Berechnung von V-Values auf einem 2D-Grid

```python
def calc_v_values(env, state = None):
    v = np.zeros(env.num_states())
    if state is None:
        for state in range(len(v)):
            v[state] = calc_v_values(env, state)[state]

    else:
        for action in range(env.num_actions()):
            next_state, reward, done = env.step_dp(state, action)
            v[state] = reward + GAMMA * v[next_state] * (done < 0.5)
    return v
```

Berechnung von Q-Values auf einem 2D-Grid

```python
def calc_q_values(env, v_table, state=None):
    if state is None:
        q = np.zeros((env.num_states(), env.num_actions()))
        for state in range(len(v_table)):
            q[state, :] = calc_q_values(env, v_table, state)

    else:
        q = np.zeros(env.num_actions())
        for action in range(env.num_actions()):
            next_state, reward, done = env.step_dp(state, action)
            q[action] = reward + GAMMA * v_table[next_state] * (done < 0.5)

    return q
```

Value Iteration

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

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

```python
def value_iteration(env):
    v_table = np.zeros(env.num_states())

    # calculate optimal value funciton
    while True:
        delta = 0
        for state in range(len(v_table)):
            # calculate q values for all action for the given state
            q = calc_q_values((env, v_table, state))
            best_q = np.max(q)
            # update delta
            delta = max(delta, np.abs(best_q - v_table[state]))
            # update v-value with best q-vaule
            v_table[state] = best_q

        if delta < THETA:
            break

    # calculate corresponding policy
    policy = np.zeros(env.num_states(), env.num_actions())
    for state in range(env.num_states()):
        q = calc_q_values(env, v_table, state)
        # set probability of best action to 1
        policy[state, np.argmax(q)] = 1

    return v_table, policy
```

Policy Iteration

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$; $V(terminal) \doteq 0$

2. Policy Evaluation
   Loop:
   $\quad\Delta \leftarrow 0$
   $\quad$Loop for each $s \in \mathcal{S}$:
   $\quad\quad v \leftarrow V(s)$
   $\quad\quad \boxed{V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))[r + \gamma V(s')]}$
   $\quad\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $\quad$ *old-action* $\leftarrow \pi(s)$
   $\quad \pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
   $\quad$ If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

```python
def policy_iteration(env):
    # use uniform policy as initialization
    policy = np.ones((env.num_states(), env.num_actions())) / env.num_actions()

    # calculate optimal value function
    while True:
        v_table = policy_evaluation(env, policy)
        policy, policy_stable = policy_improvement(env, policy, v_table)

        if policy_stable:
            break

    return v_table, policy
```

```python
def policy_improvement(env, policy_old, v_table):
    policy = np.zeros((env.num_states(), env.num_actions()))

    # greedy policy
    for state in range(env.num_states()):
        q = calc_q_values(env, v_table, state)
        policy[state, np.argmax(q)] = 1

    # check if policy has been changed since last iteration
    policy_stable = np.array_equal(policy, policy_old)

    return policy, policy_stable
```

```python
def policy_evaluation(env, policy):
    v_table = np.zeros(env.num_states())

    # calculate optimal value function
    while True:
        delta = 0
        for state in range(len(v_table)):
            # calculate q values for all action for the given state
            q = calc_q_values(env, v_table, state)
            pol = policy[state]
            # update delta
            delta = max(delta, np.abs(pol @ q - v_table[state]))
            # update v-value with best Q-value
            v_table[state] = pol @ q

        if delta < THETA:
            break

    return v_table
```

Monte Carlo policy evaluation

Idea: Average Q-values based on episodes run so far

$$Q(s,a) = \frac{1}{n} \sum_{i=1}^{n} G_i$$

Gi: return of i-the episode of state s/action a

Incremental MC updates: Apply incremental average formula to Q-values

$$Q(s,a) \leftarrow Q(s,a) + \frac{1}{n}\big(G - Q(s,a)\big)$$

If the true Q-values change over time (nonstationary problems) it can be useful to track an exponential moving average, i.e. to forget old episodes over time

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(G - Q(s,a)\big)$$

SARSA

Idea: Combine incremental Monte Carlo update with Bellmann expectation/optimality equations

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(G - Q(s,a)\big)$$

$$G = R + \gamma Q(s',a')$$

$$Q_*(s,a) = E\left[R + \gamma \max_{a'} Q_*(s',a')\right]$$

G in Q einsetzen:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(R + \gamma Q(s',a') - Q(s,a)\big)$$

Q-Learning: G in Q* einsetzen

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(R + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$$