$$

For the $\epsilon = 0$ greedy case the expected reward and optimal action selection probability depends on the order of sampled actions and the likelihood of getting close to or on the optimal action enough to push its Q estimation to the top. From the plots in figure 2.2 in practice that seems to lead to an average reward of ~1.05 and an optimal action selection probability of 0.3825. For long term cummulative reward this case will have roughly $1.05 \times num\_steps$. For the other two cases, the long term cummulative reward is based on the expected value of the highest reward mean which is empiracally ~1.55. For a random action the expected reward should be 0 due to the normal distribution of the action mean rewards.

$$E(long\_term\_step\_reward | \epsilon = 0.1) = 0.91 \times 1.55 = 1.41$$

$$E(long\_term\_step\_reward | \epsilon = 0.01) = 0.991 \times 1.55 = 1.536$$

For each case the long run cumulative reward is just this long term expected reward per step times the number of steps.

2.5 Tracking a Nonstationary Problem

Exercise 2.4 If the step-size parameters, α_n, are not constant, then the estimate Q_n is a weighted average of previously received rewards with a weighting different from that given by (2.6). What is the weighting on each prior reward for the general case, analogous to (2.6), in terms of the sequence of step-size parameters?

From (2.6): $Q_{n+1} = Q_n + \alpha[R_n - Q_n]$ so here we consider the case where α is not a constant but rather can have a unique value for each step n. All expressions below are expansions of the right side of the equation.

$$= Q_n + \alpha_n[R_n - Q_n]$$

$$= \alpha_n R_n + (1-\alpha_n)Q_n$$

$$= \alpha_n R_n + (1-\alpha_n)[\alpha_{n-1}R_{n-1} + (1-\alpha_{n-1})Q_{n-1}]$$

$$= \alpha_n R_n + (1-\alpha_n)\alpha_{n-1}R_{n-1} + (1-\alpha_n)(1-\alpha_{n-1})Q_{n-1}$$

$$= \alpha_n R_n + (1-\alpha_n)\alpha_{n-1}R_{n-1} + (1-\alpha_n)(1-\alpha_{n-1})[\alpha_{n-2}R_{n-2} + (1-\alpha_{n-2})Q_{n-2}]$$

$$= \alpha_n R_n + (1-\alpha_n)\alpha_{n-1}R_{n-1} + (1-\alpha_n)(1-\alpha_{n-1})\alpha_{n-2}R_{n-2} + \dots$$

$$= Q_1\prod_{i=1}^{n}(1-\alpha_i) + \sum_{i=1}^{n}\left[(R_i\alpha_i)\prod_{j=i+1}^{n}(1-\alpha_j)\right]$$

For example if $\alpha_i = 1/i$ then the product in the first term is 0 and the formula becomes:

$$Q_{n+1} = \sum_{i=1}^{n}\left[\frac{R_i}{i}\prod_{j=i+1}^{n}\frac{j-1}{j}\right]$$

$$= \sum_{i=1}^{n}\left[\frac{R_i}{i}\frac{i}{i+1}\frac{i+1}{i+2}\cdots\frac{n-1}{n}\right]$$

$$= \sum_{i=1}^{n}\frac{R_i}{n}$$

from the expanded product we can see that all of the numerators and denominators cancel out leaving only $\frac{R_i}{n}$ which we expect for tihs form of α which was derived earlier for a simple running average.

Exercise 2.5 (programming) Design and conduct an experiment to demonstrate the difficulties that sample-average methods have for nonstationary problems. Use a modified version of the 10-armed testbed in which all the q⇤(a) start out equal and then take independent random walks (say by adding a normally distributed increment with mean 0 and standard deviation 0.01 to all the q⇤(a) on each step). Prepare plots like Figure 2.2 for an action-value method using sample averages, incrementally computed, and another action-value method using a constant step-size parameter, ↵ = 0.1. Use " = 0.1 and longer runs, say of 10,000 steps.

See code and figures below for answer

```
nonstationary_algorithm (generic function with 1 method)
```

```
function nonstationary_algorithm(k::Integer, ϵ::AbstractFloat; steps = 10000, σ = 0.01, α = 0.0)
    qs = zeros(k)
    Q = zeros(k)
    N = zeros(k)
    accum_reward = 0.0
    step_reward = zeros(steps)
    accum_reward_ideal = 0.0
    step_reward_ideal = zeros(steps)
    cum_reward_ideal = zeros(steps)
    cum_reward = zeros(steps)
    optimalcount = 0
    optimalaction_pct = zeros(steps)
    optimalstep = fill(false, steps)
    actions = collect(1:k)
    for i = 1:steps
        shuffle!(actions) #so that ties are broken randomly with argmax
        a = if rand() < ϵ
            rand(actions)
        else
            actions[argmax(Q[actions])]
        end
        optimalaction = argmax(qs)
        if a == optimalaction
            optimalcount += 1
            optimalstep[i] = true
        end
        bandit(a) = sample_bandit(a, qs)
        step_reward[i] = bandit(a)
        step_reward_ideal[i] = bandit(optimalaction)
        accum_reward_ideal += step_reward_ideal[i]
        accum_reward += step_reward[i]
        cum_reward_ideal[i] = accum_reward_ideal
        cum_reward[i] = accum_reward
        optimalaction_pct[i] = optimalcount / i
        N[a] += 1.0
        if α == 0.0
            Q[a] += (1.0/N[a])*(step_reward[i] - Q[a])
        else
            Q[a] += α*(step_reward[i] - Q[a])
        end
        qs .+= randn(k) .*σ #update q values with random walk
    end
    return (;Q, step_reward, optimalstep, step_reward_ideal, cum_reward, cum_reward_ideal, optimalaction_pct)
end
```

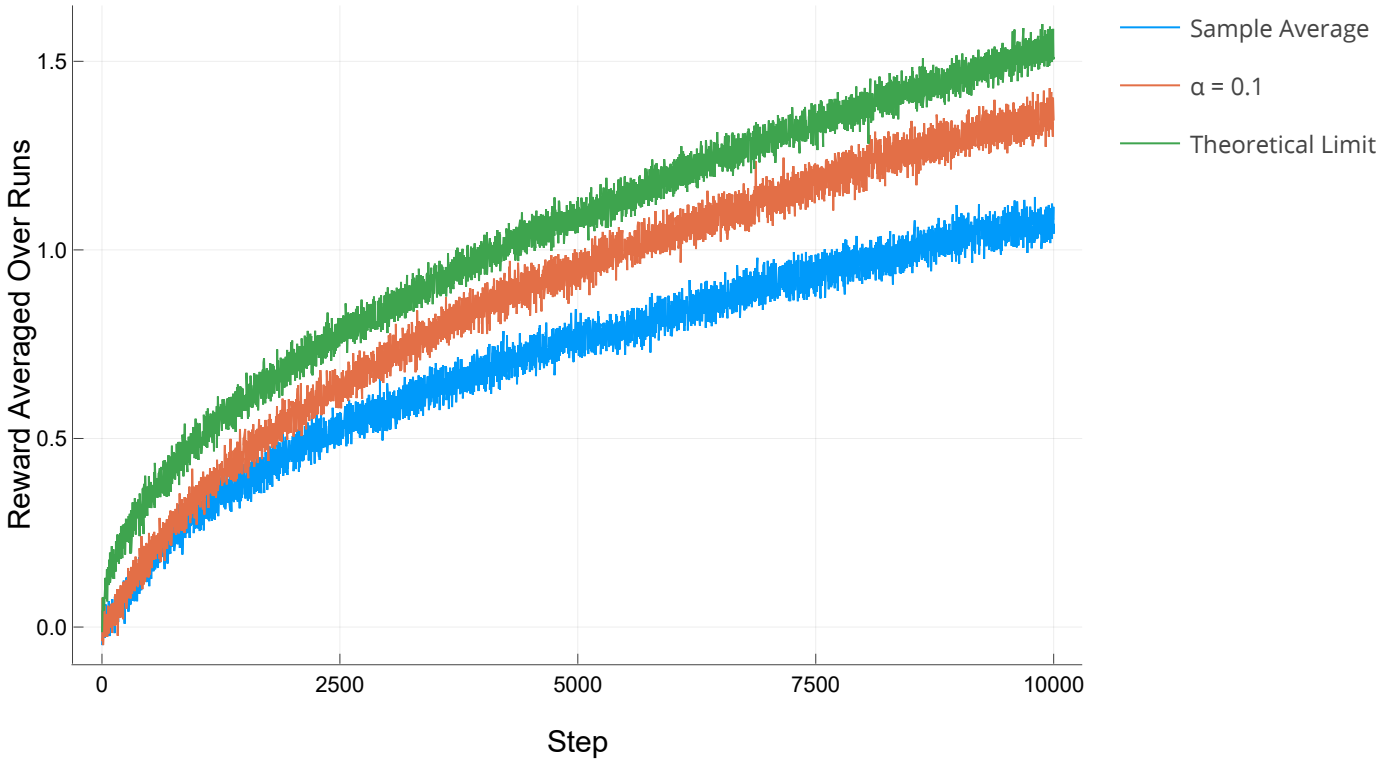
```
average_nonstationary_runs (generic function with 1 method)
```

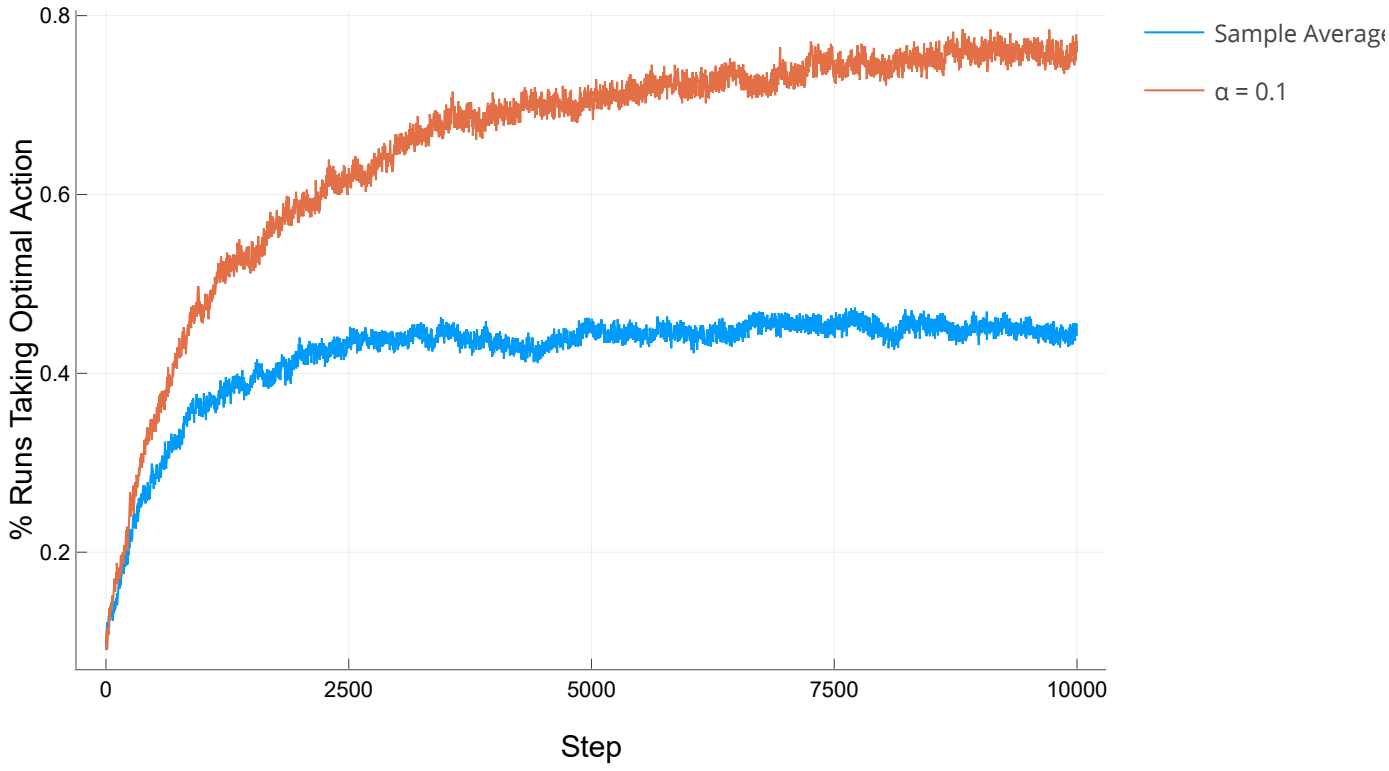
```
sample_average_run =
▸ ([0.0439943, -0.00923059, 0.0189733, 0.0394079, 0.00926411, -0.047898, -0.0244272, -0.0011
```

```
sample_average_run = average_nonstationary_runs(10, 0.1, 0.0)
```

```
constant_step_update_run =
▸ ([-0.031279, -0.0372005, -0.0137845, -0.0173931, -0.0200386, 0.0260778, 0.0776436, -0.0194
```

```
constant_step_update_run = average_nonstationary_runs(10, 0.1, 0.1)
```





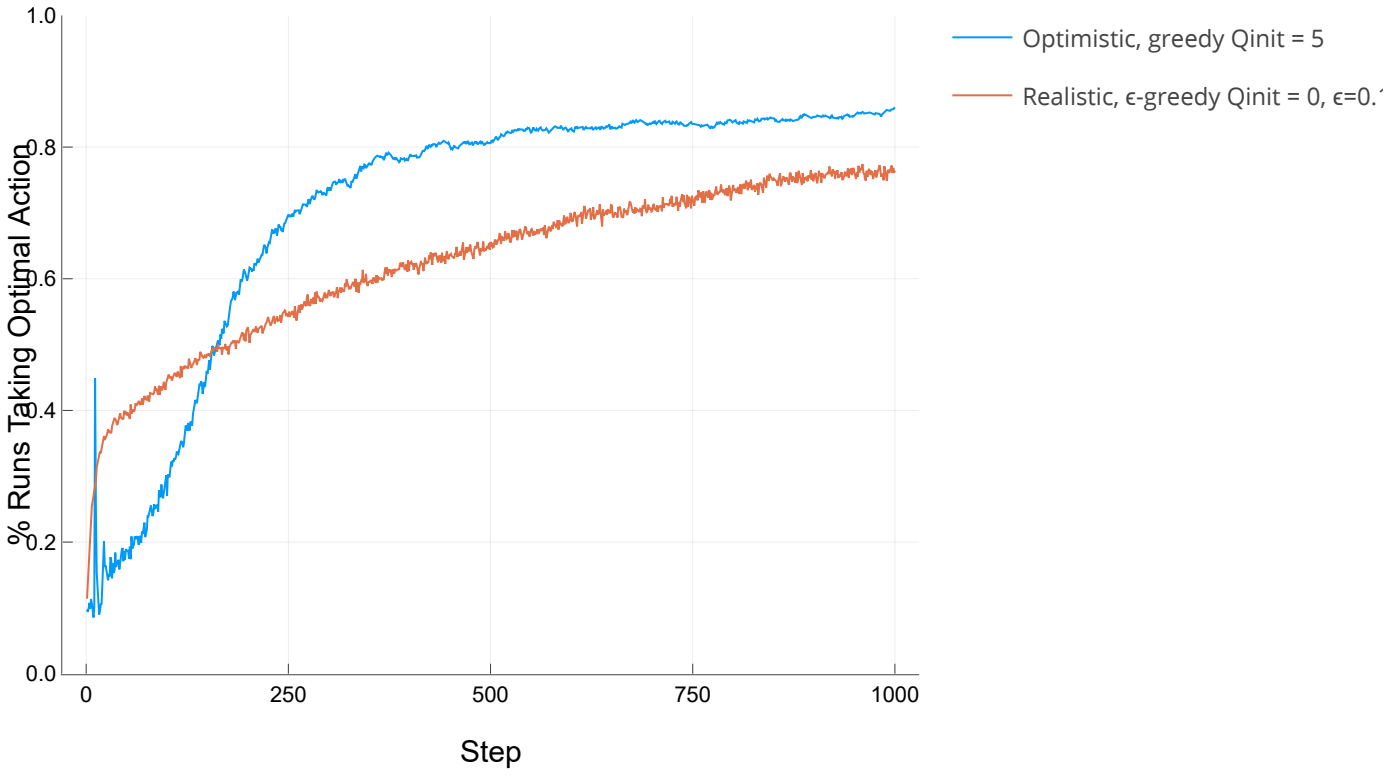
2.6 Optmisitic Initial Values

```
optimistic_greedy_runs =
▸ ([-0.0467812, 0.0110457, -0.0561513, -0.00532945, -0.000739888, 0.0352612, 0.0400952, -0.0
```

```
optimistic_greedy_runs = average_simple_runs(k, 0.0, Qinit = 5.0, α = 0.1)
```

```
realistic_ϵ_runs =
▸ ([0.0168352, 0.249116, 0.36556, 0.525634, 0.587257, 0.632847, 0.707907, 0.741762, 0.735042
```

```
realistic_ϵ_runs = average_simple_runs(k, 0.1, α = 0.1)
```



Exercise 2.6: Mysterious Spikes The results shown in Figure 2.3 should be quite reliable because they are averages over 2000 individual, randomly chosen 10-armed bandit tasks. Why, then, are there oscillations and spikes in the early part of the curve for the optimistic method? In other words, what might make this method perform particularly better or worse, on average, on particular early steps?

The spike occurs on step 11. Due to the initial Q values it is almost 100% likely that a given run with sample each of the 10 possible actions once before repeating any. That would mean that at any given step only 10% of the runs would select the optimal action and indeed for the first 10 steps about 10% of the runs are selecting the optimal action as we'd expect from random chance. On the 11th step, the Q value estimate for each action is $(0.9 \times 5) + (0.1 \times action\_reward)$. The optimal action for each bandit has the highest mean reward, but there is some chance that one of the other 10 actions produced a higher reward the step it was sampled. However, on the 11th step, the number of runs that select the optimal action will be equal to the probability that the optimal action produced the highest reward when it was sampled which empirally is ~44%. Due to the Q value initialization though, the reward on step 11 for those cases that selected the optimal action will almost certainly lower the Q value estimate for that action below the others resulting in the sudden drop of the optimal action selection on step 12.

Exercise 2.7: Unbiased Constant-Step-Size Tick In most of this chapter we have used sample averages to estimate action values because sample averages do not produce the initial bias that constant step sizes do (see analysis leading to (2.6)). However, sample averages are not a completely satisfactory solution because they may perform poorly on nonstationary problems. Is it possible to avoid the bias of constant sample sizes while retaining their advantages on nonstationary problems? One way is to use a step size of

$$\beta_n \doteq \alpha / \bar{o}_n,$$

to process the nth reward for a particular action, where $\alpha > 0$ is a conventional constant step size, and $\bar{o}_n$ is a trace of one that starts at 0:

$$\bar{o}_n \doteq \bar{o}_{n-1} + \alpha(1 - \bar{o}_{n-1}), \text{ for } n \geq 0, \text{ with } \bar{o}_0 \doteq 0.$$

Carry out an analysis like that in (2.6) to show that Q_n is an exponential recency-weighted average *without initial bias*.

$$Q_{n+1} = Q_n + \beta_n [R_n - Q_n]$$

where $\beta_n \doteq \alpha / \bar{o}_n$ and $\bar{o}_n \doteq \bar{o}_{n-1} + \alpha(1 - \bar{o}_{n-1})$

$$\bar{o}_n = \bar{o}_{n-1} + \alpha(1 - \bar{o}_{n-1}) = \bar{o}_{n-1}(1 - \alpha) + \alpha$$

We can expand $\bar{o}_n$ backwards to get an explicit formula.

$$\bar{o}_n = \bar{o}_{n-1}(1 - \alpha) + \alpha,$$

$$\bar{o}_n = (\bar{o}_{n-2}(1 - \alpha) + \alpha)(1 - \alpha) + \alpha$$

$$\bar{o}_n = \bar{o}_{n-2}(1 - \alpha)^2 + \alpha((1 - \alpha) + 1)$$

$$\bar{o}_n = (\bar{o}_{n-3}(1 - \alpha) + \alpha)(1 - \alpha)^2 + \alpha((1 - \alpha) + 1)$$

$$\bar{o}_n = \bar{o}_{n-3}(1 - \alpha)^3 + \alpha((1 - \alpha^2) + (1 - \alpha) + 1)$$

$$\vdots$$

$$\bar{o}_n = \bar{o}_0(1 - \alpha)^n + \alpha \sum_{i=0}^{n-1} (1 - \alpha)^i = \alpha \sum_{i=0}^{n-1} (1 - \alpha)^i$$

This sum has an explicit formula as can be seen by:

$$S = 1 + (1-\alpha) + (1-\alpha)^2 + \cdots + (1-\alpha)^{n-1}$$

$$S(1-\alpha) = (1-\alpha) + \cdots + (1-\alpha)^n = S - 1 + (1-\alpha)^n$$

$$-S\alpha = -1 + (1-\alpha)^n$$

$$S = \frac{1-(1-\alpha)^n}{\alpha}$$

Therefore, $\bar{o}_n = \alpha\frac{1-(1-\alpha)^n}{\alpha} = 1-(1-\alpha)^n$, and since $0 < \alpha < 1$, then $(1-\alpha)^n \to 0 \text{ as } n \to \inf$.

$$\beta_n = \frac{\alpha}{\bar{o}_n} = \frac{\alpha}{1-(1-\alpha)^n} \implies \beta_1 = 1$$

From exercise 2.4, we have the formula for Q_n with a non-constant coefficient α_n which we can trivially replace here with β_n

$$Q_n = Q_1 \prod_{i=1}^{n}(1-\beta_i) + \sum_{i=1}^{n}\left[R_i\beta_i \prod_{j=i+1}^{n}(1-\beta_j)\right]$$

Since $\beta_1 = 1$, the product associated with Q_1 will be 0. Since there is no dependency on the initial value of Q, we can say this formula for updating Q has *no initial bias*. If we then make the substitution $\beta_n = \frac{\alpha}{1-(1-\alpha)^n}$, we have

$$Q_n = \sum_{i=1}^{n}\left[R_i\frac{\alpha}{1-(1-\alpha)^i}\prod_{j=i+1}^{n}\left(1-\frac{\alpha}{1-(1-\alpha)^j}\right)\right]$$

$$Q_n = \alpha\sum_{i=1}^{n}\left[\frac{R_i}{1-(1-\alpha)^i}\prod_{j=i+1}^{n}\left(\frac{(1-\alpha)(1-(1-\alpha)^{j-1})}{1-(1-\alpha)^j}\right)\right]$$

$$Q_n = \alpha\sum_{i=1}^{n}\left[\frac{R_i(1-\alpha)^{n-i}}{1-(1-\alpha)^i}\prod_{j=i+1}^{n}\left(\frac{1-(1-\alpha)^{j-1}}{1-(1-\alpha)^j}\right)\right]$$

Examining the product term on its own, we can see it simplifies.

$$\prod_{j=i+1}^{n}\left(\frac{1-(1-\alpha)^{j-1}}{1-(1-\alpha)^j}\right)$$

$$\frac{1-(1-\alpha)^i}{1-(1-\alpha)^{i+1}}\frac{1-(1-\alpha)^{i+1}}{1-(1-\alpha)^{i+2}}\cdots\frac{1-(1-\alpha)^{n-1}}{1-(1-\alpha)^n} = \frac{1-(1-\alpha)^i}{1-(1-\alpha)^n} \text{ for i}\leq\text{n}$$

Replacing this expression for the product in the expression for Q_n we have:

$$Q_n = \alpha \sum_{i=1}^{n} \left[\frac{R_i (1-\alpha)^{n-i}}{1-(1-\alpha)^i} \frac{1-(1-\alpha)^i}{1-(1-\alpha)^n} \right] = \frac{\alpha}{1-(1-\alpha)^n} \sum_{i=1}^{n} R_i (1-\alpha)^{n-i}$$

If we expand this sum going backwards from $i = n$:

$$Q_n = \frac{\alpha}{1-(1-\alpha)^n} \left[R_n + R_{n-1}(1-\alpha) + R_{n-2}(1-\alpha)^2 + \cdots + R_1 (1-\alpha)^{n-1} \right]$$

The constant term starts off at 1 for $n = 1$ and approaches α in the limit of $n \to \text{inf}$. If $0 < \alpha < 1$, then the coefficients in the sum section for R_i decrease exponentially from 1 for $i = n$ to $(1-\alpha)^{n-1}$ for $i = 1$. So the average over rewards includes every reward back to R_1 like the simple average but the coefficients become exponentially smaller approaching 0 as $n \to \text{inf}$.

2.7 Upper-Confidence-Bound Action Selection

```
c = 2.0
```

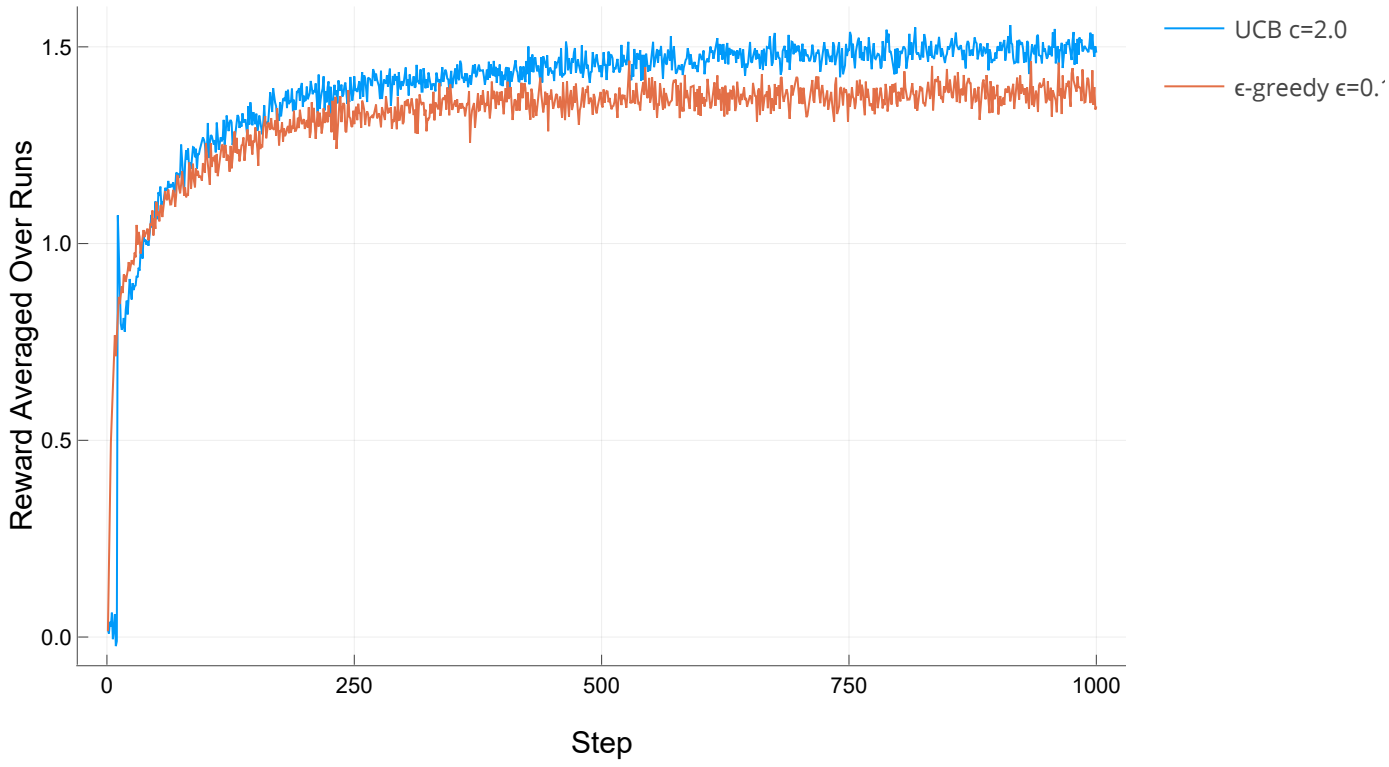
```
c = 2.0
```

```
ucb_runs =
([0.0182253, 0.00893618, 0.0387318, 0.026957, 0.0619752, -0.00544705, 0.0205782, 0.0577078
```

```
ucb_runs = average_simple_runs(k, 0.0, c = c)
```

```
ϵ_runs =
([0.0150056, 0.277806, 0.409952, 0.494397, 0.564805, 0.67186, 0.704051, 0.766899, 0.71384,
```

```
ϵ_runs = average_simple_runs(k, 0.1)
```



Exercise 2.8: UCB Spikes In Figure 2.4 the UCB algorithm shows a distinct spike in performance on the 11th step. Why is this? Note that for your answer to be fully satisfactory it must explain both why the reward increases on the 11th step and why it decreases on the subsequent steps. Hint: If $c = 1$, then the spike is less prominent.

Due to the UCB calculation, any state that has not been visited will have an infinite Q value, thus for the first 10 steps similar to the optimistic Q initialization each run will sample each of the 10 possible actions. One the 11th step, the exploration incentive for each action will be equal, so the most likely action to be selected is the optimal action since it is the most likely to have produced a reward higher than any other action. If c is very large, then on step 12 no matter how good of a reward we received for the optimal action, that action will be penalized compared to the others because it will have double the visit count. In particular for c = 2.0, the exploration bonus for the optimal action if selected on step 11 will be 2.2293 vs 3.31527 for all other actions. Since the q's are normally distributed it is unlikely that the reward average for the optimal action is >1 than the next best action. The larger c is the more of a relative bonus the other actions have and the probability of selecting the optimal action twice in a row drops to zero. As c changes the improved reward on step 11 remains similar but the dropoff on step 12 becomes more severe the larger c is.

2.8 Gradient Bandit Algorithms

> *Exercise 2.9* Show that in the case of two actions, the soft-max distribution is the same as that given by the logistic, or sigmoid, function often used in statistics and artificial neural networks.

The sigmoid function is defined as: $S(x) = \frac{1}{1+e^{-x}}$. For two actions, let's denote them a_1 and a_2. Now for the action probabilities we have.

$$\pi(a_1) = \frac{e^{H_t(a_1)}}{e^{H_t(a_1)} + e^{H_t(a_2)}} = \frac{1}{1 + e^{-(H_t(a_1) - H_t(a_2))}}$$

This expression for $\pi(a_1)$ is equivalent to $S(x)$ with $x = H_t(a_1) - H_t(a_2)$ which is the degree of preference for action 1 over action 2. As expected, if the preferences are equal then it is equavalent to $x = 0$ with a probability of 50%. The same analysis applies to action 2 with the actions reversed from this case.

```
calc_πvec (generic function with 1 method)
```

```
#calculates a vector of probabilities for selecting each action given exponentiated "perferences" given in expH using the softmax distribution
function calc_πvec(expH::AbstractVector)
    s = sum(expH)
    expH ./ s
end
```

```
update_H! (generic function with 1 method)
```

```
function update_H!(a::Integer, H::AbstractVector, π_vec::AbstractVector, α, R, R̄)
    for i in eachindex(H)
        H[i] = H[i] + α*(R - R̄)*(i == a ? (1.0 - π_vec[i]) : -π_vec[i])
    end
end
```

```
#necessary for doing weighted sampling
using StatsBase ✓
```

```
sample_action (generic function with 1 method)
```

```
sample_action(actions, π_vec) = sample(actions, pweights(π_vec))
```

```
gradient_stationary_bandit_algorithm (generic function with 1 method)
```

```
function gradient_stationary_bandit_algorithm(qs::Vector{Float64}, k::Integer; steps = 1000, α = 0.1, baseline = true)
    bandit(a) = sample_bandit(a, qs)
    H = zeros(k)
    expH = exp.(H)
    π_vec = calc_πvec(expH)
    R̄ = 0.0
    accum_reward_ideal = 0.0
    accum_reward = 0.0
    cum_reward_ideal = zeros(steps)
    step_reward_ideal = zeros(steps)
    cum_reward = zeros(steps)
    step_reward = zeros(steps)
    bestaction = argmax(qs)
    optimalstep = fill(false, steps)
    optimalcount = 0
    optimalaction_pct = zeros(steps)
    actions = collect(1:k)
    for i = 1:steps
        a = sample_action(actions, π_vec)
        if a == bestaction
            optimalstep[i] = true
            optimalcount += 1
        end
        step_reward[i] = bandit(a)
        step_reward_ideal[i] = bandit(bestaction)
        accum_reward_ideal += step_reward_ideal[i]
        cum_reward_ideal[i] = accum_reward_ideal
        accum_reward += step_reward[i]
        cum_reward[i] = accum_reward
        optimalaction_pct[i] = optimalcount / i
        update_H!(a, H, π_vec, α, step_reward[i], R̄)

        #update R̄ with running average if baseline is true
        if baseline
            R̄ += (1.0/i)*(step_reward[i] - R̄)
        end

        #update expH and π_vec
        expH .= exp.(H)
        π_vec .= calc_πvec(expH)
    end
    return (;step_reward, step_reward_ideal, cum_reward, cum_reward_ideal, optimalstep, optimalaction_pct)
end
```

```
(step_reward = [-3.23564, -2.26858, 0.257366, -1.99482, -3.05336, -0.446014, 1.54878, -2.2
```

```
gradient_stationary_bandit_algorithm(create_bandit(k), k)
```

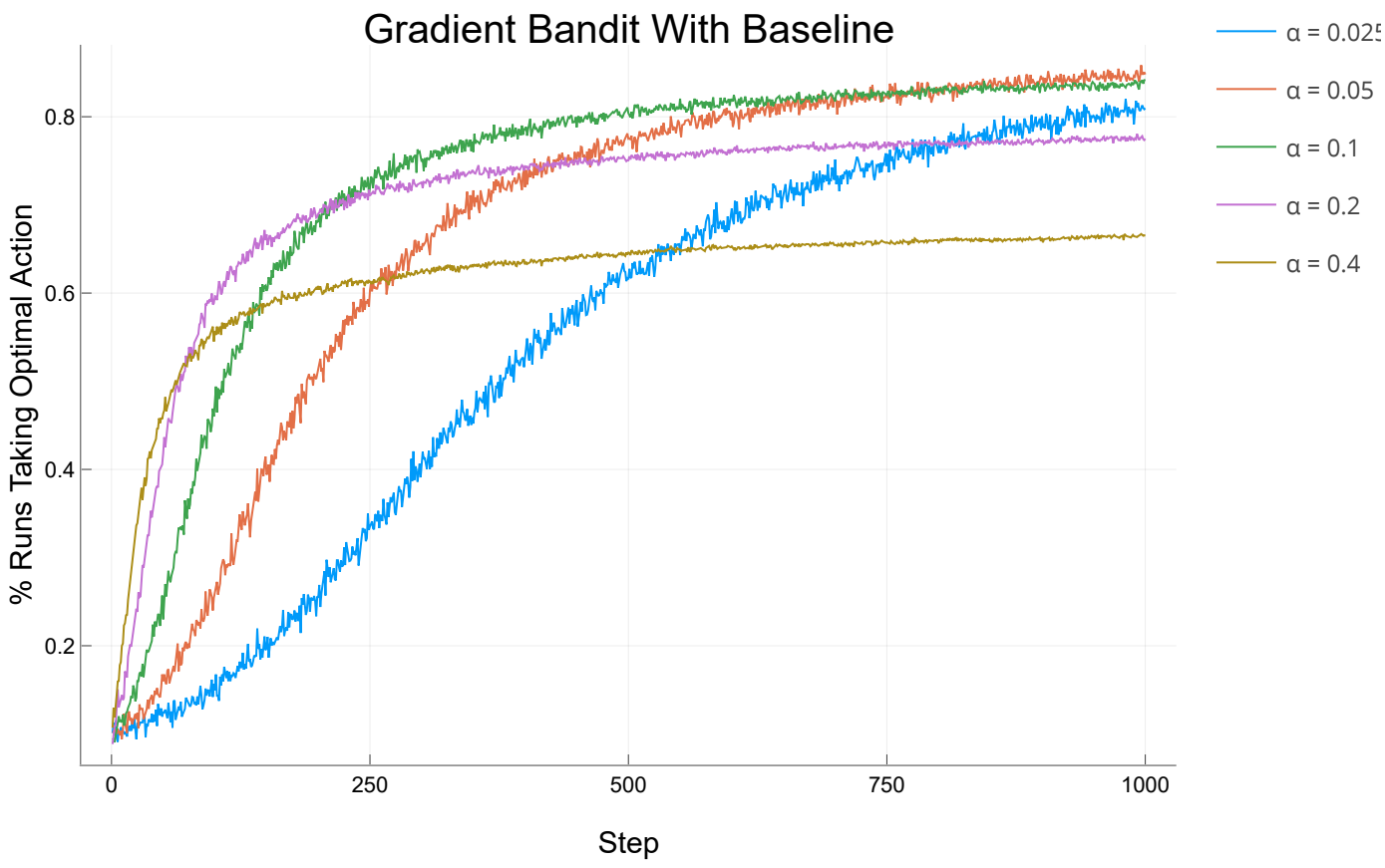
```
average_gradient_stationary_runs (generic function with 1 method)
```

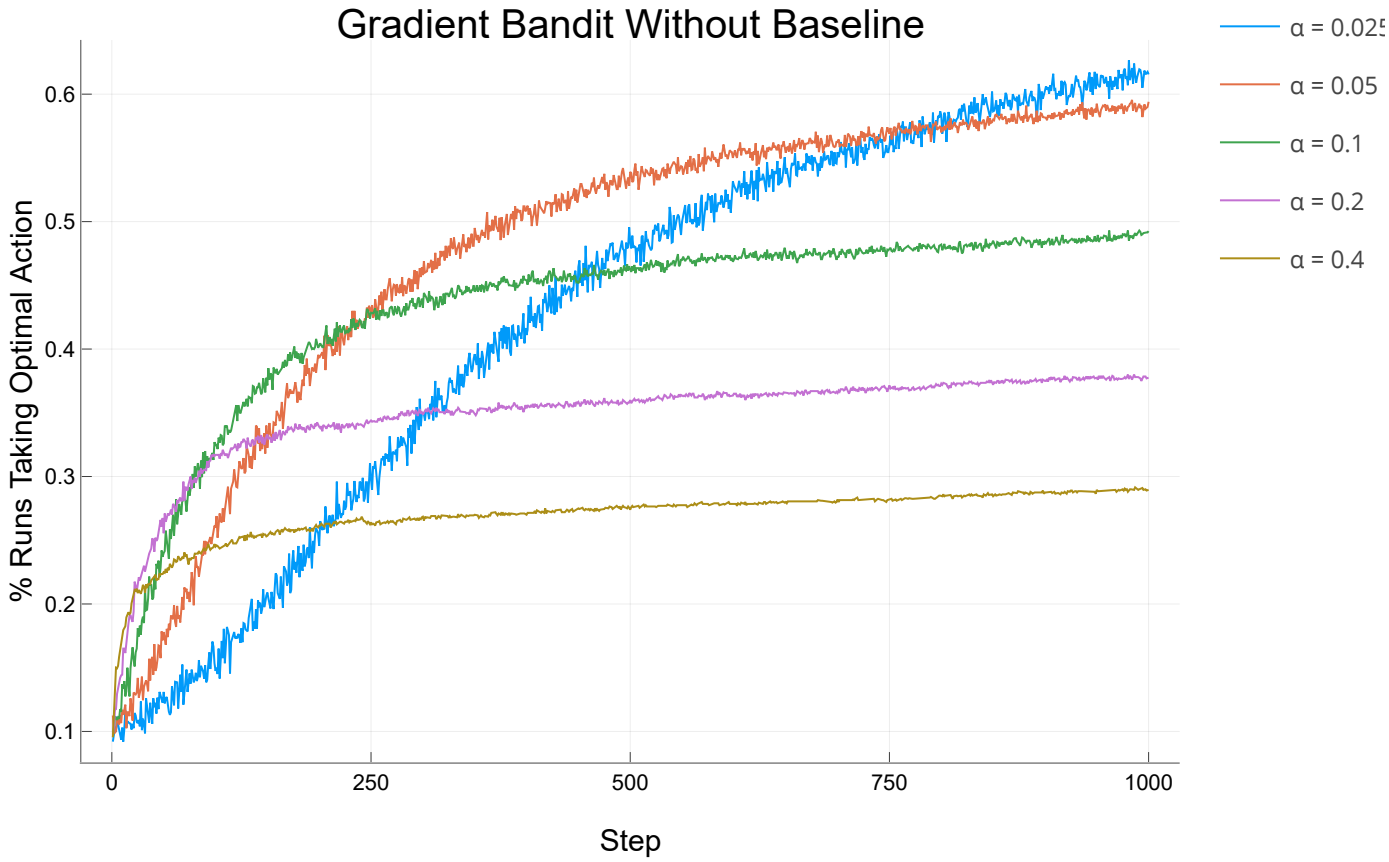
```
function average_gradient_stationary_runs(k; steps = 1000, n = 2000, α=0.1, offset = 0.0, baseline = true)
    runs = Vector{NamedTuple}(undef, n)
    @threads for i in 1:n
        qs = create_bandit(k, offset = offset)
        runs[i] = gradient_stationary_bandit_algorithm(qs, k, steps = steps, α = α, baseline = baseline)
    end
    map(i -> mapreduce(a -> a[i], (a, b) -> a .+ b, runs)./n, (:step_reward, :step_reward_ideal, :optimalstep, :cum_reward, :cum_reward_ideal, :optimalaction_pct))
end
```

```
α_list = [0.025, 0.05, 0.1, 0.2, 0.4]
```

```
α_list = [0.025, 0.05, 0.1, 0.2, 0.4]
```

For the plots below the bandit parameters were created with an offset of 4.0 so the expected reward value for any action is centered at 4.0 instead of 0. For the gradient bandit with a baseline, it doesn't affect the curves at all, but if the baseline is removed then the results are worse as seen in the second plot. However, if α is made smaller it seems like it will also converge to a similar success rate just over a longer time. The optimal value of α is much lower than when the baseline is removed"





Code Refactoring

Due to the variety of algorithms and parameters for the bandit, I have rewritten the test environment with types that represent the different algorithms. The run simulator will dispatch on the types to correctly simulate that method with its parameters. Some of the previous simluations are plots are generated again. Because of the style used, only one simulation function is needed with the flexibility to select any combination of techniques in the chapter.

```
run_bandit (generic function with 1 method)
```

```
function run_bandit(qs::Vector{T}, algorithm::BanditAlgorithm{T}; steps = 1000, μ::T
= zero(T), σ::T = zero(T)) where T <: AbstractFloat
    bandit(a) = sample_bandit(a, qs)
    #in this case the bandit is not stationary
    updateq = (μ != 0) || (σ != 0)
    function qupdate!(qs)
        for i in eachindex(qs)
            qs[i] += (randn()*σ) + μ
        end
        return nothing
    end
    #initialize values to keep track of
    actions = collect(eachindex(qs))
    accum_reward_ideal = zero(T)
    accum_reward = zero(T)
    cum_reward_ideal = zeros(T, steps)
    step_reward_ideal = zeros(T, steps)
    cum_reward = zeros(T, steps)
    step_reward = zeros(T, steps)
    optimalstep = fill(false, steps)
    optimalcount = 0
    optimalaction_pct = zeros(T, steps)
    bestaction = argmax(qs)
    for i = 1:steps
        a = sample_action(algorithm, i, actions)
        # a = 1
        if a == bestaction
            optimalstep[i] = true
            optimalcount += 1
        end
        step_reward[i] = bandit(a)
        step_reward_ideal[i] = bandit(bestaction)
        accum_reward_ideal += step_reward_ideal[i]
        cum_reward_ideal[i] = accum_reward_ideal
        accum_reward += step_reward[i]
        cum_reward[i] = accum_reward
        optimalaction_pct[i] = optimalcount / i
        #update anything required before sampling the next action
        update_estimator!(algorithm, a, step_reward[i], i)
        #will only update qs in the non-stationary case and get a new bestaction
        if updateq
            qupdate!(qs)
            bestaction = argmax(qs)
        end
    end
    return (;step_reward, step_reward_ideal, cum_reward, cum_reward_ideal,
optimalstep, optimalaction_pct)
end
```

```
run_bandit_cumreward (generic function with 1 method)
```

```
function run_bandit_cumreward(qs::Vector{T}, algorithm::BanditAlgorithm{T}; steps = 1000, μ::T = zero(T), σ::T = zero(T), cumstart = 1) where T <: AbstractFloat
#same as run_bandit but only saved the average cumulative reward per step, so it is faster
    bandit(a) = sample_bandit(a, qs)
    #in this case the bandit is not stationary
    updateq = (μ != 0) || (σ != 0)
    function qupdate!(qs)
        for i in eachindex(qs)
            qs[i] += (randn(T)*σ) + μ
        end
        return nothing
    end
    #initialize values to keep track of
    actions = collect(eachindex(qs))
    accum_reward_ideal = zero(T)
    accum_reward = zero(T)
    bestaction = argmax(qs)
    for i = 1:steps
        a = sample_action(algorithm, i, actions)
        r = bandit(a)
        r_ideal = bandit(bestaction)
        if i >= cumstart
            accum_reward_ideal += r_ideal
            accum_reward += r
        end
        #update anything required before sampling the next action
        update_estimator!(algorithm, a, r, i)
        #will only update qs in the non-stationary case and get a new bestaction
        if updateq
            qupdate!(qs)
            bestaction = argmax(qs)
        end
    end
    step_cum_reward = accum_reward/(steps - cumstart + 1)
    step_cum_reward_ideal = accum_reward_ideal/(steps - cumstart + 1)
    return (;step_cum_reward, step_cum_reward_ideal)
end
```

```
abstract type Explorer{T <: AbstractFloat} end
```

```
struct ϵ_Greedy{T <: AbstractFloat} <: Explorer{T}
    ϵ::T
end
```

```
#creates additional constructor with default value
(::Type{ϵ_Greedy})() = ϵ_Greedy(0.1)
```

```
#extrends default value to other types
(::Type{ϵ_Greedy{T}})() where T<:AbstractFloat = ϵ_Greedy(T(0.1))
```

```
(::Type{ϵ_Greedy{T}})(e::ϵ_Greedy) where T<:AbstractFloat = ϵ_Greedy(T(e.ϵ))
```

```
#how to convert type of explorer when it is already the same
(::Type{T})(e::T) where T <: ϵ_Greedy = e
```

▸ ϵ_Greedy(1.0)

```
#already knows how to convert if the argument passed is wrong
ϵ_Greedy{BigFloat}(1.0f0)
```

▸ ϵ_Greedy(0.1)

```
ϵ_Greedy()
```

▸ ϵ_Greedy(0.1f0)

```
ϵ_Greedy{Float32}()
```

```
explorer = ▸ ϵ_Greedy(0.1)
```

```
explorer = ϵ_Greedy(0.1)
```

▸ ϵ_Greedy(0.1f0)

```
ϵ_Greedy{Float32}(explorer)
```

```
CodeInfo(
1 —     return e
)
```

```
with_terminal() do
    @code_lowered ϵ_Greedy{Float64}(explorer)
end
```

```
struct UCB{T <: AbstractFloat} <: Explorer{T}
    c::T
end
```

```
(::Type{UCB})() = UCB(2.0)
```

```
(::Type{UCB{T}})() where T <: AbstractFloat = UCB(T(2.0))
```

```
(::Type{T})(e::T) where T <: UCB = e
```

```
(::Type{UCB{T}})(e::UCB) where {T<:AbstractFloat} = UCB(T(e.c))
```

```
ex2 = ▸ UCB(1.0)
```

```
ex2 = UCB(1.0)
```

```
▸ UCB(1.0f0)
```

```
UCB{Float32}(ex2)
```

```
CodeInfo(
1 — %1 = Base.getproperty(e, :c)
│   %2 = ($(Expr(:static_parameter, 1)))(%1)
│   %3 = Main.workspace#2.UCB(%2)
└──      return %3
)
```

```
with_terminal() do
    @code_lowered UCB{Float32}(ex2)
end
```

```
true
```

```
ϵ_Greedy{Float64} <: Explorer{Float64}
```

```
abstract type AverageMethod{T<:AbstractFloat} end
```

```
struct ConstantStep{T<:AbstractFloat} <: AverageMethod{T}
    α::T
end
```

```
mutable struct UnbiasedConstantStep{T<:AbstractFloat}<:AverageMethod{T}
    α::T
    o::T
    UnbiasedConstantStep(α::T) where T<:AbstractFloat = new{T}(α, zero(T))
end
```

```
(::Type{ConstantStep})() = ConstantStep(0.1)
```

```
(::Type{ConstantStep{T}})() where T<:AbstractFloat = ConstantStep(T(0.1))
```

```
(::Type{U})(a::U) where U <: ConstantStep = a
```

```
(::Type{ConstantStep{T}})(a::ConstantStep) where {T<:AbstractFloat} = ConstantStep{T}
(a.α)
```

```
(::Type{UnbiasedConstantStep})() = UnbiasedConstantStep(0.1)
```

```
(::Type{UnbiasedConstantStep{T}})(a::UnbiasedConstantStep) where
T<:AbstractFloat=UnbiasedConstantStep(T(a.α))
```

```
(::Type{U})(a::U) where U<:UnbiasedConstantStep = a
```

```
▸ UnbiasedConstantStep(0.1, 0.0)
```

```
UnbiasedConstantStep()
```

```
q_avg = ▸ ConstantStep(0.1)
```

```
q_avg = ConstantStep()
```

```
▸ ConstantStep(0.1000000000000000055511151231257827021181583404541015625)
```

```
ConstantStep{BigFloat}()
```

```
▸ ConstantStep(0.1f0)
```

```
ConstantStep{Float32}(q_avg)
```

```
struct SampleAverage{T<:AbstractFloat} <: AverageMethod{T}
end
```

```
(::Type{SampleAverage})() = SampleAverage{Float64}()
```

```
(::Type{U})(a::U) where U <: SampleAverage = a
```

```
(::Type{U})(a::SampleAverage) where U <: SampleAverage = U()
```

```
q_avg2 = ▸ SampleAverage()
```

```
q_avg2 = SampleAverage()
```

```
▸ SampleAverage()
```

```
SampleAverage{BigFloat}(q_avg2)
```

```
abstract type BanditAlgorithm{T} end
```

```
(::Type{Explorer{T}})(e::ϵ_Greedy) where T<:AbstractFloat = ϵ_Greedy{T}(e)
```

```
(::Type{Explorer{T}})(e::ϵ_Greedy{T}) where T<:AbstractFloat = e
```

```
(::Type{Explorer{T}})(e::UCB{T}) where T<:AbstractFloat = e
```

```
(::Type{Explorer{T}})(e::UCB) where T<:AbstractFloat = UCB{T}(e)
```

```
(::Type{AverageMethod{T}})(a::SampleAverage) where T<:AbstractFloat = SampleAverage{T}
()
```

```
(::Type{AverageMethod{T}})(a::ConstantStep) where T<:AbstractFloat = ConstantStep{T}
(a)
```

```
(::Type{AverageMethod{T}})(a::UnbiasedConstantStep) where T<:AbstractFloat =
UnbiasedConstantStep{T}(a)
```

```
struct ActionValue{T<:AbstractFloat} <: BanditAlgorithm{T}
    N::Vector{T}
    Q::Vector{T}
    explorer::Explorer{T}
    q_avg::AverageMethod{T}
    #use the type of Qinit to initialize vectors and convert the other types if necessary
    function ActionValue(k::Integer, Qinit::T, explorer::Explorer, q_avg::AverageMethod) where {T<:AbstractFloat}
        N = zeros(T, k)
        Q = ones(T, k) .* Qinit
        new_e = Explorer{T}(explorer)
        new_avg = AverageMethod{T}(q_avg)
        new{T}(N, Q, new_e, new_avg)
    end
end
```

```
est1 =
▸ ActionValue([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [1.0, 1.0, 1.0, 1.0, 1.0, 1
```

```
est1 = ActionValue(10, 1.0f0, ϵ_Greedy(), SampleAverage())
```

```
(::Type{ActionValue})(k::Integer; Qinit = 0.0, explorer::Explorer = ϵ_Greedy(), q_avg::AverageMethod = SampleAverage()) = ActionValue(k, Qinit, explorer, q_avg)
```

```
▸ ActionValue([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 0.0, 0
```

```
ActionValue(10, Qinit = 0.0f0)
```

```
est2 =
▸ ActionValue([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [1.0, 1.0, 1.0, 1.0, 1.0, 1
```

```
est2 = ActionValue(10, Qinit = BigFloat(1.0), explorer = UCB(1.0f0), q_avg = ConstantStep())
```

```
mutable struct GradientReward{T<:AbstractFloat} <: BanditAlgorithm{T}
    H::Vector{T}
    expH::Vector{T}
    πvec::Vector{T}
    α::T
    R̄::T
    update::AverageMethod{T}
    function GradientReward(k::Integer, α::T, update::AverageMethod) where T<:AbstractFloat
        H = zeros(T, k)
        expH = exp.(H)
        πvec = ones(T, k) ./ k
        R̄ = zero(T)
        new_update = AverageMethod{T}(update)
        new{T}(H, expH, πvec, α, R̄, new_update)
    end
end
```

```
(::Type{GradientReward})(k::Integer; α::T=0.1, update::AverageMethod = SampleAverage()) where T<:AbstractFloat = GradientReward(k, α, update)
```

```
grad_est1 =
▸ GradientReward([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [1.0, 1.0, 1.0, 1.0, 1.0
```

```
grad_est1 = GradientReward(10)
```

```
grad_est2 =
▸ GradientReward([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [1.0, 1.0, 1.0, 1.0, 1.0
```

```
grad_est2 = GradientReward(10, update = ConstantStep(0.0f0))
```

```
sample_action (generic function with 2 methods)
```

```
sample_action(est::GradientReward, i::Integer, actions) = sample(actions, weights(est.πvec))
```

```
sample_action (generic function with 3 methods)
```

```
sample_action(est::ActionValue, i::Integer, actions::AbstractVector) = sample_action(est::ActionValue, est.explorer, i, actions)
```

```
sample_action (generic function with 4 methods)
```

```
function sample_action(est::ActionValue, explorer::ϵ_Greedy, i::Integer, actions::AbstractVector)
    shuffle!(actions)
    # ϵ = explorer.ϵ
    if rand() < explorer.ϵ
        rand(actions)
    else
        actions[argmax(view(est.Q, actions))]
    end
end
```

```
sample_action (generic function with 5 methods)
```

```julia
function sample_action(est::ActionValue, explorer::UCB, i::Integer,
actions::AbstractVector)
    (Q, N, c) = (est.Q, est.N, explorer.c)
    argmax(view(Q, actions) .+ c .* sqrt.(log(i) ./ view(N, actions)))
    # argmax(est.Q[actions] .+ (est.explorer.c .* sqrt.(log(i) ./ est.N[actions])))
end
```

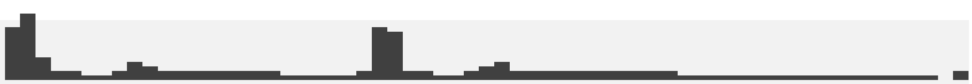
```
BenchmarkTools.Trial: 10000 samples with 376 evaluations.
 Range (min … max):  257.713 ns … 397.340 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     280.851 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   285.394 ns ±  14.280 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
  258 ns           Histogram: frequency by time          336 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

```julia
with_terminal() do
    actions = collect(1:10)
    est = ActionValue(10)
    # @code_warntype sample_action(est, 10, actions)
    @benchmark sample_action($est, 10, $actions)
end
```

```
BenchmarkTools.Trial: 10000 samples with 788 evaluations.
 Range (min … max):  158.249 ns …  14.602 μs  ┊ GC (min … max): 0.00% … 98.27%
 Time  (median):     213.452 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   212.939 ns ± 397.277 ns  ┊ GC (mean ± σ):  6.39% ±  3.39%
```

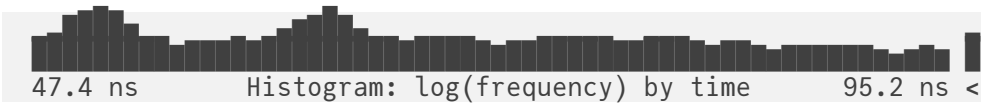


```
  158 ns           Histogram: frequency by time          298 ns <

 Memory estimate: 144 bytes, allocs estimate: 1.
```

```julia
with_terminal() do
    actions = collect(1:10)
    est = ActionValue(10, explorer = UCB())
    # @code_warntype sample_action(est, 10, actions)
    @benchmark sample_action($est, 10, $actions)
end
```

```
BenchmarkTools.Trial: 10000 samples with 990 evaluations.
 Range (min … max):  47.374 ns …   9.614 μs  ┊ GC (min … max): 0.00% … 99.10%
 Time  (median):     52.424 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   60.845 ns ± 162.857 ns  ┊ GC (mean ± σ):  4.61% ±  1.72%
```




```
Memory estimate: 32 bytes, allocs estimate: 1.
```

```
with_terminal() do
    actions = collect(1:10)
    est = GradientReward(10)
    # @code_warntype sample_action(est, 10, actions)
    @benchmark sample_action($est, 10, $actions)
end
```

```
update_estimator! (generic function with 1 method)
```

```
function update_estimator!(est::GradientReward{T}, a::Integer, r::T, step::Integer)
where T <: AbstractFloat
    (H, expH, πvec, α, R̄, r_avg) = (est.H, est.expH, est.πvec, est.α, est.R̄,
est.update)
    rdiff = r - R̄
    c = α*rdiff
    for i in eachindex(H)
        H[i] += c*(i == a ? (one(T) - πvec[i]) : -πvec[i])
    end
    #update R̄ with desired method
    updatecoef(::SampleAverage) = one(T)/step
    updatecoef(r_avg::ConstantStep) = r_avg.α
    function updatecoef(r_avg::UnbiasedConstantStep)
        r_avg.o = r_avg.o + r_avg.α*(one(T) - r_avg.o)
        r_avg.α/r_avg.o
    end
    est.R̄ = R̄ + updatecoef(r_avg)*rdiff

    #update expH and π_vec
    expH .= exp.(H)
    πvec .= calc_πvec(expH)
end
```

```
update_estimator! (generic function with 2 methods)
```

```
function update_estimator!(est::ActionValue{T}, a::Integer, r::T, i::Integer) where T
<: AbstractFloat
    (N, Q, q_avg) = (est.N, est.Q, est.q_avg)
    N[a] += one(T)
    updatecoef(q_avg::SampleAverage) = one(T) / N[a]
    updatecoef(q_avg::ConstantStep) = q_avg.α
    function updatecoef(r_avg::UnbiasedConstantStep)
        r_avg.o = r_avg.o + r_avg.α*(one(T) - r_avg.o)
        r_avg.α/r_avg.o
    end
    c = updatecoef(q_avg)
    Q[a] += c*(r - Q[a])
end
```

```
BenchmarkTools.Trial: 10000 samples with 999 evaluations.
 Range (min … max):  11.211 ns … 117.518 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     11.411 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   11.717 ns ±   2.117 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%

 11.2 ns       Histogram: log(frequency) by time      21.3 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

```
with_terminal() do
    actions = collect(1:10)
    est = ActionValue(10)
    # @code_warntype update_estimator!(est, 5, 1.0, 10)
    @benchmark update_estimator!($est, 5, 1.0, 10)
end
```

```
BenchmarkTools.Trial: 10000 samples with 750 evaluations.
 Range (min … max):  168.400 ns …  15.811 μs  ┊ GC (min … max): 0.00% … 98.25%
 Time  (median):     223.733 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   225.637 ns ± 403.796 ns  ┊ GC (mean ± σ):  5.86% ±  3.24%

 168 ns           Histogram: frequency by time          314 ns <

 Memory estimate: 144 bytes, allocs estimate: 1.
```

```
with_terminal() do
    actions = collect(1:10)
    est = GradientReward(10)
    # @code_warntype update_estimator!(est, 5, 1.0, 10)
    @benchmark update_estimator!($est, 5, 1.0, 10)
end
```

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  225.400 μs …   8.291 ms  ┊ GC (min … max): 0.00% … 95.45%
 Time  (median):     324.200 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   328.957 μs ± 126.148 μs  ┊ GC (mean ± σ):  0.73% ±  1.87%
```


```
Memory estimate: 40.89 KiB, allocs estimate: 7.
```

```
with_terminal() do
    bandit = create_bandit(k)
    estimator = ActionValue(k, q_avg=ConstantStep())
    @benchmark run_bandit($bandit, $estimator)
end
```

```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  275.800 μs …   8.126 ms  ┊ GC (min … max): 0.00% … 96.06%
 Time  (median):     292.600 μs               ┊ GC (median):    0.00%
 Time  (mean ± σ):   297.719 μs ± 139.240 μs  ┊ GC (mean ± σ):  0.91% ±  1.90%
```


```
Memory estimate: 41.31 KiB, allocs estimate: 10.
```

```
with_terminal() do
    bandit = create_bandit(k)
    @benchmark simple_algorithm($bandit, $k, 0.1)
end
```

```
average_stationary_runs (generic function with 1 method)
```

```
function average_stationary_runs(k, algorithm; steps = 1000, n = 2000, offset = 0.0)



        est = algorithm(k)
        runs[i] = run_bandit(qs, est, steps = steps)
    end
    map(i -> mapreduce(a -> a[i], (a, b) -> a .+ b, runs)./n, (:step_reward,
:step_reward_ideal, :optimalstep, :cum_reward, :cum_reward_ideal, :optimalaction_pct))
end
```

```
average_stationary_runs_cum_reward (generic function with 1 method)
```

```
function average_stationary_runs_cum_reward(k, algorithm; steps = 1000, n = 2000,
offset::T = 0.0) where T <: AbstractFloat
    r_step = Atomic{T}(zero(T))
    r_step_ideal = Atomic{T}(zero(T))
    @threads for i in 1:n
        qs = create_bandit(k, offset=offset)
        est = algorithm(k)
        rewards = run_bandit_cumreward(qs, est, steps = steps)
        atomic_add!(r_step, rewards[1])
        atomic_add!(r_step_ideal, rewards[2])
    end
    (r_step[]/n, r_step_ideal[]/n)
end
```

```
average_nonstationary_runs (generic function with 2 methods)
```

```
function average_nonstationary_runs(k, algorithm; steps = 10000, n = 2000,
qinit::T=0.0) where T<:AbstractFloat
    runs = Vector{NamedTuple}(undef, n)
    @threads for i in 1:n
        qs = ones(T, k) .* qinit
        est = algorithm(k)
        runs[i] = run_bandit(qs, est, steps = steps, σ=0.01)
    end
    map(i -> mapreduce(a -> a[i], (a, b) -> a .+ b, runs)./n, (:step_reward,
:step_reward_ideal, :optimalstep, :cum_reward, :cum_reward_ideal, :optimalaction_pct))
end
```

```
average_nonstationary_runs_cum_reward (generic function with 1 method)
```

```
function average_nonstationary_runs_cum_reward(k, algorithm; steps = 10000, n = 2000,
qinit::T = 0.0) where T<:AbstractFloat
    r_step = Atomic{T}(zero(T))
    r_step_ideal = Atomic{T}(zero(T))
    @threads for i in 1:n
        qs = ones(T, k) .* qinit
        est = algorithm(k)
        rewards = run_bandit_cumreward(qs, est, steps = steps, σ=0.01, cumstart =
floor(Int64, steps/2) + 1)
        atomic_add!(r_step, rewards[1])
        atomic_add!(r_step_ideal, rewards[2])
    end
    (r_step[]/n, r_step_ideal[]/n)
end
```

```
BenchmarkTools.Trial: 159 samples with 1 evaluation.
 Range (min … max):  30.973 ms …  32.800 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     31.254 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   31.441 ms ± 394.064 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```


```
Memory estimate: 1.70 MiB, allocs estimate: 14205.
```

```
with_terminal() do
    @benchmark average_stationary_runs_cum_reward(k, k -> ActionValue(k))
end
```

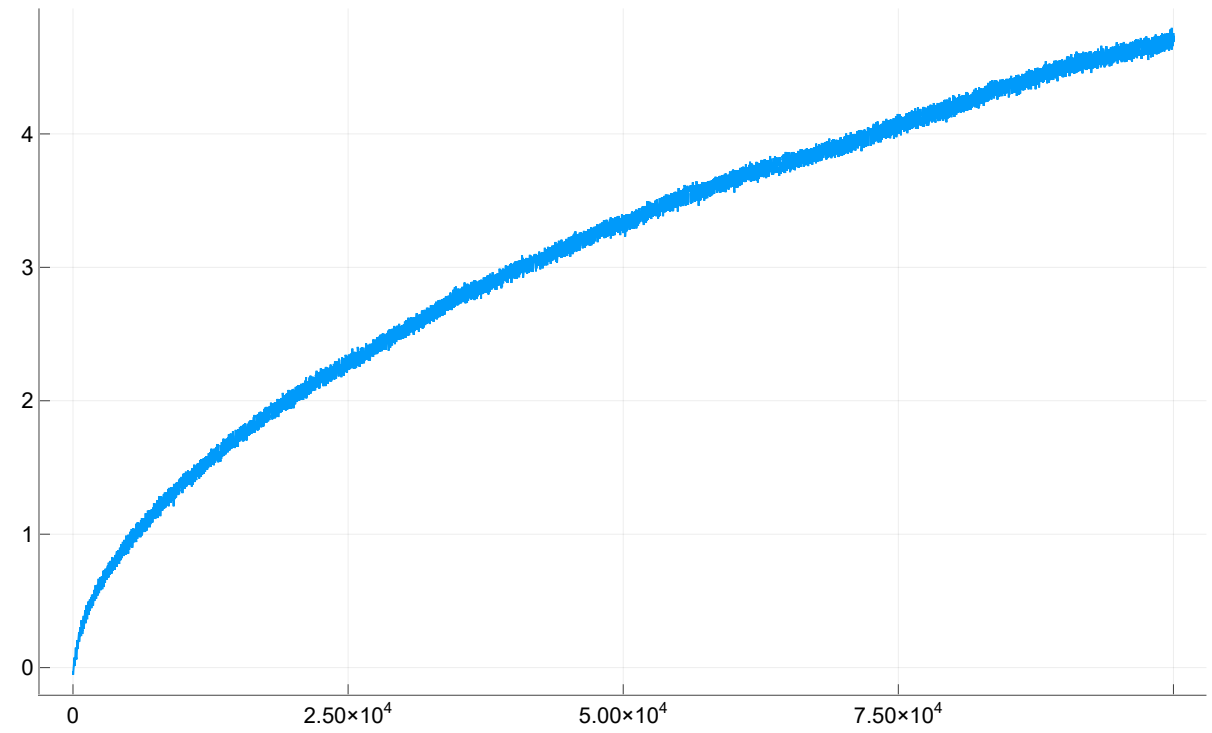
```
BenchmarkTools.Trial: 159 samples with 1 evaluation.
 Range (min … max):  29.148 ms …  33.231 ms  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     31.912 ms               ┊ GC (median):    0.00%
 Time  (mean ± σ):   31.604 ms ± 903.803 μs  ┊ GC (mean ± σ):  0.00% ± 0.00%
```


```
Memory estimate: 1.24 MiB, allocs estimate: 14206.
```

```
with_terminal() do
#repeat benchmark with Float32 to see how much faster it is
    @benchmark average_stationary_runs_cum_reward(k, k -> ActionValue(k, Qinit =
0.0f0), offset = 0.0f0)
end
```

```
nonstationaryruns =
▸ ([-0.0298772, -0.00137096, -0.0145537, -0.0434356, -0.00296158, -0.00237583, 0.0282632, 0.
```

```
nonstationaryruns = average_nonstationary_runs(k, k -> ActionValue(k, explorer =
ϵ_Greedy(1/128), q_avg = ConstantStep()), steps = 100000)
```




```
plot(nonstationaryruns[1], legend = false)
```

```
gradient_est =
▸ GradientReward([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [1.0, 1.0, 1.0, 1.0, 1.0
```

```
gradient_est = GradientReward(k)
```

```
▸ (step_reward = [-0.845815, 0.69385, -0.870045, -1.82516, -0.871048, -0.700327, -0.77561, 0
```

```
run_bandit(create_bandit(k), gradient_est)
```

```
αlist = ▸ [0.025, 0.05, 0.1, 0.2, 0.4]
```

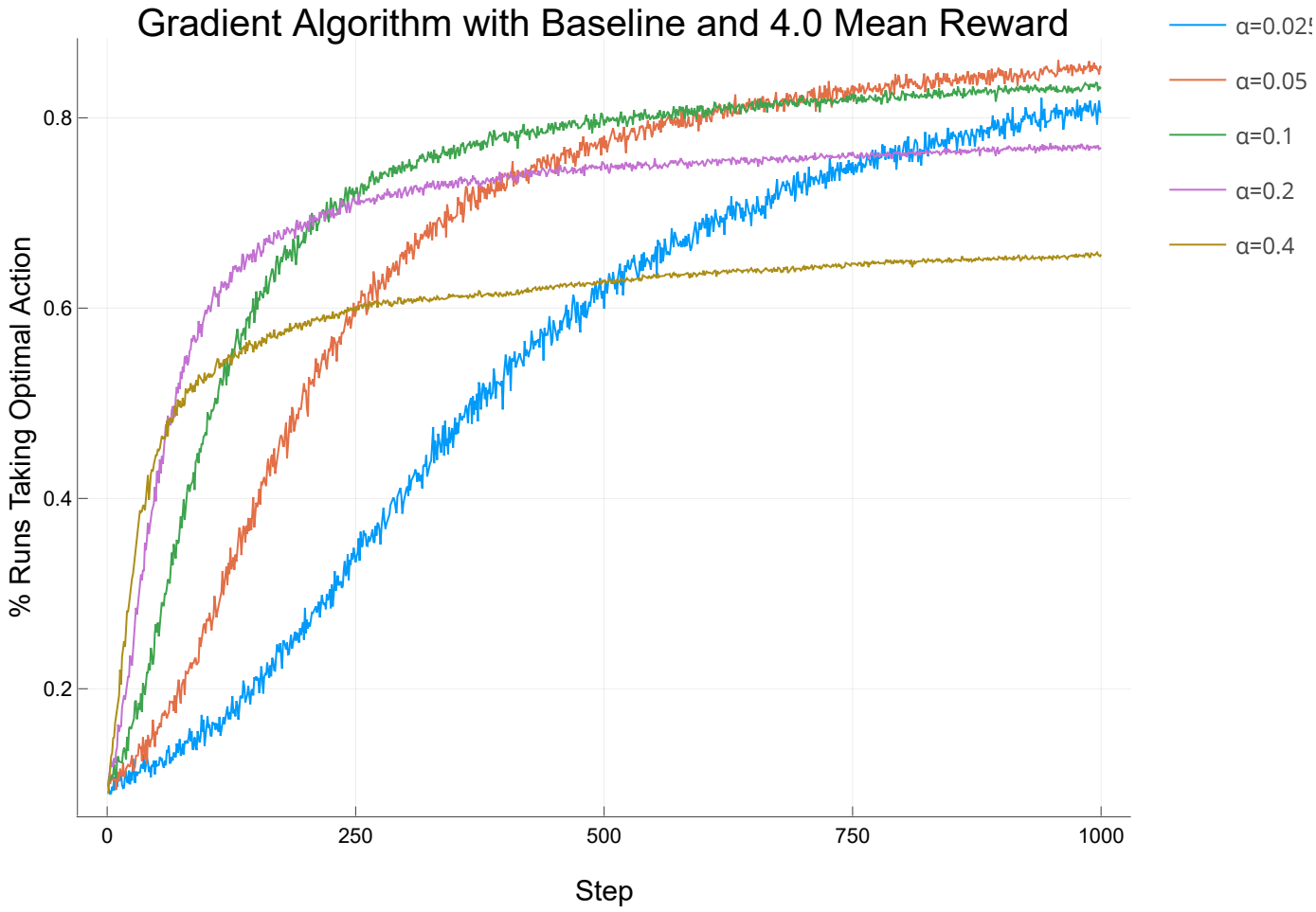
```
αlist = [0.025, 0.05, 0.1, 0.2, 0.4]
```

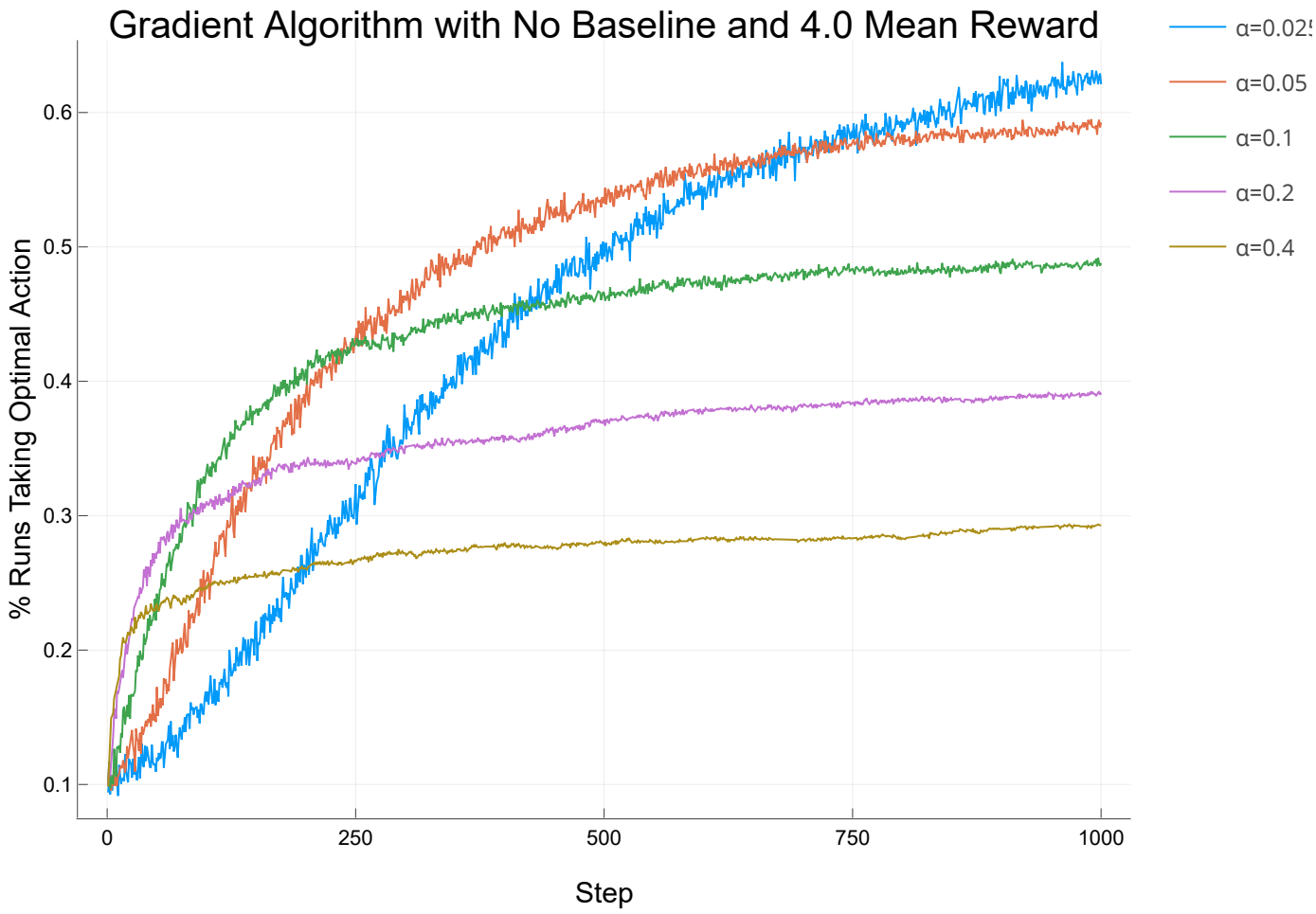
```
gradientruns_baseline =
▸ [([3.98575, 4.00436, 3.96155, 3.97902, 4.00884, ⋯ more ,5.44846], [5.52464, 5.54379, 5.51(
```

```
gradientruns_baseline = [average_stationary_runs(k, k -> GradientReward(k, α=α),
offset = 4.0) for α in αlist]
```

```
gradientruns_nobaseline =
▸ [([3.99706, 3.95122, 4.01578, 4.02285, 3.90947, ⋯ more ,5.32317], [5.54207, 5.52971, 5.55(
```

```
gradientruns_nobaseline = [average_stationary_runs(k, k -> GradientReward(k, α=α,
update = ConstantStep(0.0)), offset = 4.0) for α in αlist]
```


Gradient Algorithm with Baseline and 4.0 Mean Reward
% Runs Taking Optimal Action
0.2
0.4
0.6
0.8
0
250
500
750
1000
Step
α=0.025
α=0.05
α=0.1
α=0.2
α=0.4


Gradient Algorithm with No Baseline and 4.0 Mean Reward
% Runs Taking Optimal Action
0.1
0.2
0.3
0.4
0.5
0.6
0
250
500
750
1000
Step
α=0.025
α=0.05
α=0.1
α=0.2
α=0.4

Expercise 2.10 Suppose you face a 2-armed bandit task whose true action values change randomly from time step to time step. Specifically, suppose that, for any time step, the true values of actions 1 and 2 are respectively 10 and 20 with probability 0.5 (case A), and 90 and 80 with probability 0.5 (case B). If you are not able to tell which case you face at any step, what is the best expected reward you can achieve and how should you behave to achieve it? Now suppose that on each step you are told whether you are facing case A or case B (although you still don't know the true action values). This is an associative search task. What is the best expected reward you can achieve in this task, and how should you behave to achieve it?

For the case in which we do not know which case we are facing, we can calculate the expected reward for each action across all cases.

$$E[R_1] = 0.5 \times 10 + 0.5 \times 90 = 50$$

$$E[R_2] = 0.5 \times 20 + 0.5 \times 80 = 50$$

Since the expected reward of each action is equal, the best we can do is pick randomly which will have an expected reward of 50.

For the case in which we know if we are in case A or case B, we now can select the best action for each case which has a value of 20 (action 2) for case A and 90 (action 1) for case B. However, we have a 50% probability of facing each case so our overall expected reward is.

$$E[R] = 20 \times 0.5 + 90 \times 0.5 = 55$$

Parameter Studies

```
print_power2 (generic function with 1 method)
```

```
function print_power2(n)
    if abs(n) > 7
        "2^$n"
    elseif n < 0
        "1/$(2^-n)"
    else
        "$(2^n)"
    end
end
```

```
get_param_list (generic function with 1 method)
```

```
function get_param_list(n1::Integer, n2::Integer; base::T = 2.0) where
T<:AbstractFloat
    nlist = collect(n1:n2)
    plist = base .^nlist
    # namelist = print_power2.(nlist)
    return plist, nlist
end
```

```
stationary_param_search (generic function with 1 method)
```

```
function stationary_param_search(algorithm, n1, n2; base::T = 2.0) where
T<:AbstractFloat
    plist, nlist = get_param_list(n1, n2, base = base)
    runs = Vector{Tuple{T, T}}(undef, length(plist))
    cum_rewards = Vector{T}(undef, length(plist))
    cum_rewards_ideal = similar(cum_rewards)
    for i in eachindex(runs)
        run = average_stationary_runs_cum_reward(k, k -> algorithm(plist[i], k))
        cum_rewards[i] = run[1]
        cum_rewards_ideal[i] = run[2]
    end
    plist, nlist, (cum_rewards, cum_rewards_ideal)
end
```

```
plotstationaryparamsearch (generic function with 1 method)
```

```
function plotstationaryparamsearch(param_search, pname; addplot = false, ideal =
false)
    (plist, nlist, runrewards) = param_search
    pltfunc = addplot ? plot! : plot
    rewardindex = ideal ? 2 : 1
    pltfunc(plist, runrewards[rewardindex], xaxis = :log, xticks = (plist,
print_power2.(nlist)), lab = pname, yaxis = ("Average Reward over first 1000 steps",
[0.5, 1.6]), size = (650, 450))
end
```

```
run_or_load (generic function with 1 method)
```

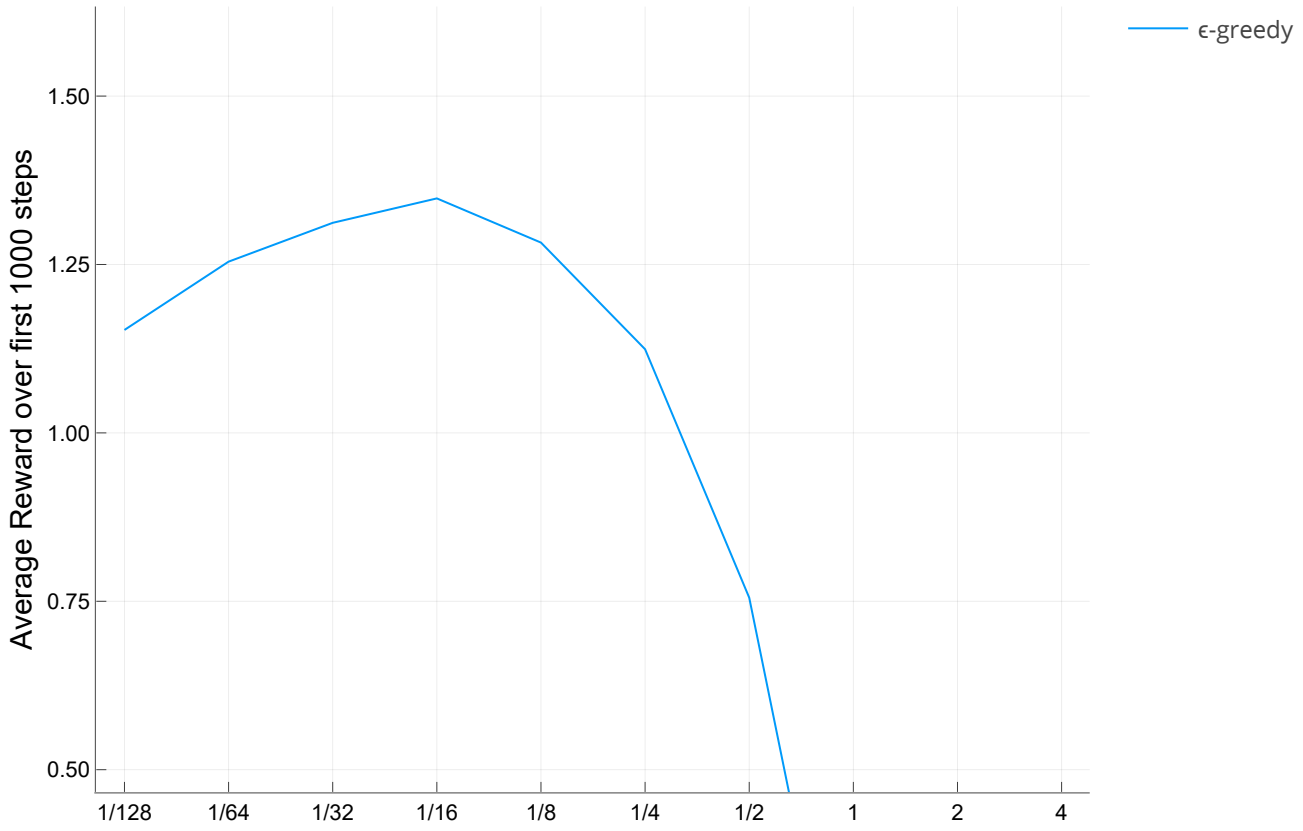
```
function run_or_load(varname::String, operation::Function)
    if !isfile("$varname.jld2")
        data = operation()
        jldsave("$varname.jld2"; data)
    else
        data = read(jldopen("$varname.jld2"), "data")
    end
    return data
end
```

```
save_data (generic function with 1 method)
```

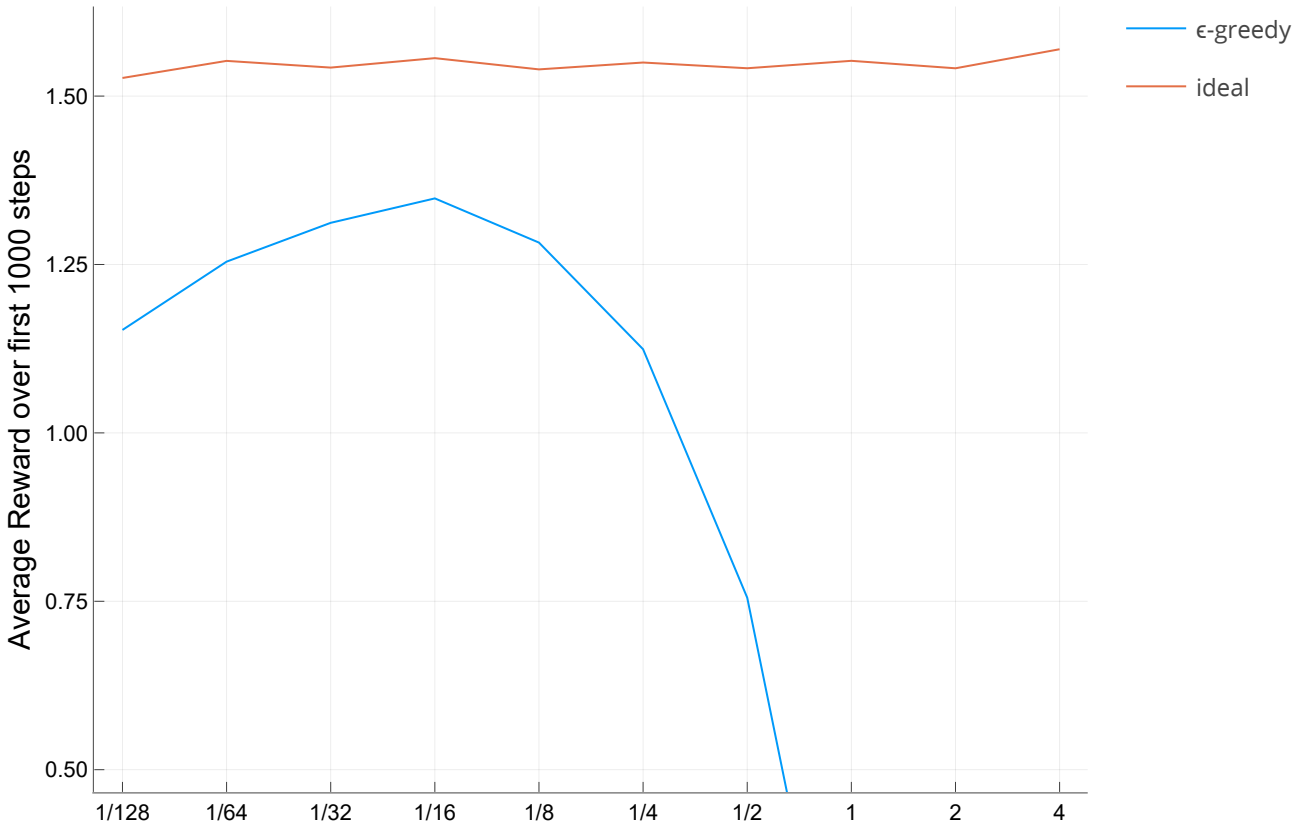
```
save_data(varname, data) = jldsave("$varname.jld2"; data)
```

```
ϵ_greedy_stationary_param_search =
▸ ([0.0078125, 0.015625, 0.03125, 0.0625, 0.125, 0.25, 0.5, 1.0, 2.0, 4.0], [-7, -6, -5, -4,
```

```
ϵ_greedy_stationary_param_search = run_or_load("ϵ_greedy_stationary_param_search", ()->
stationary_param_search((p, k) -> ActionValue(k, explorer = ϵ_Greedy(p)), -7, 2))
```



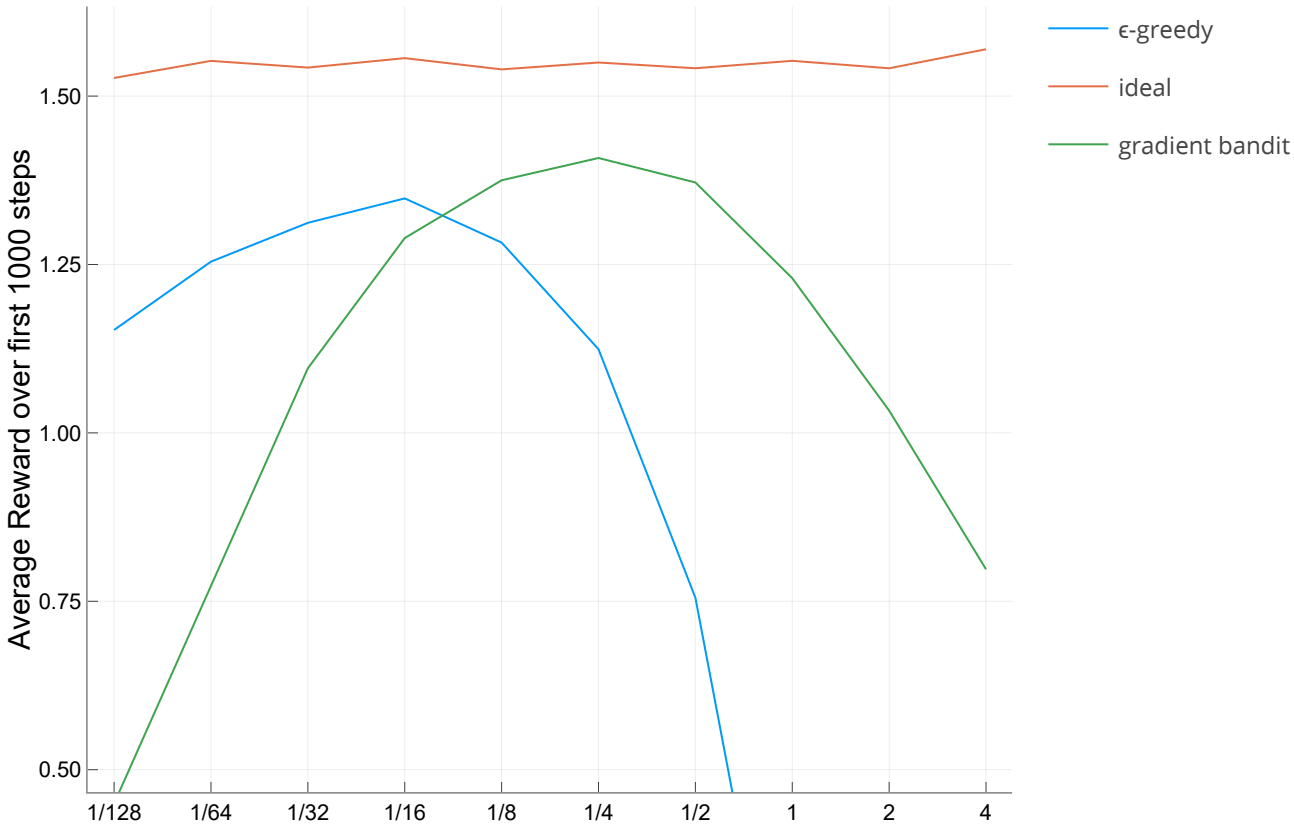

```
plotstationaryparamsearch(ϵ_greedy_stationary_param_search, "ϵ-greedy")
```




```
plotstationaryparamsearch(ϵ_greedy_stationary_param_search, "ideal", addplot = true, ideal = true)
```

```
gradient_stationary_param_search =
([0.0078125, 0.015625, 0.03125, 0.0625, 0.125, 0.25, 0.5, 1.0, 2.0, 4.0], [-7, -6, -5, -4,
```

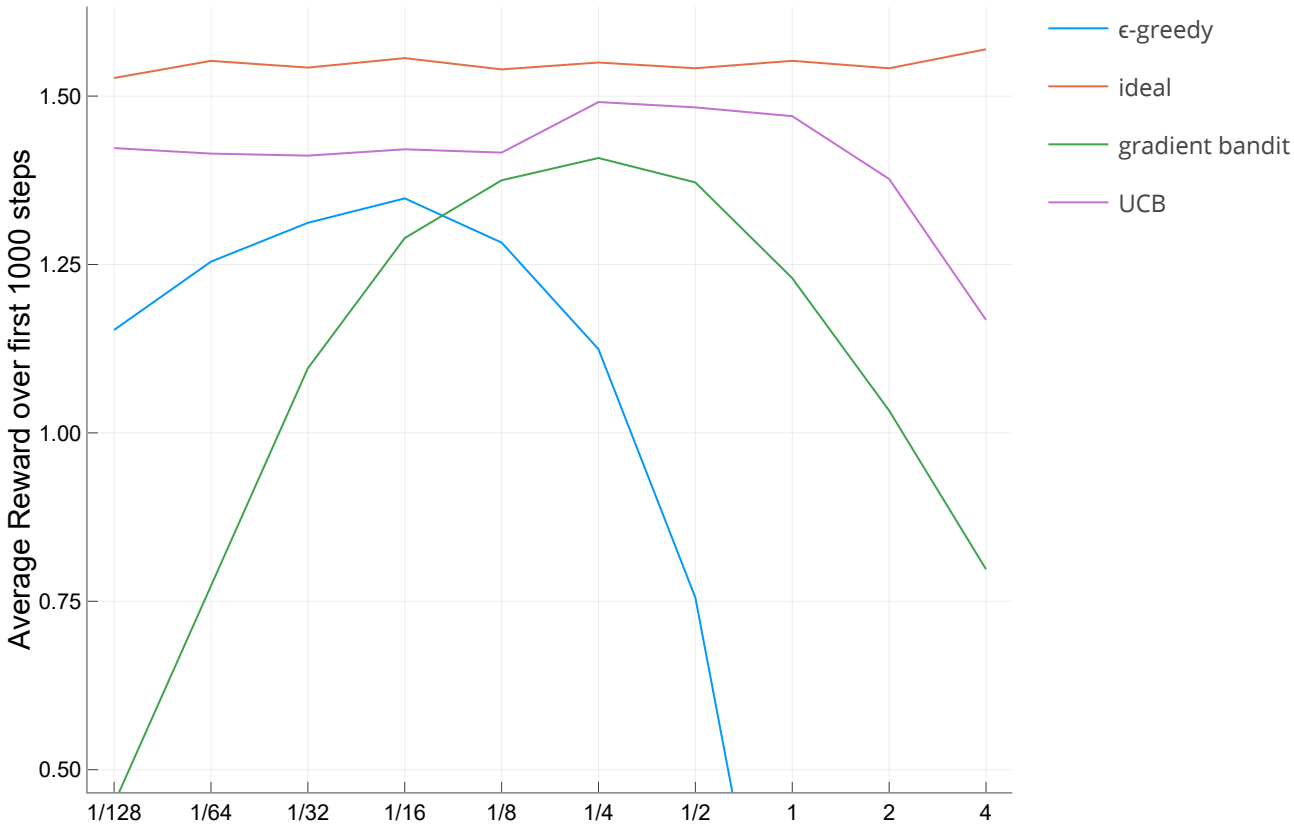
```
gradient_stationary_param_search = run_or_load("gradient_stationary_param_search", ()->stationary_param_search((p, k) ->
    GradientReward(k, α=p), -7, 2))
```




```
plotstationaryparamsearch(gradient_stationary_param_search, "gradient bandit", addplot = true)
```

```
UCB_stationary_param_search =
▸ ([0.0078125, 0.015625, 0.03125, 0.0625, 0.125, 0.25, 0.5, 1.0, 2.0, 4.0], [-7, -6, -5, -4,
```

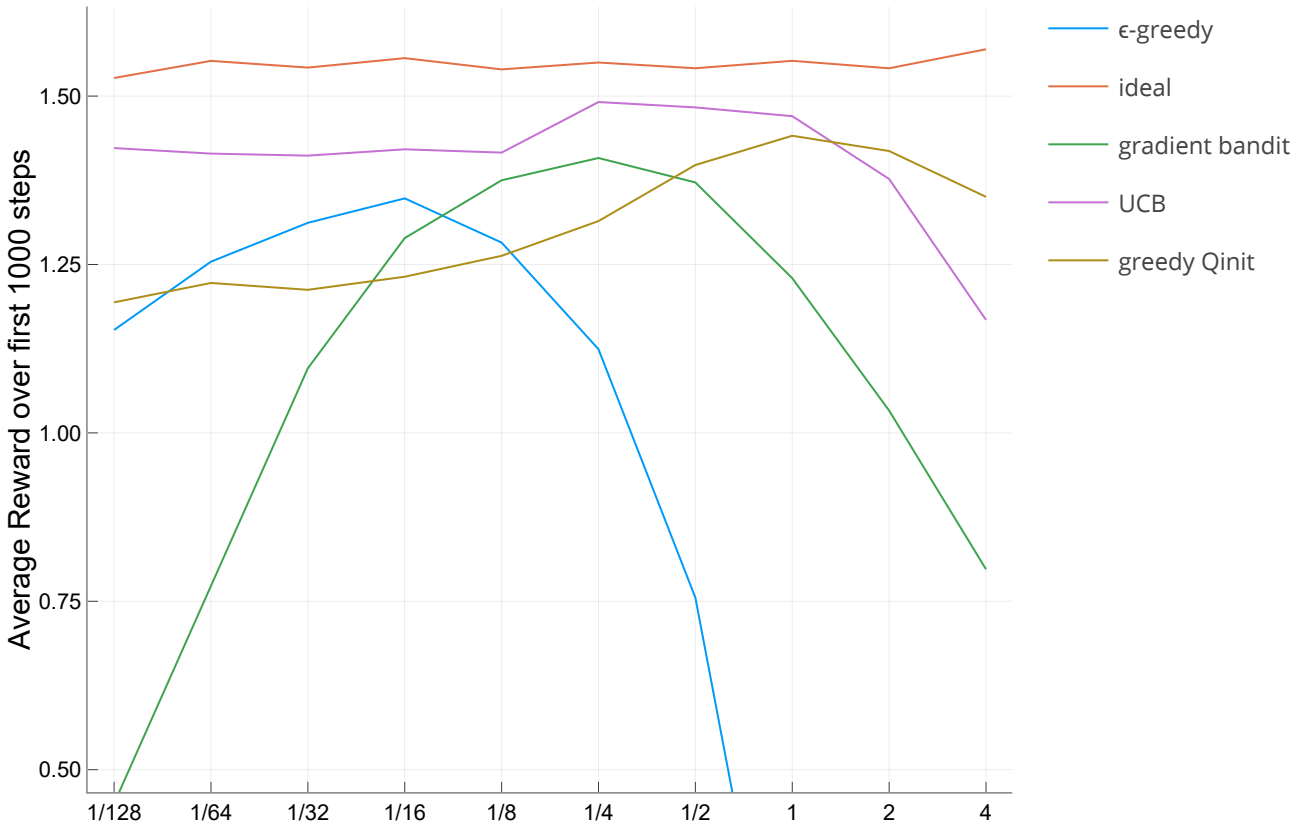
```
UCB_stationary_param_search = run_or_load("UCB_stationary_param_search", ()->stationary_param_search((p, k) -> ActionValue(k, explorer = UCB(p)), -7, 2))
```




```
plotstationaryparamsearch(UCB_stationary_param_search, "UCB", addplot = true)
```

```
optin_stationary_param_search =
▸ ([0.0078125, 0.015625, 0.03125, 0.0625, 0.125, 0.25, 0.5, 1.0, 2.0, 4.0], [-7, -6, -5, -4,
```

```
optin_stationary_param_search = run_or_load("optin_stationary_param_search", ()->stationary_param_search((p, k) -> ActionValue(k, Qinit = p, explorer = ϵ_Greedy(0.0), q_avg = ConstantStep()), -7, 2))
```




```
plotstationaryparamsearch(optin_stationary_param_search, "greedy Qinit", addplot = true)
```

Exercise 2.11 (programming) Make a figure analogous to Figure 2.6 for the nonstionary case outlined in Exercise 2.5. Include the constant-step-size ϵ-greedy algorithm with α=0.1. Use runs of 200,000 steps and, as a performance measure for each algorithm and parameter setting, use the average reward over the last 100,000 steps.

```
numsteps = 200000
```

```
numsteps = 200000
```

```
param_search (generic function with 1 method)
```

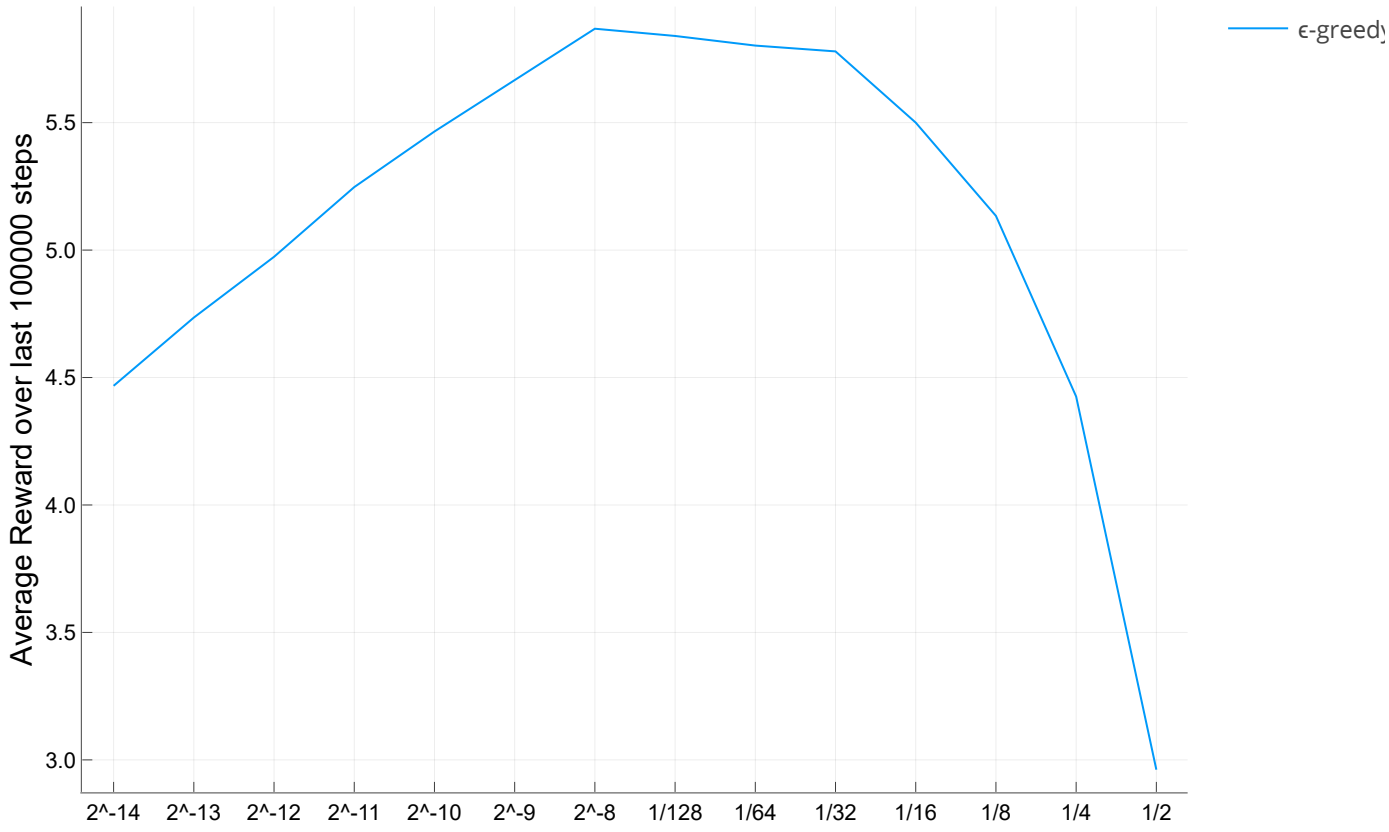
```
function param_search(run_function, algorithm, n1, n2; steps = 1000, base::T = 2.0)
where T<:AbstractFloat
    (plist, nlist) = get_param_list(n1, n2, base=base)
    cum_rewards = Vector{T}(undef, length(plist))
    cum_rewards_ideal = similar(cum_rewards)
    for i in eachindex(plist)
        run = run_function(k, k -> algorithm(plist[i], k), steps = steps)
        cum_rewards[i] = run[1]
        cum_rewards_ideal[i] = run[2]
    end
    (plist, nlist, (cum_rewards, cum_rewards_ideal), steps)
end
```

```
plotnonstationaryparamsearch (generic function with 1 method)
```

```
function plotnonstationaryparamsearch(param_search, pname; addplot = false, ideal = false)
    (plist, nlist, runrewards, steps) = param_search
    pltfunc = addplot ? plot! : plot
    rewardindex = ideal ? 2 : 1
    pltfunc(plist, runrewards[rewardindex], xaxis = :log, xticks = (plist, print_power2.(nlist)), lab = pname, yaxis = ("Average Reward over last $(floor(Int64, steps/2)) steps",), size = (700, 450))
end
```

```
ϵ_greedy_nonstationary_paramsearch =
([6.10352e-5, 0.00012207, 0.000244141, 0.000488281, 0.000976562, 0.00195312, 0.00390625, 0
```

```
ϵ_greedy_nonstationary_paramsearch = run_or_load("ϵ_greedy_nonstationary_paramsearch", () -> param_search(average_nonstationary_runs_cum_reward, (p, k) -> ActionValue(k, explorer = ϵ_Greedy(p), q_avg = ConstantStep()), -14, -1, steps = numsteps))
```

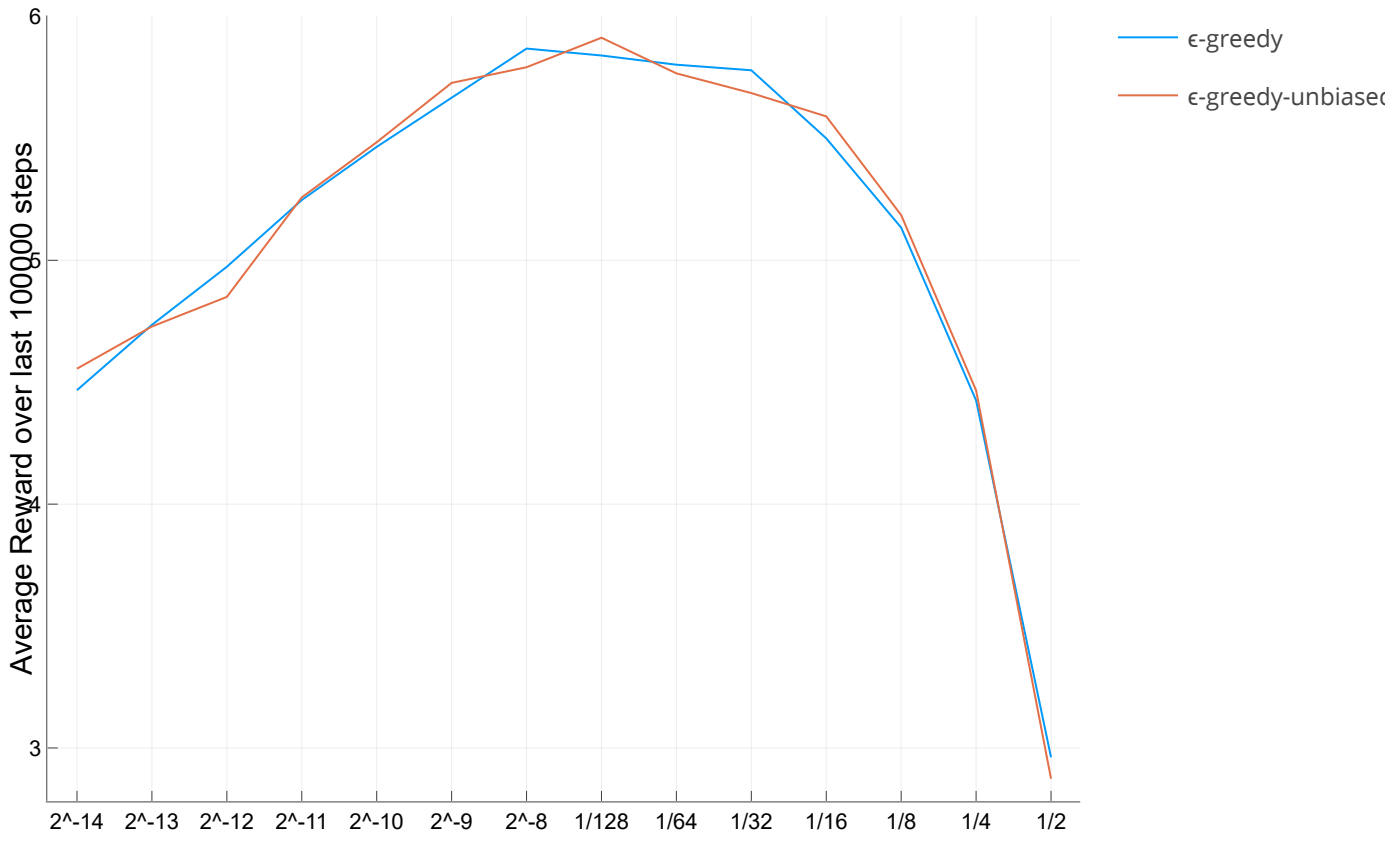



```
plotnonstationaryparamsearch(ϵ_greedy_nonstationary_paramsearch, "ϵ-greedy")
```

```
ϵ_greedy_unbiased_nonstationary_paramsearch =
```

▸ ([6.10352e-5, 0.00012207, 0.000244141, 0.000488281, 0.000976562, 0.00195312, 0.00390625, 0

```
ϵ_greedy_unbiased_nonstationary_paramsearch = run_or_load("ϵ_greedy_unbiased_nonstationary_paramsearch", () -> param_search(average_nonstationary_runs_cum_reward, (p, k) -> ActionValue(k, explorer = ϵ_Greedy(p), q_avg = UnbiasedConstantStep()), -14, -1, steps = numsteps))
```

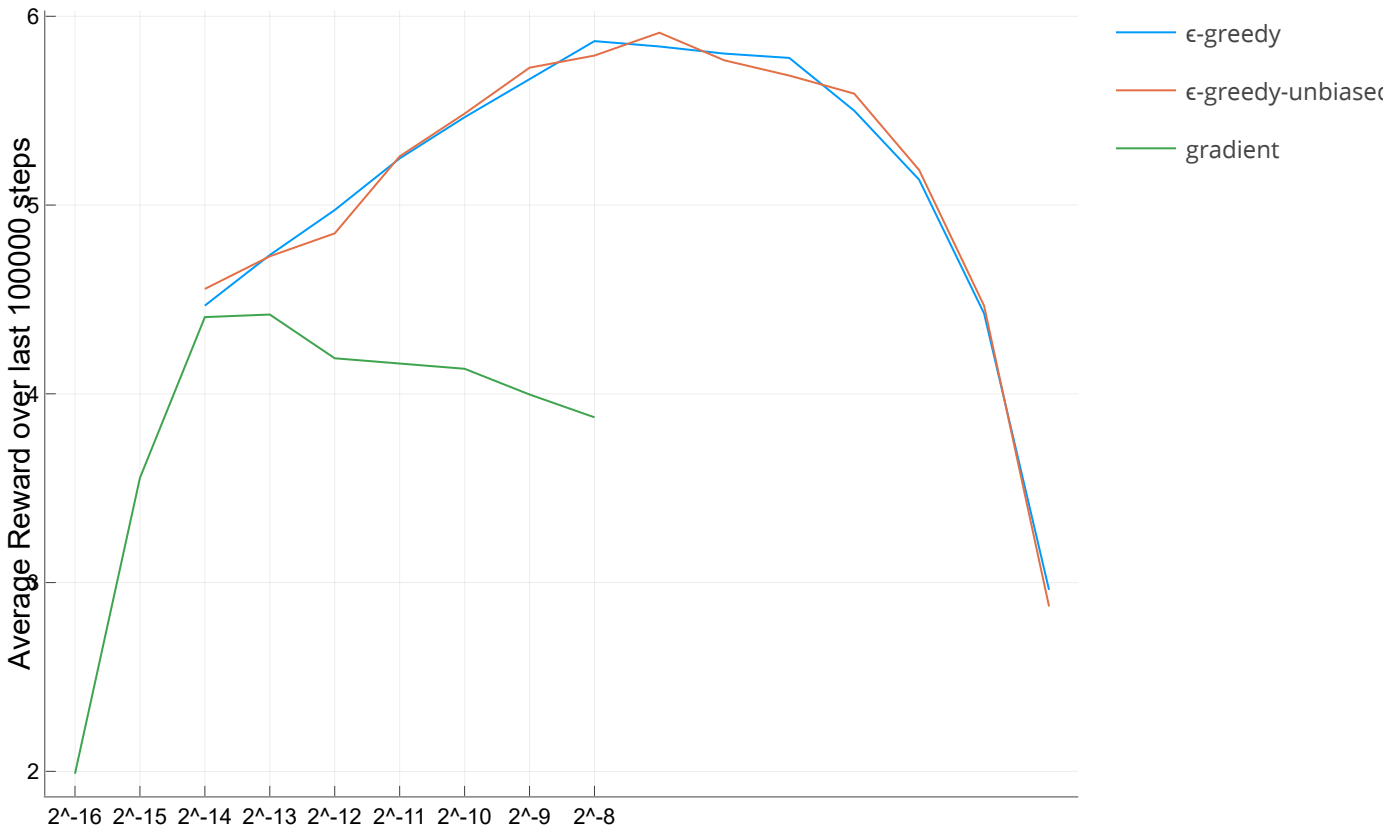



```
plotnonstationaryparamsearch(ϵ_greedy_unbiased_nonstationary_paramsearch, "ϵ-greedy-unbiased", addplot=true)
```

```
gradient_nonstationary_paramsearch =
```

▸ ([1.52588e-5, 3.05176e-5, 6.10352e-5, 0.00012207, 0.000244141, 0.000488281, 0.000976562, 0

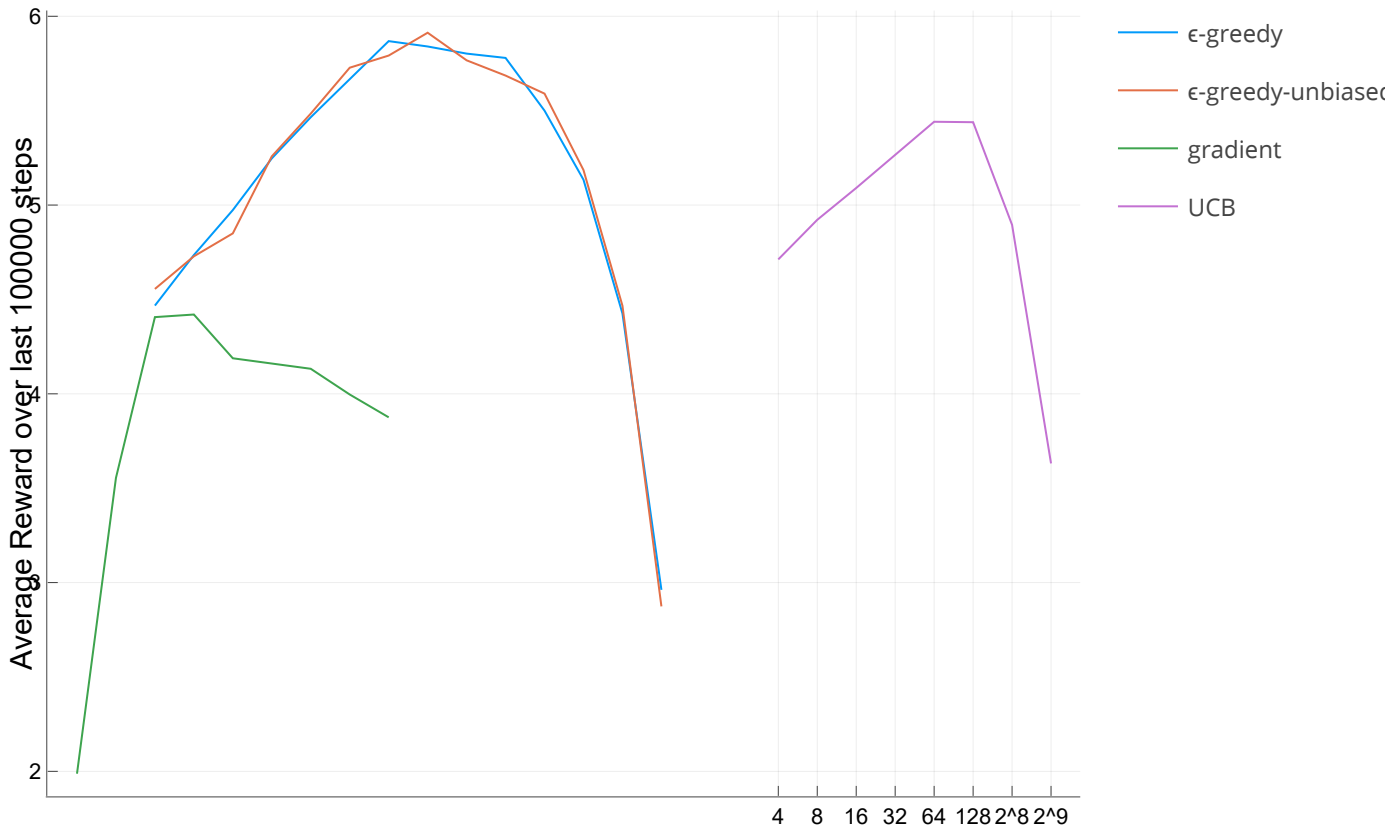
```
gradient_nonstationary_paramsearch = run_or_load("gradient_nonstationary_paramsearch", () -> param_search(average_nonstationary_runs_cum_reward, (p, k) -> GradientReward(k, α=p, update = ConstantStep()), -16, -8, steps = numsteps))
```




```
plotnonstationaryparamsearch(gradient_nonstationary_paramsearch, "gradient",
addplot=true)
```

```
UCB_nonstationary_paramsearch =
([4.0, 8.0, 16.0, 32.0, 64.0, 128.0, 256.0, 512.0], [2, 3, 4, 5, 6, 7, 8, 9], ([4.71209, 4.9
```

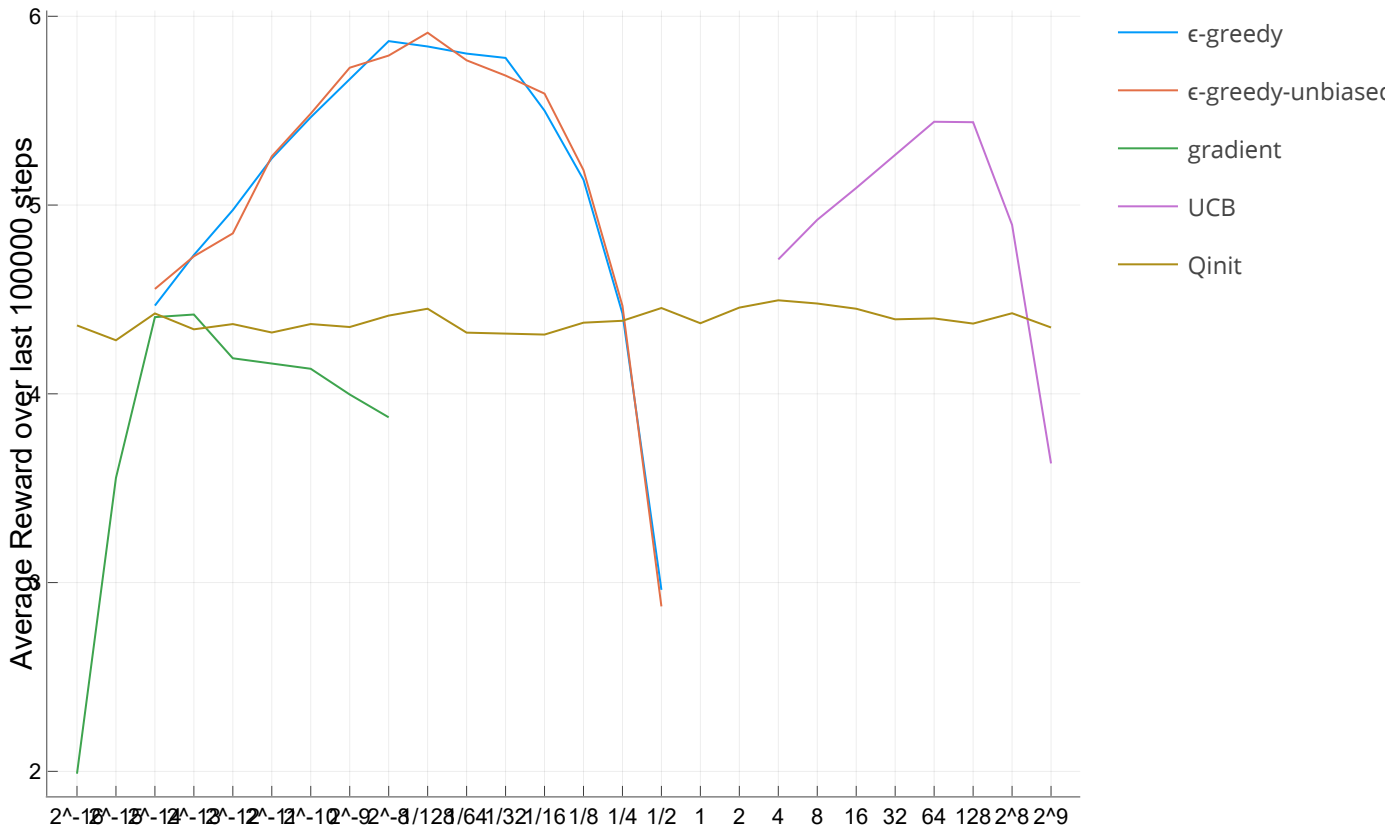
```
UCB_nonstationary_paramsearch = run_or_load("UCB_nonstationary_paramsearch", () ->
param_search(average_nonstationary_runs_cum_reward, (p, k) -> ActionValue(k, explorer
= UCB(p), q_avg = ConstantStep()), 2, 9, steps = numsteps))
```



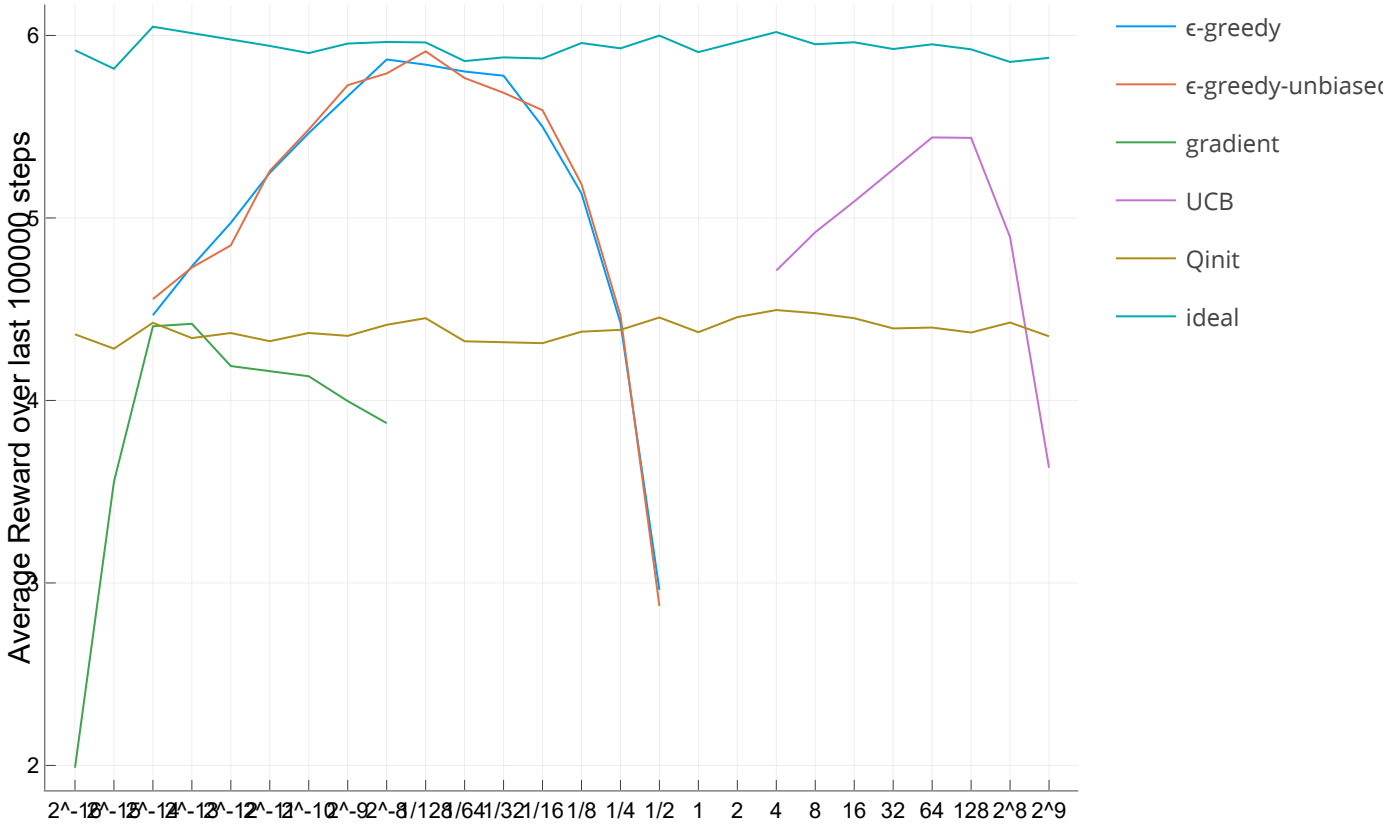

```
plotnonstationaryparamsearch(UCB_nonstationary_paramsearch, "UCB", addplot=true)
```

```
optinit_nonstationary_paramsearch =
▸ ([1.52588e-5, 3.05176e-5, 6.10352e-5, 0.00012207, 0.000244141, 0.000488281, 0.000976562, 0
```

```
optinit_nonstationary_paramsearch = run_or_load("optinit_nonstationary_paramsearch",
() -> param_search(average_nonstationary_runs_cum_reward, (p, k) -> ActionValue(k,
Qinit = p, explorer = ϵ_Greedy(0.0), q_avg = ConstantStep()), -16, 9, steps =
numsteps))
```




```
plotnonstationaryparamsearch(optinit_nonstationary_paramsearch, "Qinit", addplot=true)
```




```
plotnonstationaryparamsearch(optinit_nonstationary_paramsearch, "ideal",
addplot=true, ideal=true)
```

Recreation of Figure 2.6 for the non-stationary case. In all cases where average values are updated, a constant step size of α=0.1 is used. The parameters that vary along the x-axis correspond to the algorithms as follows: ϵ-greedy -> ϵ, gradient -> α, UCB -> c, Qinit -> initial Q estimate. Finally the ideal reward if the optimal action is selected each step is also shown. Unlike in the stationary case, the ϵ-greedy method with α=0.1 for updating the Q values performs the best at a very small ϵ value of 2^{-8} . The UCB is a close second but requires a very large c value of 64 compared to ~1 for the stationary case in which it was the best performer.