Assignment 2: Markov Decision Processes

Homework Instructions

All your answers should be written in this notebook. You shouldn't need to write or modify any other files. Look for four instances of "YOUR CODE HERE"--those are the only parts of the code you need to write. To grade your homework, we will check whether the printouts immediately following your code match up with the results we got. The portions used for grading are highlighted in yellow. (However, note that the yellow highlighting does not show up when github renders this file.)

To submit your homework, send an email to berkeleydeeprlcourse@gmail.com (mailto:berkeleydeeprlcourse@gmail.com) with the subject line "Deep RL Assignment 2" and two attachments:

1. This `ipynb` file
2. A pdf version of this file (To make the pdf, do `File - Print Preview`)

The homework is due Febrary 22nd, 11:59 pm.

Introduction

This assignment will review the two classic methods for solving Markov Decision Processes (MDPs) with finite state and action spaces. We will implement value iteration (VI) and policy iteration (PI) for a finite MDP, both of which find the optimal policy in a finite number of iterations.

The experiments here will use the Frozen Lake environment, a simple gridworld MDP that is taken from `gym` and slightly modified for this assignment. In this MDP, the agent must navigate from the start state to the goal state on a 4x4 grid, with stochastic transitions.

In [1]:
```
from frozen_lake import FrozenLakeEnv
env = FrozenLakeEnv()
print(env.__doc__)
```

```
    Winter is here. You and your friends were tossing around a frisbee
 at the park
    when you made a wild throw that left the frisbee out in the middle
 of the lake.
    The water is mostly frozen, but there are a few holes where the ice
has melted.
    If you step into one of those holes, you'll fall into the freezing
 water.
    At this time, there's an international frisbee shortage, so it's ab
solutely imperative that
    you navigate across the lake and retrieve the disc.
    However, the ice is slippery, so you won't always move in the direc
tion you intend.
    The surface is described using a grid like the following

        SFFF
        FHFH
        FFFH
        HFFG

    S : starting point, safe
    F : frozen surface, safe
    H : hole, fall to your doom
    G : goal, where the frisbee is located

    The episode ends when you reach the goal or fall in a hole.
    You receive a reward of 1 if you reach the goal, and zero otherwis
e.
```

Let's look at what a random episode looks like.

In [68]:
```python
# Some basic imports and setup
import numpy as np, numpy.random as nr, gym
np.set_printoptions(precision=3)
def begin_grading(): print("\x1b[43m")
def end_grading(): print("\x1b[0m")

# Seed RNGs so you get the same printouts as me
env.seed(0); from gym.spaces import prng; prng.seed(10)
# Generate the episode
env.reset()
for t in range(100):
    env.render()
    a = env.action_space.sample()
    ob, rew, done, _ = env.step(a)
    if done:
        break
assert done
env.render();
```

```
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
```

In the episode above, the agent falls into a hole after two timesteps. Also note the stochasticity--on the first step, the DOWN action is selected, but the agent moves to the right.

We extract the relevant information from the gym Env into the MDP class below. The `env` object won't be used any further, we'll just use the `mdp` object.

In [69]:
```
class MDP(object):
    def __init__(self, P, nS, nA, desc=None):
        self.P = P # state transition and reward probabilities, explaine
d below
        self.nS = nS # number of states
        self.nA = nA # number of actions
        self.desc = desc # 2D array specifying what each grid cell means
 (used for plotting)
mdp = MDP( {s : {a : [tup[:3] for tup in tups] for (a, tups) in a2d.item
s()} for (s, a2d) in env.P.items()}, env.nS, env.nA, env.desc)


print("mdp.P is a two-level dict where the first key is the state and th
e second key is the action.")
print("The 2D grid cells are associated with indices [0, 1, 2, ..., 15]
 from left to right and top to down, as in")
print(np.arange(16).reshape(4,4))
print("mdp.P[state][action] is a list of tuples (probability, nextstate,
 reward).\n")
print("For example, state 0 is the initial state, and the transition inf
ormation for s=0, a=0 is \nP[0][0] =", mdp.P[0][0], "\n")
print("As another example, state 5 corresponds to a hole in the ice, whi
ch transitions to itself with probability 1 and reward 0.")
print("P[5][0] =", mdp.P[5][0], '\n')
```

```
mdp.P is a two-level dict where the first key is the state and the seco
nd key is the action.
The 2D grid cells are associated with indices [0, 1, 2, ..., 15] from l
eft to right and top to down, as in
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
mdp.P[state][action] is a list of tuples (probability, nextstate, rewar
d).

('For example, state 0 is the initial state, and the transition informa
tion for s=0, a=0 is \nP[0][0] =', [(0.1, 0, 0.0), (0.8, 0, 0.0), (0.1,
4, 0.0)], '\n')
As another example, state 5 corresponds to a hole in the ice, which tra
nsitions to itself with probability 1 and reward 0.
('P[5][0] =', [(1.0, 5, 0)], '\n')
```

Part 1: Value Iteration

Problem 1: implement value iteration

In this problem, you'll implement value iteration, which has the following pseudocode:

Initialize $V^{(0)}(s) = 0$, for all s

For $i = 0, 1, 2, \dots$

- $V^{(i+1)}(s) = \max_a \sum_{s'} P(s, a, s')[R(s, a, s') + \gamma V^{(i)}(s')]$, for all s

---

We additionally define the sequence of greedy policies $\pi^{(0)}, \pi^{(1)}, \dots, \pi^{(n-1)}$, where

$$\pi^{(i)}(s) = \arg\max_a \sum_{s'} P(s, a, s')[R(s, a, s') + \gamma V^{(i)}(s')]$$

Your code will return two lists: $[V^{(0)}, V^{(1)}, \dots, V^{(n)}]$ and $[\pi^{(0)}, \pi^{(1)}, \dots, \pi^{(n-1)}]$

To ensure that you get the same policies as the reference solution, choose the lower-index action to break ties in $\arg\max_a$. This is done automatically by np.argmax. This will only affect the "# chg actions" printout below--it won't affect the values computed.

Warning: make a copy of your value function each iteration and use that copy for the update--don't update your value function in place. Updating in-place is also a valid algorithm, sometimes called Gauss-Seidel value iteration or asynchronous value iteration, but it will cause you to get different results than me.

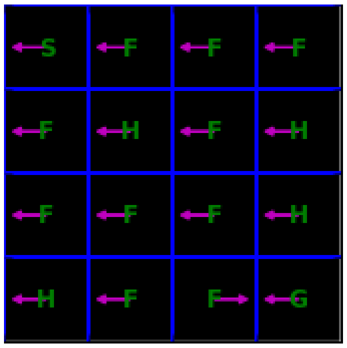
In [79]:
```
def value_iteration(mdp, gamma, nIt):
    """
    Inputs:
        mdp: MDP
        gamma: discount factor
        nIt: number of iterations, corresponding to n above
    Outputs:
        (value_functions, policies)

    len(value_functions) == nIt+1 and len(policies) == n
    """
    print("Iteration | max|V-Vprev| | # chg actions | V[0]")
    print("----------+--------------+---------------+---------")
    Vs = [np.zeros(mdp.nS)] # list of value functions contains the initi
al value function V^{(0)}, which is zero
    pis = []
    for it in range(nIt):
        oldpi = pis[-1] if len(pis) > 0 else None # \pi^{(it)} = Greedy
[V^{(it-1)}]. Just used for printout
        Vprev = Vs[-1] # V^{(it)}
        # YOUR CODE HERE
        # Your code should define the following two variables
        # pi: greedy policy for Vprev,
        #     corresponding to the math above: \pi^{(it)} = Greedy[V^{(i
t)}]
        #     numpy array of ints
        # V: bellman backup on Vprev
        #     corresponding to the math above: V^{(it+1)} = T[V^{(it)}]
        #     numpy array of floats

        nA = mdp.nA # Number of actions
        nS = mdp.nS # Number of states
        pi = np.zeros(nS) #pi is a numpy array of length 16 (one action
 per state)
        V = np.zeros(nS) # V is a numpy array of length 16 (one value pe
r state)

        for state in range(nS):
            expected_values = np.zeros(nA) # Each (s,a) pair will have o
ne expected value

            # Calculate all these expected values
            for action in range(nA):
                expected_value = 0
                # Now for each (s,a) there will be many possible s'
                for s_prime in mdp.P[state][action]:
                    expected_value += s_prime[0]*(s_prime[2] + gamma*Vpr
ev[s_prime[1]])
                expected_values[action] = expected_value

            # Now greedily choose the best action based on expected valu
es
            pi[state] = np.argmax(expected_values)
            V[state] = np.max(expected_values)

        max_diff = np.abs(V - Vprev).max()
        nChgActions="N/A" if oldpi is None else (pi != oldpi).sum()
        print("%4i      | %6.5f      | %4s          | %5.3f"%(it, max_di
ff, nChgActions, V[0]))
        Vs.append(V)
        pis.append(pi)
    return Vs, pis

GAMMA=0.95 # we'll be using this same value in subsequent problems
begin_grading()
Vs_VI, pis_VI = value_iteration(mdp, gamma=GAMMA, nIt=20)
end_grading()
```

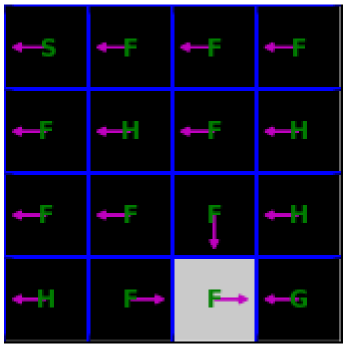
```
Iteration | max|V-Vprev| | # chg actions | V[0]
----------+--------------+---------------+---------
   0      | 0.80000      |  N/A          | 0.000
   1      | 0.60800      |    2          | 0.000
   2      | 0.51984      |    2          | 0.000
   3      | 0.39508      |    2          | 0.000
   4      | 0.30026      |    2          | 0.000
   5      | 0.25355      |    1          | 0.254
   6      | 0.10478      |    0          | 0.345
   7      | 0.09657      |    0          | 0.442
   8      | 0.03656      |    0          | 0.478
   9      | 0.02772      |    0          | 0.506
  10      | 0.01111      |    0          | 0.517
  11      | 0.00735      |    0          | 0.524
  12      | 0.00310      |    0          | 0.527
  13      | 0.00190      |    0          | 0.529
  14      | 0.00083      |    0          | 0.530
  15      | 0.00049      |    0          | 0.531
  16      | 0.00022      |    0          | 0.531
  17      | 0.00013      |    0          | 0.531
  18      | 0.00006      |    0          | 0.531
  19      | 0.00003      |    0          | 0.531
```

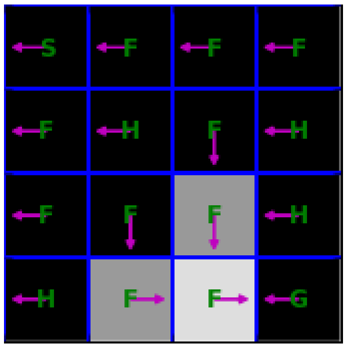
Below, we've illustrated the progress of value iteration. Your optimal actions are shown by arrows. At the bottom, the value of the different states are plotted.

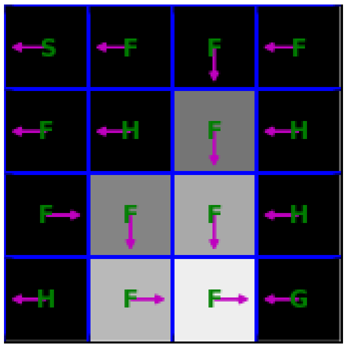
In [71]:
```
import matplotlib.pyplot as plt
%matplotlib inline
for (V, pi) in zip(Vs_VI[:10], pis_VI[:10]):
    plt.figure(figsize=(3,3))
    plt.imshow(V.reshape(4,4), cmap='gray', interpolation='none', clim=
(0,1))
    ax = plt.gca()
    ax.set_xticks(np.arange(4)-.5)
    ax.set_yticks(np.arange(4)-.5)
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    Y, X = np.mgrid[0:4, 0:4]
    a2uv = {0: (-1, 0), 1:(0, -1), 2:(1,0), 3:(-1, 0)}
    Pi = pi.reshape(4,4)
    for y in range(4):
        for x in range(4):
            a = Pi[y, x]
            u, v = a2uv[a]
            plt.arrow(x, y,u*.3, -v*.3, color='m', head_width=0.1, head_
length=0.1)
            plt.text(x, y, str(env.desc[y,x].item().decode()),
                     color='g', size=12,  verticalalignment='center',
                     horizontalalignment='center', fontweight='bold')
    plt.grid(color='b', lw=2, ls='-')
plt.figure()
plt.plot(Vs_VI)
plt.title("Values of different states");
```

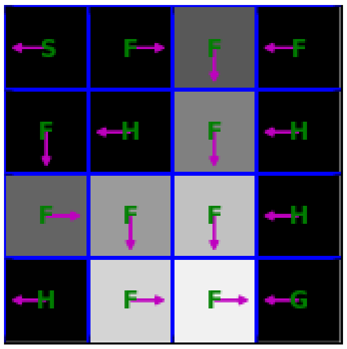
```
/Users/Ted/anaconda2/lib/python2.7/site-packages/matplotlib/font_manage
r.py:273: UserWarning: Matplotlib is building the font cache using fc-l
ist. This may take a moment.
  warnings.warn('Matplotlib is building the font cache using fc-list. T
his may take a moment.')
```

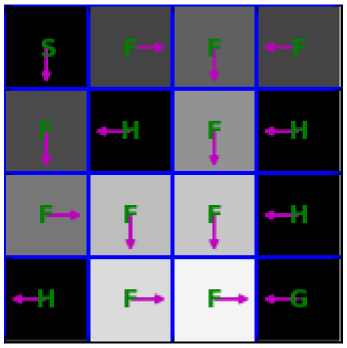


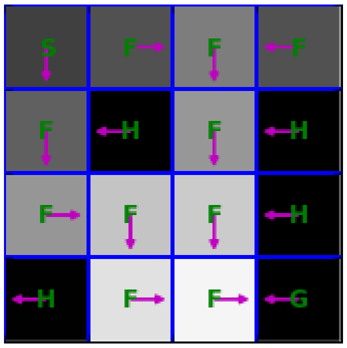


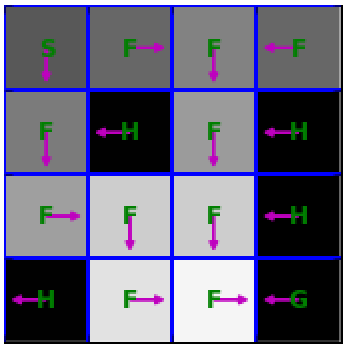


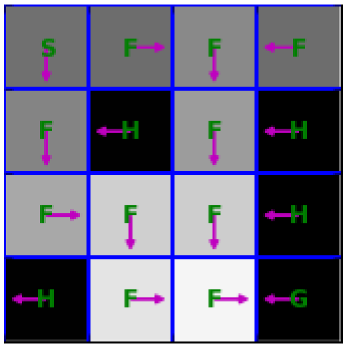


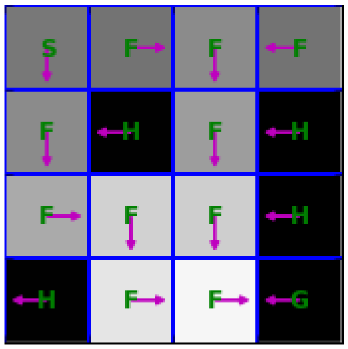

S F F F
F H F H
F F F H
H F F G

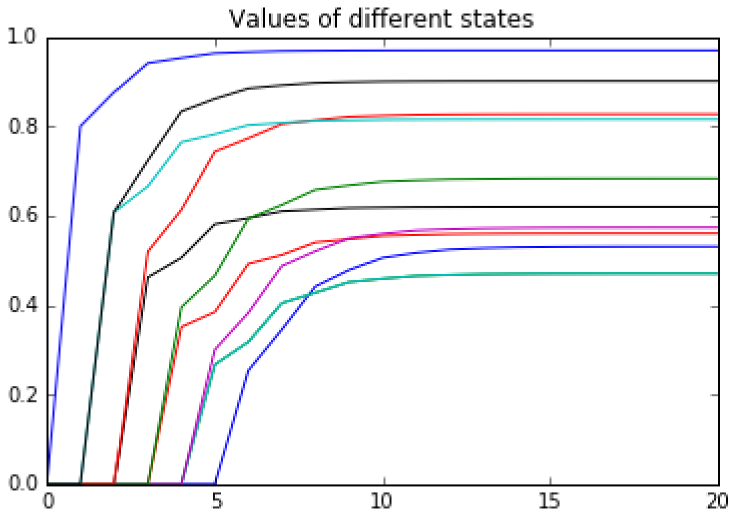

S F F F
F H F H
F F F H
H F F G


S F F F
F H F H
F F F H
H F F G


S F F F
F H F H
F F F H
H F F G


S F F F
F H F H
F F F H
H F F G





Problem 2: construct an MDP where value iteration takes a long time to converge

When we ran value iteration on the frozen lake problem, the last iteration where an action changed was iteration 6--i.e., value iteration computed the optimal policy at iteration 6. Are there any guarantees regarding how many iterations it'll take value iteration to compute the optimal policy? There are no such guarantees without additional assumptions--we can construct the MDP in such a way that the greedy policy will change after arbitrarily many iterations.

Your task: define an MDP with at most 3 states and 2 actions, such that when you run value iteration, the optimal action changes at iteration >= 50. Use discount=0.95. (However, note that the discount doesn't matter here--you can construct an appropriate MDP with any discount.)

In [80]:
```
chg_iter = 50
# YOUR CODE HERE
# Your code will need to define an MDP (mymdp)
# like the frozen lake MDP defined above
num_states = 3 # 0, 1, 2
num_actions = 2 # 0, 1

transitions = {0: {0: [(1.0, 1, 0.0)], 1: [(1.0, 0, 0.0993)]},
 1: {0: [(1.0, 2, 1.0)], 1: [(0.5, 2, 0.0), (0.5, 0, 0.2)]},
 2: {0: [(1.0, 2, 0.0)], 1: [(1.0, 2, 0.0)]}}

mymdp = MDP(transitions, num_states , num_actions,
          "This is an MDP designed for Value Iteration to take a long ti
me to converge")

begin_grading()
Vs, pis = value_iteration(mymdp, gamma=GAMMA, nIt=chg_iter+1)
end_grading()
```

```
Iteration | max|V-Vprev| | # chg actions | V[0]
----------+--------------+---------------+---------
   0      | 1.00000      |  N/A          | 0.099
   1      | 0.85070      |    1          | 0.950
   2      | 0.05180      |    1          | 1.002
   3      | 0.04921      |    0          | 1.051
   4      | 0.04675      |    0          | 1.098
   5      | 0.04441      |    0          | 1.142
   6      | 0.04219      |    0          | 1.184
   7      | 0.04008      |    0          | 1.224
   8      | 0.03808      |    0          | 1.263
   9      | 0.03617      |    0          | 1.299
  10      | 0.03437      |    0          | 1.333
  11      | 0.03265      |    0          | 1.366
  12      | 0.03101      |    0          | 1.397
  13      | 0.02946      |    0          | 1.426
  14      | 0.02799      |    0          | 1.454
  15      | 0.02659      |    0          | 1.481
  16      | 0.02526      |    0          | 1.506
  17      | 0.02400      |    0          | 1.530
  18      | 0.02280      |    0          | 1.553
  19      | 0.02166      |    0          | 1.574
  20      | 0.02058      |    0          | 1.595
  21      | 0.01955      |    0          | 1.615
  22      | 0.01857      |    0          | 1.633
  23      | 0.01764      |    0          | 1.651
  24      | 0.01676      |    0          | 1.668
  25      | 0.01592      |    0          | 1.683
  26      | 0.01513      |    0          | 1.699
  27      | 0.01437      |    0          | 1.713
  28      | 0.01365      |    0          | 1.727
  29      | 0.01297      |    0          | 1.740
  30      | 0.01232      |    0          | 1.752
  31      | 0.01170      |    0          | 1.764
  32      | 0.01112      |    0          | 1.775
  33      | 0.01056      |    0          | 1.785
  34      | 0.01003      |    0          | 1.795
  35      | 0.00953      |    0          | 1.805
  36      | 0.00906      |    0          | 1.814
  37      | 0.00860      |    0          | 1.823
  38      | 0.00817      |    0          | 1.831
  39      | 0.00776      |    0          | 1.838
  40      | 0.00738      |    0          | 1.846
  41      | 0.00701      |    0          | 1.853
  42      | 0.00666      |    0          | 1.860
  43      | 0.00632      |    0          | 1.866
  44      | 0.00601      |    0          | 1.872
  45      | 0.00571      |    0          | 1.878
  46      | 0.00542      |    0          | 1.883
  47      | 0.00515      |    0          | 1.888
  48      | 0.00489      |    0          | 1.893
  49      | 0.00465      |    0          | 1.898
  50      | 0.00442      |    1          | 1.902
```

Problem 3: Policy Iteration

The next task is to implement exact policy iteration (PI), which has the following pseudocode:

Initialize π_0

For $n = 0, 1, 2, \ldots$

- Compute the state-value function V^{π_n}
- Using V^{π_n}, compute the state-action-value function Q^{π_n}
- Compute new policy $\pi_{n+1}(s) = \operatorname{argmax}_a Q^{\pi_n}(s, a)$

Below, you'll implement the first and second steps of the loop.

Problem 3a: state value function

You'll write a function called `compute_vpi` that computes the state-value function V^π for an arbitrary policy π. Recall that V^π satisfies the following linear equation:

$$V^\pi(s) = \sum_{s'} P(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^\pi(s')]$$

You'll have to solve a linear system in your code. (Find an exact solution, e.g., with `np.linalg.solve`.)

In [81]:
```
def compute_vpi(pi, mdp, gamma):
    # YOUR CODE HERE
    nA = mdp.nA # Number of actions
    nS = mdp.nS # Number of states
    a = np.zeros((nS, nS)) # a is a (nS, nS) matrix that represents the
 coefficients in the linear system aV = b
    b = np.zeros(nS) # b is (nS, 1)

    for state in range(nS):
        a[state][state] += 1 # Vpi(state) on LHS of equation
        action = pi[state] # Get action from policy
        b_state = 0 # The entry in the b vector for each system
        for s_prime in mdp.P[state][action]:
            b_state += s_prime[0]*s_prime[2]
            a_coeff = gamma * s_prime[0]
            a[state][s_prime[1]] -= a_coeff # Can't set entry, since s'
 may not be unique. Move from RHS to LHS
        b[state] = b_state

    V = np.linalg.solve(a, b)
    return V
```

Now let's compute the value of an arbitrarily-chosen policy.

```
In [82]: begin_grading()
         print(compute_vpi(np.ones(16), mdp, gamma=GAMMA))
         end_grading()
```

```
[ 0.016  0.024  0.232  0.024  0.017 -0.     0.299  0.     0.02   0.188
  0.393  0.    -0.     0.196  0.494  0.   ]
```

As a sanity check, if we run `compute_vpi` on the solution from our previous value iteration run, we should get approximately (but not exactly) the same values produced by value iteration.

```
In [83]: Vpi=compute_vpi(pis_VI[15], mdp, gamma=GAMMA)
         V_vi = Vs_VI[15]
         print("From compute_vpi", Vpi)
         print("From value iteration", V_vi)
         print("Difference", Vpi - V_vi)
```

```
('From compute_vpi', array([ 0.531,  0.471,  0.56 ,  0.471,  0.574, -0.
   ,  0.62 ,  0.   ,
        0.683,  0.827,  0.815,  0.   ,  0.   ,  0.901,  0.97 ,  0.
 ]))
('From value iteration', array([ 0.53 ,  0.47 ,  0.56 ,  0.47 ,  0.573,
  0.   ,  0.62 ,  0.   ,
        0.683,  0.827,  0.815,  0.   ,  0.   ,  0.901,  0.97 ,  0.
 ]))
('Difference', array([  9.580e-04,   3.839e-04,   2.254e-04,   3.839e-0
4,   4.495e-04,
        -0.000e+00,   4.522e-05,   0.000e+00,   2.612e-04,   1.071e-04,
         3.272e-05,   0.000e+00,   0.000e+00,   3.977e-05,   7.051e-06,
         0.000e+00]))
```

Problem 3b: state-action value function

Next, you'll write a function to compute the state-action value function Q^π, defined as follows

$$Q^\pi(s,a) = \sum_{s'} P(s,a,s')[R(s,a,s') + \gamma V^\pi(s')]$$

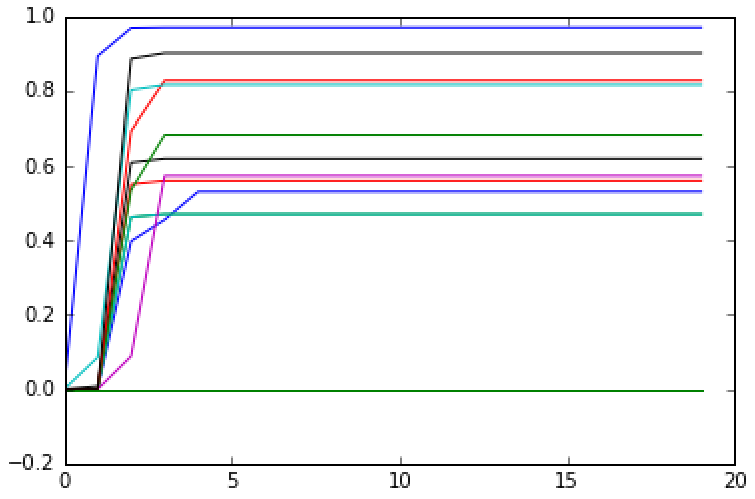
In [76]:
```
def compute_qpi(vpi, mdp,  gamma):
    # YOUR CODE HERE
    nA = mdp.nA # Number of actions
    nS = mdp.nS # Number of states
    Qpi = np.zeros((nS, nA))

    for state in range(nS):
            expected_values = np.zeros(nA) # Each (s,a) pair will have o
ne expected value
            # Calculate all these expected values
            for action in range(nA):
                expected_value = 0
                # Now for each (s,a) there will be many possible s'
                for s_prime in mdp.P[state][action]:
                    expected_value += s_prime[0]*(s_prime[2] +
gamma*vpi[s_prime[1]])
                Qpi[state][action] = expected_value
    return Qpi

begin_grading()
Qpi = compute_qpi(np.arange(mdp.nS), mdp, gamma=0.95)
print("Qpi:\n", Qpi)
end_grading()
```

```
('Qpi:\n', array([[  0.38 ,   3.135,   1.14 ,   0.095],
       [  0.57 ,   3.99 ,   2.09 ,   0.95 ],
       [  1.52 ,   4.94 ,   3.04 ,   1.9  ],
       [  2.47 ,   5.795,   3.23 ,   2.755],
       [  3.8  ,   6.935,   4.56 ,   0.855],
       [  4.75 ,   4.75 ,   4.75 ,   4.75 ],
       [  4.94 ,   8.74 ,   6.46 ,   2.66 ],
       [  6.65 ,   6.65 ,   6.65 ,   6.65 ],
       [  7.6  ,  10.735,   8.36 ,   4.655],
       [  7.79 ,  11.59 ,   9.31 ,   5.51 ],
       [  8.74 ,  12.54 ,  10.26 ,   6.46 ],
       [ 10.45 ,  10.45 ,  10.45 ,  10.45 ],
       [ 11.4  ,  11.4  ,  11.4  ,  11.4  ],
       [ 11.21 ,  12.35 ,  12.73 ,   9.31 ],
       [ 12.16 ,  13.4  ,  14.48 ,  10.36 ],
       [ 14.25 ,  14.25 ,  14.25 ,  14.25 ]]))
```

Now we're ready to run policy iteration!

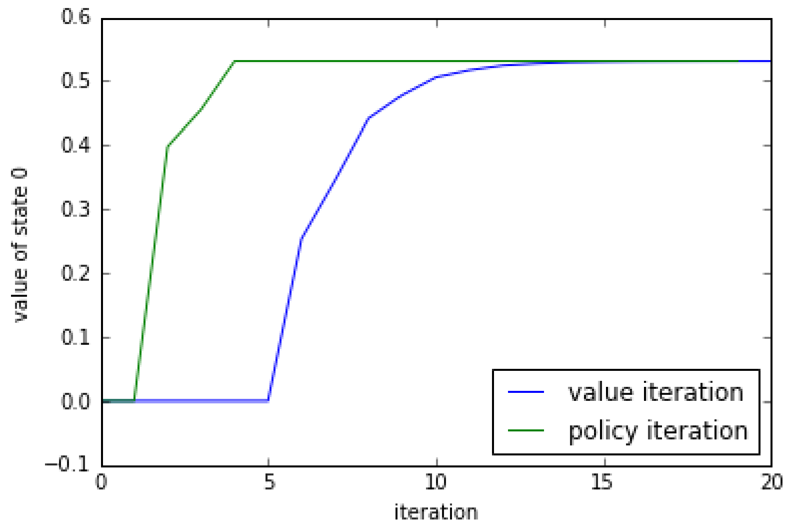
In [77]:
```
def policy_iteration(mdp, gamma, nIt):
    Vs = []
    pis = []
    pi_prev = np.zeros(mdp.nS,dtype='int')
    pis.append(pi_prev)
    print("Iteration | # chg actions | V[0]")
    print("----------+---------------+---------")
    for it in range(nIt):
        vpi = compute_vpi(pi_prev, mdp, gamma)
        qpi = compute_qpi(vpi, mdp, gamma)
        pi = qpi.argmax(axis=1)
        print("%4i      | %6i         | %6.5f"%(it, (pi != 
pi_prev).sum(), vpi[0]))
        Vs.append(vpi)
        pis.append(pi)
        pi_prev = pi
    return Vs, pis
Vs_PI, pis_PI = policy_iteration(mdp, gamma=0.95, nIt=20)
plt.plot(Vs_PI);
```

```
Iteration | # chg actions | V[0]
----------+---------------+---------
   0      |       1       | -0.00000
   1      |       9       | -0.00000
   2      |       2       | 0.39785
   3      |       1       | 0.45546
   4      |       0       | 0.53118
   5      |       0       | 0.53118
   6      |       0       | 0.53118
   7      |       0       | 0.53118
   8      |       0       | 0.53118
   9      |       0       | 0.53118
  10      |       0       | 0.53118
  11      |       0       | 0.53118
  12      |       0       | 0.53118
  13      |       0       | 0.53118
  14      |       0       | 0.53118
  15      |       0       | 0.53118
  16      |       0       | 0.53118
  17      |       0       | 0.53118
  18      |       0       | 0.53118
  19      |       0       | 0.53118
```

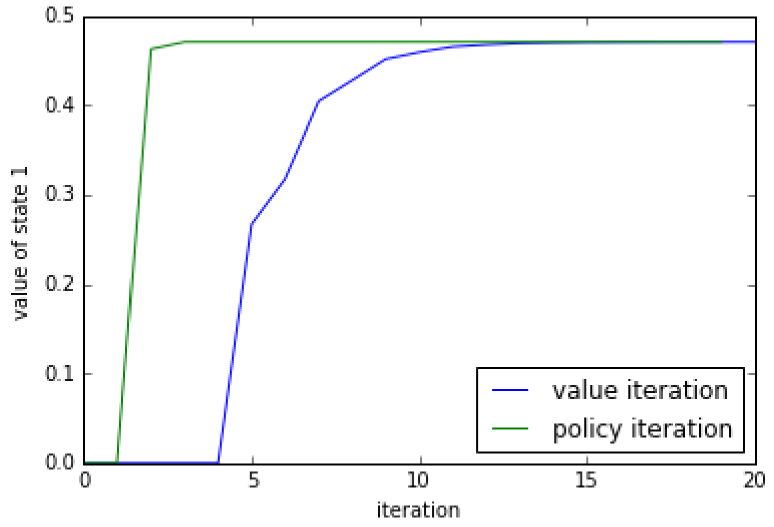


Now we can compare the convergence of value iteration and policy iteration on several states. For fun, you can try adding modified policy iteration.

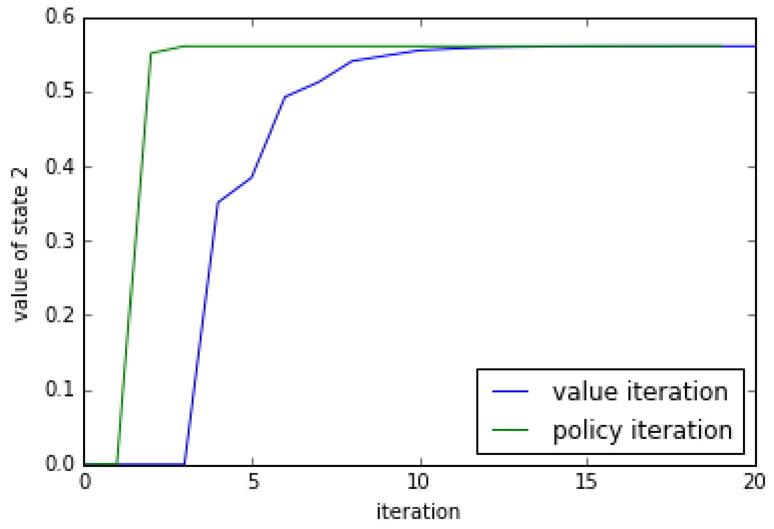
In [78]:
```
for s in range(5):
    plt.figure()
    plt.plot(np.array(Vs_VI)[:,s])
    plt.plot(np.array(Vs_PI)[:,s])
    plt.ylabel("value of state %i"%s)
    plt.xlabel("iteration")
    plt.legend(["value iteration", "policy iteration"], loc='best')
```



value of state 0
iteration
value iteration
policy iteration
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−0.1
0
5
10
15
20



value of state 1
iteration
value iteration
policy iteration
0.5
0.4
0.3
0.2
0.1
0.0
0
5
10
15
20



value of state 2
iteration
value iteration
policy iteration
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
5
10
15
20