迭代法

- 用手工列方程组的办法肯定可以得到解，但是当状态数量到几十个时，虽然都是一次线性方程组，手工计算量已经变得很大，非常容易出错。
- 矩阵法虽然可以得到数学解析解，但是，计算逆矩阵时非常耗时，时间复杂度为 $O(n^3)$，n 是状态的数量。当状态数量增加到千级、万级时，时间开销将变得不可接受。

式 $x = f(x)$ 的形式让我们想起了迭代法，比如式 1 所示的方程：

$$x = 10 + \log_{10}(\frac{x}{2}) \tag{1}$$

用迭代法是这样解，首先假定等式左侧的 x 为 $x_{[k+1]}$，而右侧的为 $x_{[k]}$，如式 2：

$$x_{[k+1]} = 10 + \log_{10}(\frac{x_{[k]}}{2}) \tag{2}$$

任意给定 x_k 的初始值（在本问题中需要 $x > 0$，否则 $\log_{10}(\frac{x_k}{2})$ 无法计算），计算出 x_{k+1}，如此重复，然后检查 $|x_{k+1} - x_k|$ 是否小于一个给定的误差，即可判定收敛。

写成代码也很简单：

【代码位置：0_Simple_Iteration.py】

```
import math

def f(x):
    y = 10 + math.log10(x/2)
    return y

if __name__=="__main__":
    x = 100
    delta = 100
    count = 0
    while delta > 1e-5:      # 判断是否收敛
        count += 1           # 计数器
        y = f(x)             # 迭代计算
        delta = abs(x - y)   # 检查误差
        print(str.format("{0}: x={1}, 相对误差={2}", count, y, delta))
        x = y                # 为下一次迭代做准备
```

在给定 x 初值为 100 的情况下，指定误差 delta 小于 1e-5 即可结束迭代，试验结果如下：

```
1: x=11.698970004336019, 相对误差=88.30102999566398
2: x=10.767117631798069, 相对误差=0.93185237253795
3: x=10.73106946212392, 相对误差=0.03604816967414948
4: x=10.72961301039834, 相对误差=0.0014564517255788445
5: x=10.72955406269718, 相对误差=5.894770116121606e-05
6: x=10.729551676708828, 相对误差=2.3859883508947632e-06
```

共迭代了 6 次，误差达到 2.38e-6，结果为 10.73。读者可自行带入式 1 中验证。

原始迭代法

根据上面的思路，我们尝试一下最原始的手工迭代，以帮助读者理解迭代过程。

根据式

$$V(s) = R(s) + \gamma \sum_{s'} P_{ss'} V(s') \tag{3}$$

可以定义 $V_{[k+1]}, V_{[k]}$ 的迭代关系如式 3：

$$V_{[k+1]}(s) = R(s) + \gamma \sum_{s'} P_{ss'} V_{[k]}(s') \tag{4}$$

可以给 $V_{[0]}$ 的初始值为全 0 ，下标 $[0]$ 表示迭代次数（下同）；然后根据式 4 进行第一次迭代，得到各个状态的奖励值 $V_{[1]}$（因为除了 $R(s)$ 以外，$V(s')$ 的值全为 0，所以第一次迭代的结果就是状态奖励值）：

$$V_{[0]} = \begin{cases} v_0 = 0 \\ v_1 = 0 \\ v_2 = 0 \\ v_3 = 0, \\ v_4 = 0 \\ v_5 = 0 \\ v_6 = 0 \end{cases} \qquad V_{[1]} = \begin{cases} v_0 = -3 \\ v_1 = 0 \\ v_2 = 1 \\ v_3 = 3 \\ v_4 = 2 \\ v_5 = -1 \\ v_6 = 0 \end{cases}$$

第二次迭代，根据式 3 可以写出方程组：

$$V_{[2]} = \begin{cases} v_0 = -3 + 0.7v_0 + 0.3v_1 = (-3) + 0.7 \cdot (-3) + 0.3 \cdot 0 = -5.1 \\ v_1 = 0 + 0.6v_0 + 0.4v_2 = 0 + 0.6 \cdot (-3) + 0.4 \cdot 1 = -1.4 \\ v_2 = 1 + 0.9v_3 + 0.1v_6 = 1 + 0.9 \cdot 3 + 0.1 \cdot 0 = 3.7 \\ v_3 = 3 + 0.2v_4 + 0.8v_5 = 3 + 0.2 \cdot 2 + 0.8 \cdot (-1) = 2.6 \\ v_4 = 2 + 0.2v_1 + 0.5v_2 + 0.3v_3 = 2 + 0.2 \cdot 0 + 0.5 \cdot 1 + 0.3 \cdot 3 = 3.4 \\ v_5 = -1 + v_6 = -1 + 0 = -1 \\ v_6 = 0 \end{cases}$$

把 $V_{[2]}$ 的结果带入式 3 做第三次迭代：

$$V_{[3]} = \begin{cases} v_0 = -3 + 0.7 \cdot (-5.1) + 0.3 \cdot (-1.4) = -6.99 \\ v_1 = 0 + 0.6 \cdot (-5.1) + 0.4 \cdot 3.7 = -1.58 \\ v_2 = 1 + 0.9 \cdot 2.6 + 0.1 \cdot 0 = 3.34 \\ v_3 = 3 + 0.2 \cdot 3.4 + 0.8 \cdot (-1) = 2.88 \\ v_4 = 2 + 0.2 \cdot (-1.4) + 0.5 \cdot 3.7 + 0.3 \cdot 2.6 = 4.35 \\ v_5 = -1 \\ v_6 = 0 \end{cases}$$

依此类推，读者可以比较 $V_{[0]}, V_{[1]}, V_{[2]}, V_{[3]}$ 的数值，是不是一步步地向着上一小节中用矩阵法得到的结果迈进。

做为一名伟大的程序员，我们当然要用代码来解决上述的繁复手工计算过程：
【代码位置：1_Linear_Equations_Iteration.py】

```
# 线性方程组原始迭代法
def linear_equations_iteration(dataModel, gamma):
    print("---原始迭代法---")
    V_next = np.zeros(dataModel.N)    # 初始化为全 0
    count = 0
    while (count < 1000):    # 1000 是随意指定的一个比较大的数，避免不收敛而导致while无限
        V = V_next.copy()    # 准备一个备份，用于比较，检查是否收敛
```

```
        count += 1  # 计数器+1
        # 列方程组，更新 V_next 的值
        V_next[0] = dataModel.R[0] + gamma*(0.7 * V[0] + 0.3 * V[1])
        V_next[1] = dataModel.R[1] + gamma*(0.6 * V[0] + 0.4 * V[2])
        V_next[2] = dataModel.R[2] + gamma*(0.9 * V[3] + 0.1 * V[6])
        V_next[3] = dataModel.R[3] + gamma*(0.2 * V[4] + 0.8 * V[5])
        V_next[4] = dataModel.R[4] + gamma*(0.2 * V[1] + 0.5 * V[2] + 0.3 *
V[3])
        V_next[5] = dataModel.R[5] + gamma*V[6]
        V_next[6] = dataModel.R[6]
        if np.allclose(V_next, V):  # 检查是否收敛
            break
    print("迭代次数 :", count)
    return V
```

输出结果如下：

```
---原始迭代法---
迭代次数 : 98
[-21.63275089 -11.6331189  3.36614097  2.62904047  2.14517952  -1.  0.]
Bug:    -21.633
Coding: -11.633
Test:   3.366
Review: 2.629
Refactor:       2.145
Merge:  -1.0
End:    0.0
```

从结果上看，与矩阵法的结果非常接近，是可以接受的解。

矩阵迭代法

更一般地，求解线性方程组有一套成熟的做法，算法如下：

---

1. 定义迭代式：$x = Ax + B$
2. 给定任意初始值$x_{[0]}$，一般可以设置为 0 或其它随机数
3. 迭代求解 $x_{[k]} = Ax_{[k-1]} + B$
4. 得到：$x_{[0]}, x_{[1]}, \cdots, x_{[k]}$，直到序列收敛于某个值

对于式 4 来说：$A = \gamma P_{ss'}, x = V(s), B = R(s)$。代码实现如下：

【代码位置：2_Matrix_Iteration.py】

```python
# 矩阵迭代法
def matrix_iteration(dataModel, gamma):
    print("---矩阵迭代法---")
    V_next = np.zeros(dataModel.N)
    count = 0   # 迭代计数器
    while (count < 1000):   # 1000 是随意指定的一个比较大的数，避免不收敛而导致while无限
        count += 1  # 计数器+1
        V = V_next.copy()   # 准备一个备份，用于比较，检查是否收敛
        V_next = dataModel.R + gamma * np.dot(dataModel.P, V)   # 式 3
        if np.allclose(V_next, V):  # 检查收敛性
            break
    print("迭代次数 :", count)
    return V
```

其结果与上面的原始迭代法完全一致：

```
---矩阵迭代法---
迭代次数 : 98
[-21.63275089 -11.6331189    3.36614097   2.62904047   2.14517952  -1.   0.]
Bug:        -21.633
Coding:     -11.633
Test:       3.366
Review:     2.629
Refactor:   2.145
Merge:      -1.0
End:        0.0
```

矩阵迭代法为什么可以收敛？下面就来做一下理论推导。

收敛定理

对于线性方程组，用矩阵迭代方式可以表示为：

$$x_{[k]} = Ax_{[k-1]} + B \tag{5}$$

假设 $x_{[*]}$ 是最终的收敛值，则当 $k \to \infty$ 时有：

$$x_{[*]} = Ax_{[*]} + B \tag{6}$$

定义第 k 次迭代的误差为：

$$\begin{aligned} \varepsilon_{[k]} &= x_{[k]} - x_{[*]} \\ (\footnotesize\text{带入式}4,5) &= Ax_{[k-1]} + B - (Ax_{[*]} + B) \\ &= A(x_{[k-1]} - x_{[*]}) \\ &= A\varepsilon_{[k-1]} \\ (\footnotesize\text{迭代带入}\varepsilon_{[k-1]} = A\varepsilon_{[k-2]}) &= A^2\varepsilon_{[k-2]} \\ &\cdots \\ &= A^k\varepsilon_{[0]} = A^k(x_{[0]} - x_{[*]}) \end{aligned} \tag{7}$$

如果

$$\lim_{k \to +\infty} A^k = 0 \tag{8}$$

则式 7 表示的误差 $\varepsilon_{[k]}$ 也趋近于 0，即，$x_{[k]}$ 可以收敛到 $x_{[*]}$。所以，我们来检查一下 A^k 是否趋近于 0 就可以了。

【代码位置：2_Matrix_Iteration.py】

```
def check_convergence(dataModel):
    print("迭代100次，检查状态转移矩阵是否趋近于 0: ")
    P_new = dataModel.P.copy()
    for i in range(100):
        P_new = np.dot(dataModel.P, P_new)
    print(np.around(P_new, 3))
```

式 5 中的 A 实际就是本问题中的状态转移矩阵 P，结果为：

```
迭代100次，检查状态转移矩阵是否趋近于 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. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]]
```

可以看到 100 次迭代后，式 8 满足条件，所以证明了矩阵迭代法是可以收敛的。

- 理论上，当 A 的谱半径小于 1 时迭代法收敛。谱半径就是特征值绝对值集合的上确界，一般若为方阵 A 的谱半径则写作 ρ(A)。
- 实际上，一般只要 A 可逆，迭代就能收敛。

双数组迭代 vs. 单数组原地更新

观察原始迭代法中的代码，读者会发现我们“聪明地”使用了两个数组，一个是 V 来存储上一轮迭代的状态数值，另一个是 V_{next} 来计算本轮迭代的状态数值，这样可以保证所有的 V_{next} 都是从上一次迭代的 V 计算得到，很“干净”。

但是，真的需要这么“干净”吗？如果把式 4 改成式 9：

$$V_{[k]}(s) = R(s) + \gamma \sum_{s'} P_{ss'} V_{[k]}(s') \tag{9}$$

会如何呢？

一个不怎么伟大的程序员也可以立刻写出如下代码：
【代码位置：3_Linear_Equations_Iteration_SingleArray.py】

```
# 单数组原始迭代法
def linear_equations_iteration_single_array(dataModel, gamma):
    print("---单数组原始迭代法---")
    V = np.zeros(dataModel.N)   # 初始化为全 0
    count = 0   # 迭代计数器
    while (count < 1000):   # 1000 是随意指定的一个比较大的数，避免不收敛而导致while无限
        count += 1          # 计数器+1
        V_old = V.copy()    # 备份上一次的迭代值用于检查收敛性
        # 线性方程组
        V[0] = dataModel.R[0] + gamma*(0.7 * V[0] + 0.3 * V[1])
        V[1] = dataModel.R[1] + gamma*(0.6 * V[0] + 0.4 * V[2])
        V[2] = dataModel.R[2] + gamma*(0.9 * V[3] + 0.1 * V[6])
        V[3] = dataModel.R[3] + gamma*(0.2 * V[4] + 0.8 * V[5])
        V[4] = dataModel.R[4] + gamma*(0.2 * V[1] + 0.5 * V[2] + 0.3 * V[3])
        V[5] = dataModel.R[5] + gamma*(1.0 * V[6])
```

```
        V[6] = dataModel.R[6]
        if np.allclose(V_old, V):   # 检查收敛
            break
    print("迭代次数 :", count)
    return V
```

与双数组方法相比，就是把线性方程组等式前面的 V_next 改成 V 了。运行结果如下：

```
---单数组原始迭代法---
迭代次数 : 81
[-21.63293714 -11.63330975   3.36612696   2.62902565   2.14510923  -1.    0.]
Bug:        -21.633
Coding:     -11.633
Test:       3.366
Review:     2.629
Refactor:   2.145
Merge:      -1.0
End:        0.0
```

读者会惊奇地发现，迭代次数从以前的 98 次变成了 81 次即达到收敛状态。这是为什么呢?

原因是这样的：观察线性方程组中的代码，当第一行计算完 V[0] 后（这里的方括号表示状态数组），第二行在计算 V[1] 时立刻就用到了新的 V[0]，……，计算 V[4] 时就已经用到了最新的 V[1],V[2],V[3]。所以，收敛的速度变快了。在处理动态规划问题时，一般都使用这种原地更新法（in place update）。

有兴趣的读者可以修改一下 V[0]~V[6] 的计算顺序，看看是否还可以提高一点儿迭代效率。如果前后依赖比较强的话，更改遍历计算的顺序会提高性能；但如果是交叉依赖，就不那么明显了。

通用的迭代实现

前面的原始迭代法其实就是矩阵迭代法，只是把矩阵运算的过程变成实例化代码了，但是只能针对本案例有效，换一个问题时，就需要重新书写代码。而矩阵迭代法只需要修改数据结构中的概率转移矩阵 dataModel.P 的具体内容就可以了适应任何场景了。

但是，有时候由于状态成千上万，没有可能写出状态转移矩阵来，对于人类来说，手工维护一个 20×20 的矩阵，已经是极限了，需要非常小心才能不出错。所以，最佳设计是只需要得到在某个状态下转移到可能达到的下游状态的列表，而不是转移到所有状态（包括不能达到的状态即概率为 0）的矩阵。

举例来说，在本问题中，状态转移矩阵是这样定义的：
【代码位置：CodeLifeCycle_DataModel_P.py】

```
# 状态转移概率
P = np.array(
    [   # B   C    T    R    F    M    E
        [0.7, 0.3, 0.0, 0.0, 0.0, 0.0, 0.0],    # Bug
        [0.6, 0.0, 0.4, 0.0, 0.0, 0.0, 0.0],    # Coding
        [0.0, 0.0, 0.0, 0.9, 0.0, 0.0, 0.1],    # Test (CI)
        [0.0, 0.0, 0.0, 0.0, 0.2, 0.8, 0.0],    # Review
        [0.0, 0.2, 0.5, 0.3, 0.0, 0.0, 0.0],    # reFactor
        [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0],    # Merge
        [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]     # End
    ]
)
```

其中第二行表示从 状态 Coding 到其它所有 7 个状态的转移概率，但其中有些状态是达不到的，所以只有两个值大于 0，是一个稀疏矩阵：

```
[0.6, 0.0, 0.4, 0.0, 0.0, 0.0, 0.0],    # Coding
```

对于这种情况，可以定义另外一种数据结构，来解决稀疏问题：

```
# 用字典代替状态转移矩阵
D = {
    States.Bug:     [(States.Bug, 0.7),     (States.Coding, 0.3)],
    States.Coding:  [(States.Bug, 0.6),     (States.Test, 0.4)],
    States.Test:    [(States.Review, 0.8),  (States.End, 0.1)],
    States.Review:  [(States.Refactor, 0.2),(States.Merge, 0.8)],
    States.Refactor:[(States.Coding, 0.2),  (States.Test, 0.5),(States.Review,
0.3)],
    States.Merge:   [(States.End, 1.0)],
    States.End:     [(States.End, 1.0)]
}
```

用第一行数据举例，它表示：在 Bug 状态下，可以以 0.7 的概率转移到 Bug，以 0.3 的概率转移到 Coding。

相应地，需要改动数据模型代码：
【代码位置：CodeLifeCycle_DataModel_D.py】

```
class DataModel(object):
    def __init__(self):
        self.D = D                          # 状态转移字典
        self.R = Rewards                    # 奖励
        self.S = States                     # 状态集
        self.N = len(self.S)                # 状态数量
        self.E = [self.S.End]       # 终止状态集

    def get_next(self, curr_s):
        list_state_prob = self.D[curr_s]    # 根据当前状态返回可用的下游状态及其概率
        return list_state_prob
```

因此，必须改进原始迭代法中的代码，让它可以适应通用场景。

单数组就地更新算法

定义误差 $error$
任意初始化 $V(s)$，其中 $V(s_{End}) = 0$
循环：
    $V_{old}(s) \leftarrow V(s)$
    对每一个 $s \in S$:
        $V(s) \leftarrow R_{ss'} + \gamma \sum P_{ss'} V(s')$
    检查收敛性 $|V_{old} - V| < error$
    如收敛则退出循环

代码如下：

【代码位置：4_Bellman_Equation_Iteration_SingleArray.py】

```python
# 贝尔曼方程单数组就地更新
def Bellman_iteration_single_array(dataModel, gamma):
    print("---单数组就地更新法---")
    V = np.zeros(dataModel.N)
    count = 0
    while True:
        count += 1
        V_old = V.copy()    # 复制备份用于检查收敛性
        # 遍历每一个 state 作为 curr_state
        for curr_state in dataModel.S:
            # 得到转移概率
            list_state_prob  = dataModel.get_next(curr_state)
            # 计算 \sum(P·V)
            v_sum = 0
            for next_state, next_prob in list_state_prob:
                v_sum += next_prob * V[next_state.value]
            # 计算 V = R + gamma * \sum(P·V)
            V[curr_state.value] = dataModel.R[curr_state.value] + gamma * v_sum
        # 检查收敛性
        if np.allclose(V, V_old):
            break
    print("迭代次数 :", count)
    return V
```

运行结果

```
迭代次数 : 81
[-21.63293714 -11.63330975   3.36612696   2.62902565   2.14510923  -1.  0.]
Bug:    -21.633
Coding: -11.633
Test:   3.366
Review: 2.629
Refactor:       2.145
Merge:  -1.0
End:    0.0
```

结果与线性方程组单数组的实现一样，都是 81 次迭代。但是，这一段代码可以适应各种应用场景，只要定义好 dataModel 中的状态转移字典即可，而不需要每次都改动算法代码。