# 强化学习基础篇（二十四）价值迭代之gamblers问题

该问题基于《Reinforcement Learning: An Introduction》在第四章的例4.4 gamblers问题.

## 1、 问题描述

一个Gamblers下注猜测一系列抛硬币实验的结果。如果硬币正面朝上，他获得这一次下注的钱；如果背面朝上则失去这一次下注的钱。这个游戏在Gamblers达到获利目标100￥或者全部输光时结束。每一次抛硬币，Gamblers必须从他的资金中选取一个整数来下注。可以将这个问题表示为一个非折扣的分幕式有限MDP。

状态为Gamblers的资金$s \in \{1, 2, 3, ..., 99\}$，动作为Gamblers下注的金额$a \in \{0, 1, ..., min(s, 100 - s)\}$。收益一般情况下均为0，只有Gamblers达到获利100￥的终止状态时为+1。状态价值函数给出了在每个状态下Gamblers获胜的概率。这里的策略是资金到赌注的映射。最优策略将会最大化这个概率。

令$p_h$为抛硬币正面向上的概率。如果$p_h$已知，那么整个问题可以由价值迭代或其他类似的算法解决。

## 2、初步分析

该问题可以视为一个无折扣的有限MDP问题：

- 状态空间：赌徒的赌资：$s \in \{1, 2, 3, ..., 99\}$
- 行为：下注金额：$a \in \{0, 1, ..., min(s, 100 - s)\}$(下注金额最多不会超过距离获胜的差距)
- 收益：赢到100￥：+1，其余：0
- 状态价值：状态s下获胜的概率。
- 策略：当前持有赌资的下注金额。

问题解决需要使用价值迭代算法：

**Value Iteration, for estimating $\pi \approx \pi_*$**

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $\quad v \leftarrow V(s)$
| $\quad V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$
| $\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\big[r + \gamma V(s')\big]$

从价值迭代算法中可以看到，其与策略评估不同之处在于：算法仅执行一遍价值评估，通过遍历行为空间，选取最大的状态价值赋值。

## 3、代码实现

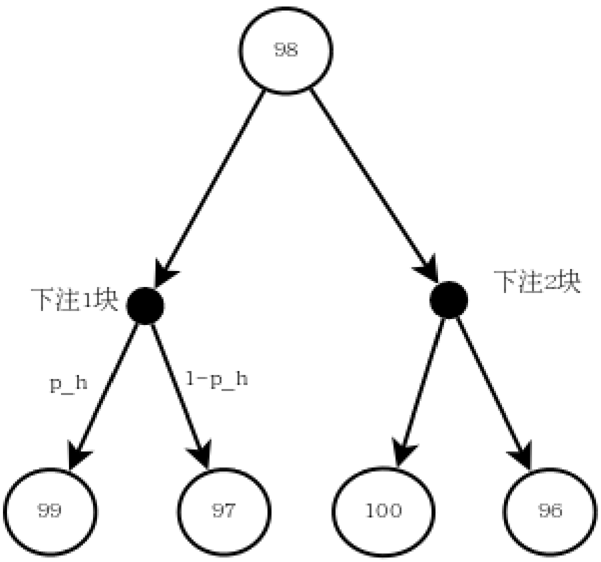
## 导入库与定义超参数

```python
import matplotlib
import matplotlib.pyplot as plt
import numpy as np

# 目标
GOAL = 100

# 这里包括了0和100，仅仅是为了方便作图
STATES = np.arange(GOAL + 1)

# 硬币正面朝上的概率
HEAD_PROB = 0.4
```

# 价值更新

这里代码按照上面的算法实现，并完成结果作图。

- 策略迭代中，价值评估仅关注在当前状态S下**执行一个确定动作**后产生的状态S′，进而遍历S′产生状态价值之和决定V(S)。
- 价值迭代中，价值评估关注在当前状态S下，**执行全部动作空间**后产生的状态价值列表L(S′)，进而取L(S′)中的最大值来决定V(S)。

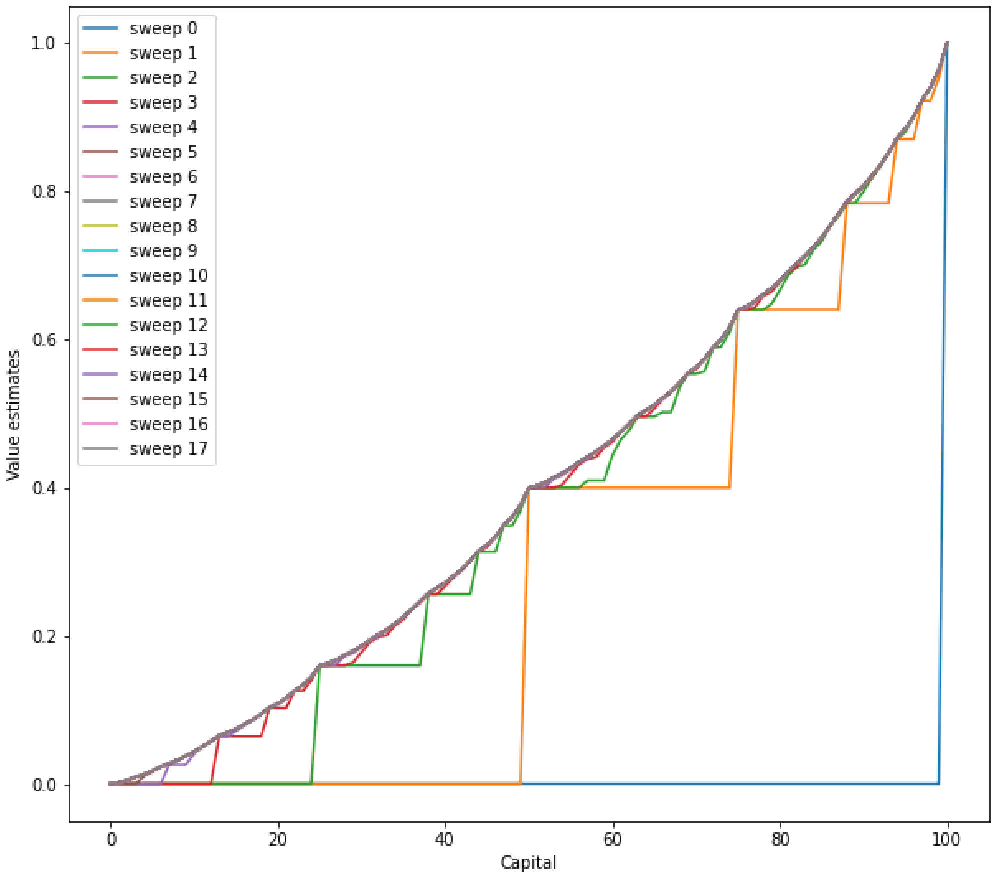


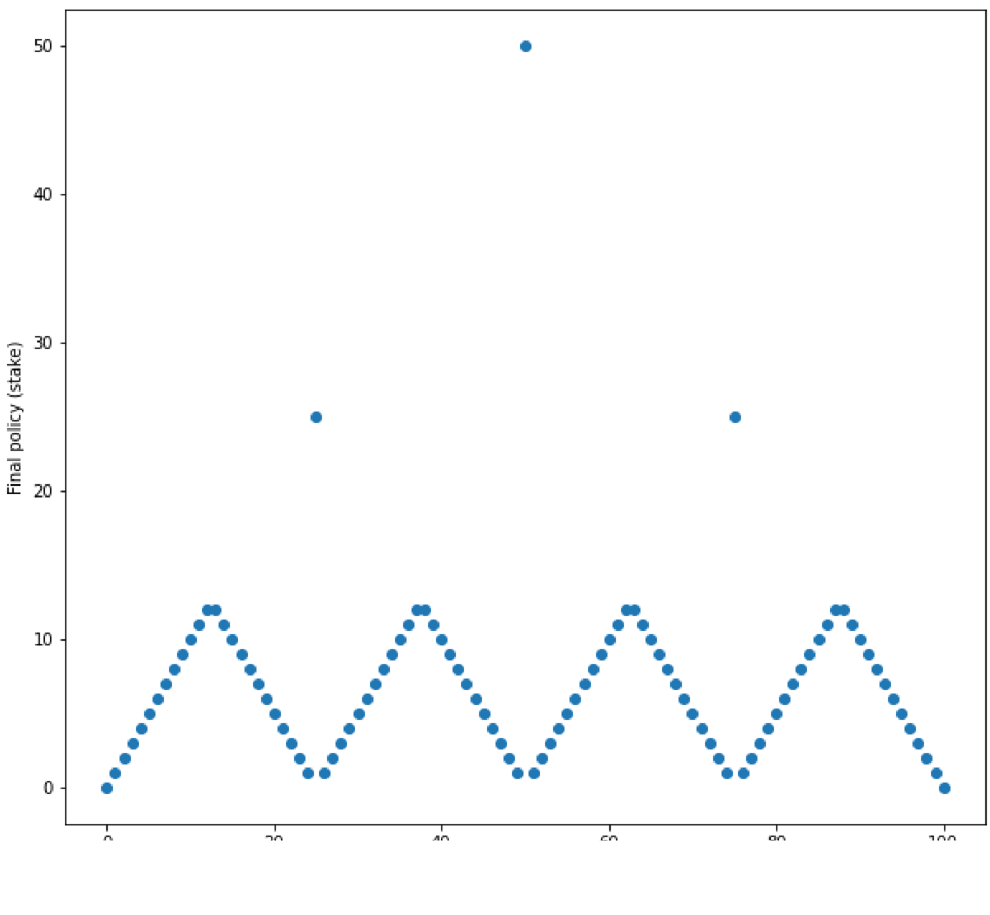

```python
def figure_4_3():
    # # state value即状态价值：记录状态s下获胜的概率。
    state_value = np.zeros(GOAL + 1)
    # goal状态下的获胜概率必然为1.0
    state_value[GOAL] = 1.0

    sweeps_history = []

    # value iteration
    while True:
        old_state_value = state_value.copy()
        sweeps_history.append(old_state_value)

        # 对每一个 s ∈ S循环：
        for state in STATES[1:GOAL]:
```

```
            # 当前状态的行为空间上界不会超过：持有金额/距离获胜所需金额
            actions = np.arange(min(state, GOAL - state) + 1)
            # 遍历行为空间，目的是找出max_a
            action_returns = []
            for action in actions:
                # p:HEAD_PROB、(1 - HEAD_PROB)
                # r: 0
                # v'(s)：后继状态价值state_value[state +(赢)\-（输） action]
                action_returns.append(
                    HEAD_PROB * state_value[state + action] + (1 - HEAD_PROB) * state_value[state - action])
            # 找出所有行为a下的max-value
            new_value = np.max(action_returns)
            state_value[state] = new_value
        # 价值收敛
        delta = abs(state_value - old_state_value).max()
        if delta < 1e-9:
            sweeps_history.append(state_value)
            break

    # 输出最终确定的最优策略
    policy = np.zeros(GOAL + 1)
    for state in STATES[1:GOAL]:
        actions = np.arange(min(state, GOAL - state) + 1)
        action_returns = []
        for action in actions:
            action_returns.append(
                HEAD_PROB * state_value[state + action] + (1 - HEAD_PROB) * state_value[state - action])

        # action_returns从1开始（0代表输光），np.round保留5位小数
        # 取action_returns最大值的下标
        policy[state] = actions[np.argmax(np.round(action_returns[1:], 5)) + 1]

    plt.figure(figsize=(10, 20))

    plt.subplot(2, 1, 1)
    for sweep, state_value in enumerate(sweeps_history):
        plt.plot(state_value, label='sweep {}'.format(sweep))
    plt.xlabel('Capital')
    plt.ylabel('Value estimates')
    plt.legend(loc='best')

    plt.subplot(2, 1, 2)
    plt.scatter(STATES, policy)
    plt.xlabel('Capital')
    plt.ylabel('Final policy (stake)')

    plt.savefig('../images/figure_4_3.png')
    plt.close()
```

# 4. 测试结果

下图显示了在价值迭代遍历过程中价值函数的变化，以及在$p_h = 0.4$时最终找到的策略，这个策略是最优的，但并不是唯一的。事实上存在一系列的最优策略，具体取决于在argmax函数相等时的动作选取。


sweep 0
sweep 1
sweep 2
sweep 3
sweep 4
sweep 5
sweep 6
sweep 7
sweep 8
sweep 9
sweep 10
sweep 11
sweep 12
sweep 13
sweep 14
sweep 15
sweep 16
sweep 17
Value estimates
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
Capital


Final policy (stake)
50
40
30
20
10
0

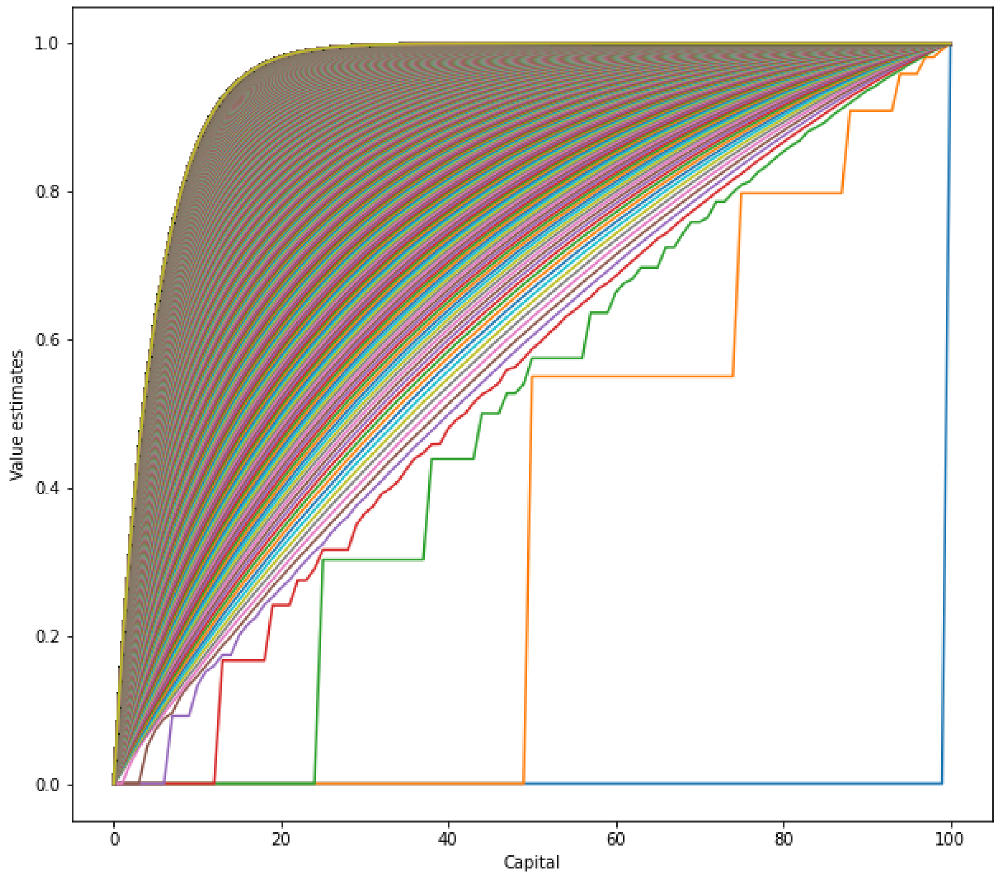

这里有个问题在于为什么最优策略看起来有点奇怪，特别是在Captial为50的时候，他会选择全部投注，但是当51时却没那么激进。

其原因应该在于，智能体是在尝试着尽快结束，以达到全输完 (reward=0），或者赢得最终奖励(reward=1)。我们注意到，在资金为25的时候，如果我们全部下注可能得到50。在50的时候同样梭哈，可以得到100，这里仅用了两次就获得了reward。但是如果我们在资金25的时候下注少于25，那么要达到100这个目标就必须多玩几局。 智能体追求下注次数少的原因应该在于，次数越少，分布越不均匀（相反，次数越多分布越均匀），智能体可以充分利用分布不均匀时候的方差来增加实现目标的可能性。

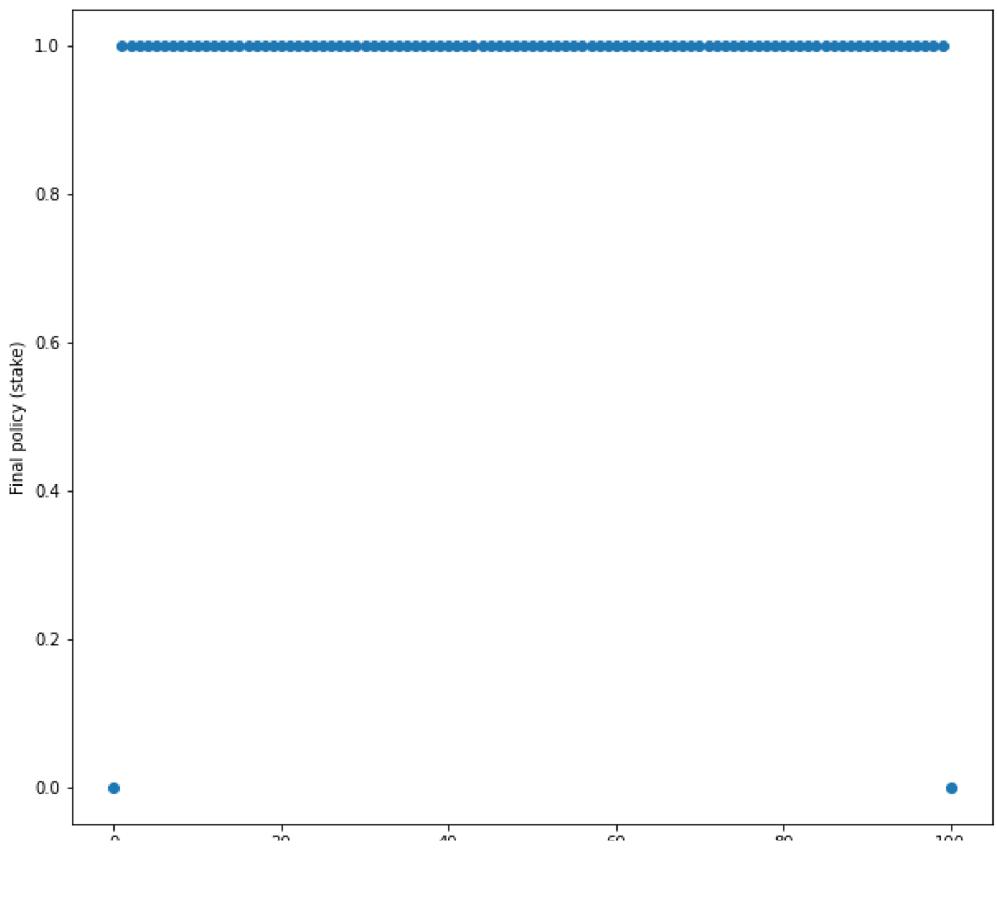
# 5、扩展分析

## $p_h = 0.55$的测试结果

上面的结果是基于硬币正面朝上的概率为HEAD_PROB = 0.4的结果，如果$p_h = 0.55$，那么经过了1600次的扫描后结果下图，可以看出，当硬币正面概率为0.55的时候，最优策略几乎总是下注1块，有点匪夷所思。可能是因为概率对我们比较有利。



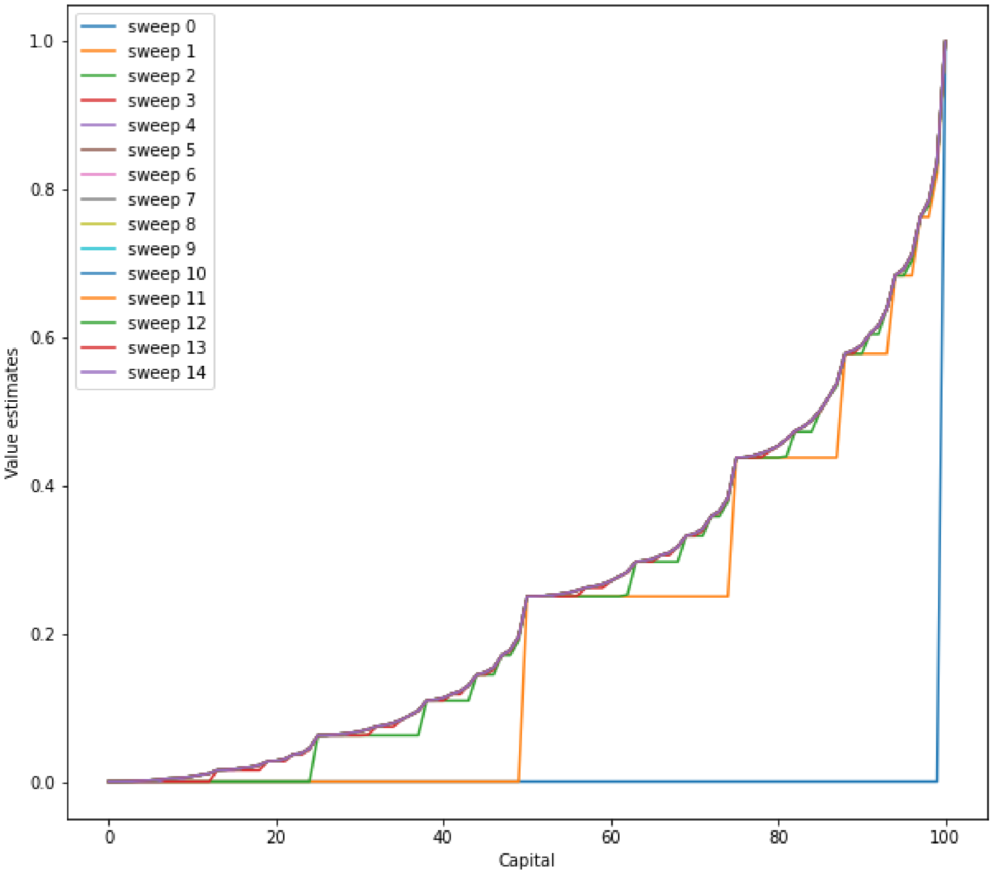
Value estimates
Capital
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100

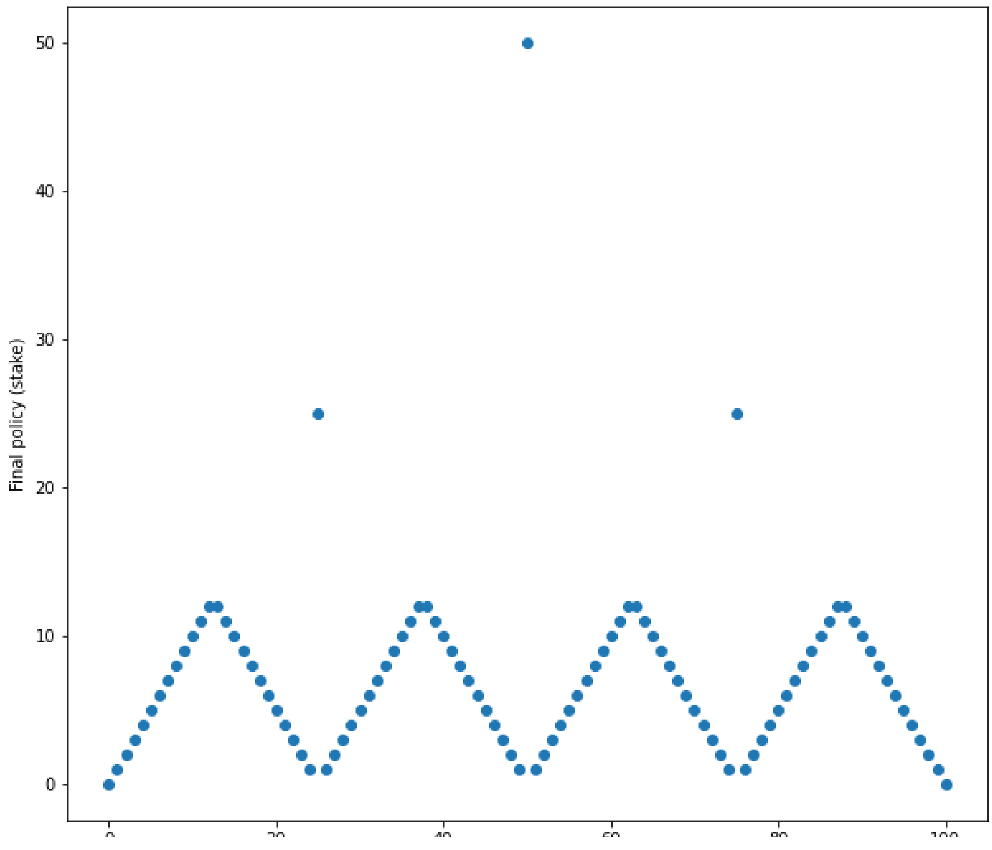



Final policy (stake)
1.0
0.8
0.6
0.4
0.2
0.0


## $p_h = 0.25$的测试结果

如果$p_h = 0.25$，那么结果为：


sweep 0
sweep 1
sweep 2
sweep 3
sweep 4
sweep 5
sweep 6
sweep 7
sweep 8
sweep 9
sweep 10
sweep 11
sweep 12
sweep 13
sweep 14
Value estimates
Capital


Final policy (stake)

0 20 40 60 80 100

Capital