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 [148]:
```
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 absolutely imperative that
    you navigate across the lake and retrieve the disc.
    However, the ice is slippery, so you won't always move in the direction 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 otherwise.
```

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

In [149]:
```
# 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 [150]:
```
class MDP(object):
    def __init__(self, P, nS, nA, desc=None):
        self.P = P # state transition and reward probabilities, explained 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.items()} 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 the 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 information 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, which transitions to itself with proba
bility 1 and reward 0.")
print("P[5][0] =", mdp.P[5][0], '\n')
print(mdp.desc)
```

```
mdp.P is a two-level dict where the first key is the state and the second key is the action.
The 2D grid cells are associated with indices [0, 1, 2, ..., 15] from left 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, reward).

For example, state 0 is the initial state, and the transition information for s=0, a=0 is
P[0][0] = [(0.1, 0, 0.0), (0.8, 0, 0.0), (0.1, 4, 0.0)]

As another example, state 5 corresponds to a hole in the ice, which transitions to itself with probability
 1 and reward 0.
P[5][0] = [(1.0, 5, 0)]

[[b'S' b'F' b'F' b'F']
 [b'F' b'H' b'F' b'H']
 [b'F' b'F' b'F' b'H']
 [b'H' b'F' b'F' b'G']]
```

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.

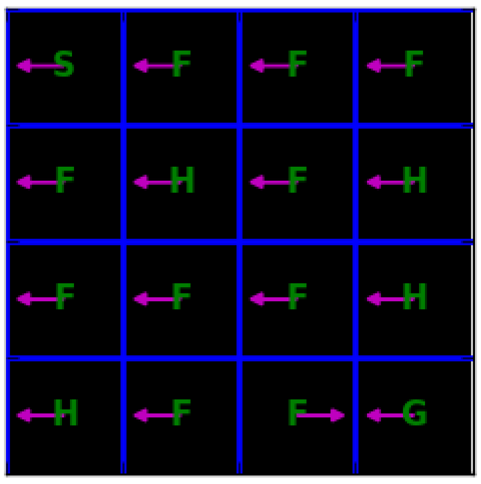
> Warning: make a copy of your value function each iteration and use that copy for the update--don't update your value function in place. Updating in-place is also a valid algorithm, sometimes called Gauss-Seidel value iteration or asynchronous value iteration, but it will cause you to get different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

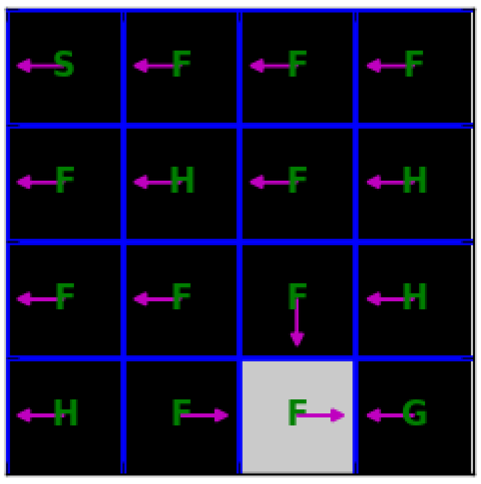
In [151]:
```python
def value_iteration(mdp, gamma, nIt):
    """
    Inputs:
        mdp: MDP
        gamma: discount factor
        nIt: number of iterations, corresponding to n above
    Outputs:
        (value_functions, policies)

    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 initial 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)}
        V = []
        pi = []
        for s in range(mdp.nS):
            futures = []
            for a in range(mdp.nA):
                prob, sprime, r = zip(*mdp.P[s][a])
                prob, sprime, r = np.array(prob), np.array(sprime), np.array(r)
                expected_reward = prob*(r + gamma*Vprev[sprime])
                futures+=[np.sum(expected_reward)]
            pi += [np.argmax(futures)]
            V += [futures[pi[s]]]



        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_diff, nChgActions, V[0]))
        Vs.append(np.array(V))
        pis.append(np.array(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()
```

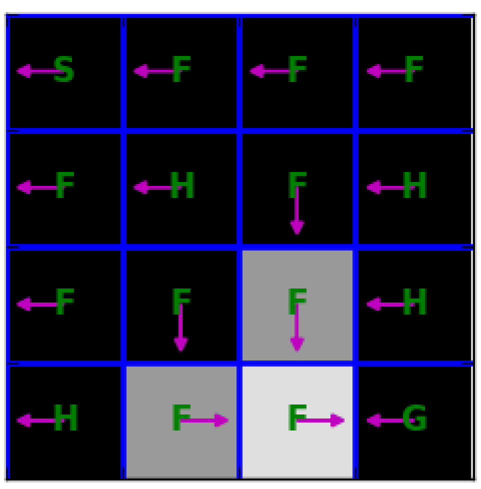
```
Iteration | max|V-Vprev| | # chg actions | V[0]
----------+--------------+---------------+---------
   0      | 0.80000      |  N/A          | 0.000
   1      | 0.60800      |    2          | 0.000
   2      | 0.51984      |    2          | 0.000
   3      | 0.39508      |    2          | 0.000
   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
```

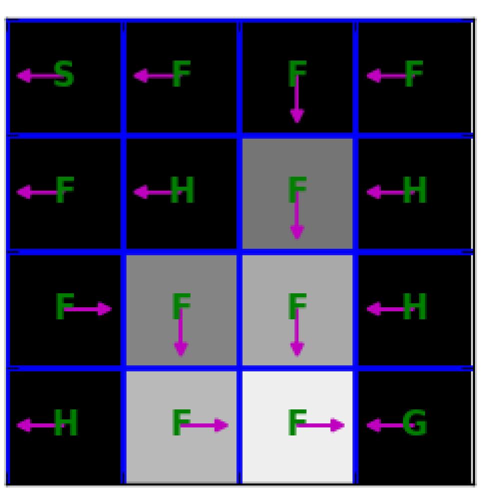
Below, we've illustrated the progress of value iteration. Your optimal actions are shown by arrows. At the bottom, the value of the different states are plotted.

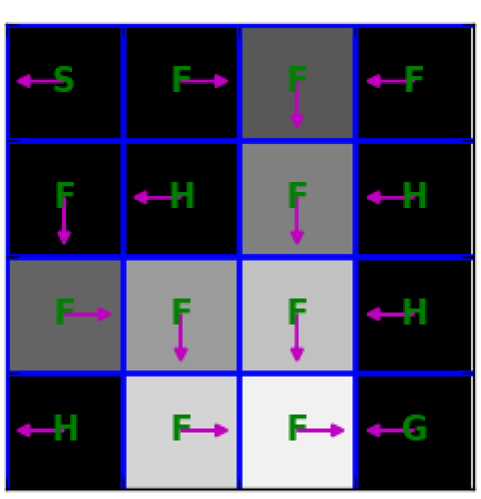
In [152]:
```
import matplotlib.pyplot as plt
%matplotlib inline
for (V, pi) in zip(Vs_VI[:10], pis_VI[:10]):
    plt.figure(figsize=(3,3))
    plt.imshow(V.reshape(4,4), cmap='gray', interpolation='none', clim=(0,1))
    ax = plt.gca()
    ax.set_xticks(np.arange(4)-.5)
    ax.set_yticks(np.arange(4)-.5)
    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");
```

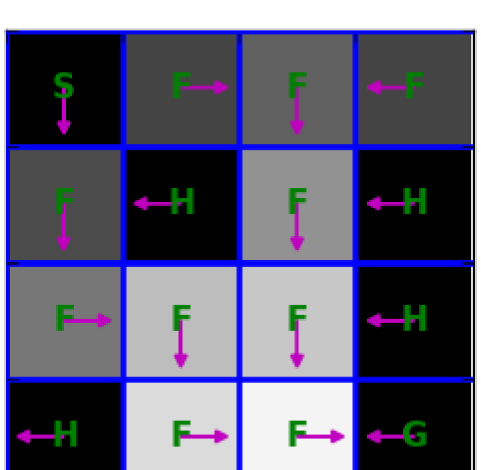

S F F F
F H F H
F F F H
H F F G

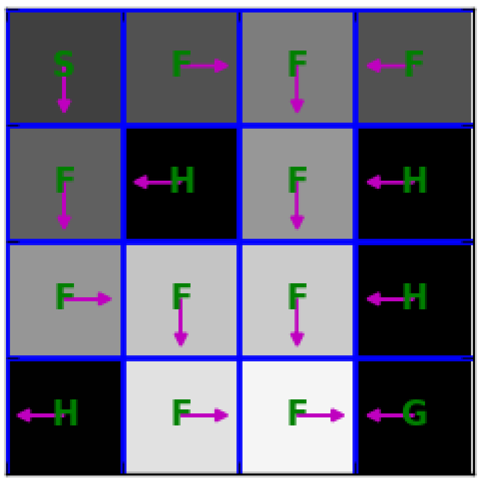

S F F F
F H F H
F F F H
H F F G


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

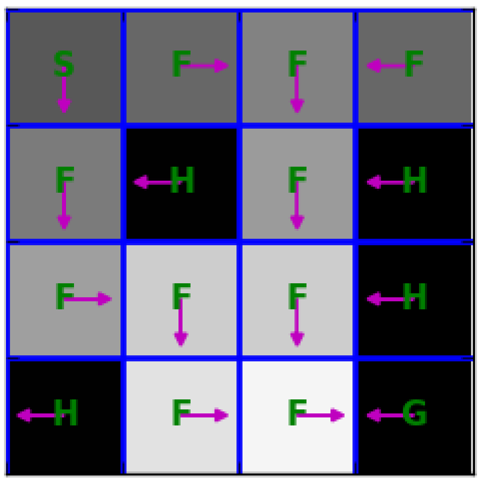

S F F F
F H F H
F F F H
H F F G


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

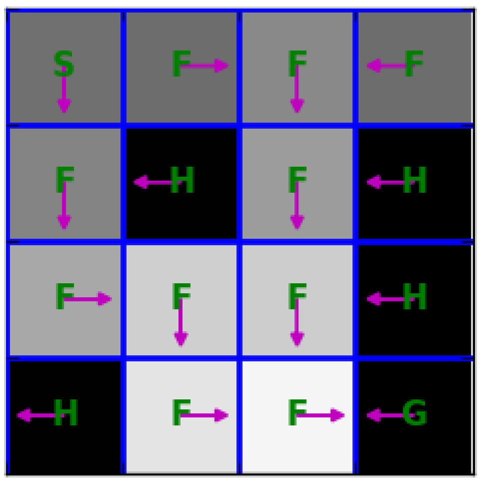

S F F F
F H F H
F F F H
H F F G



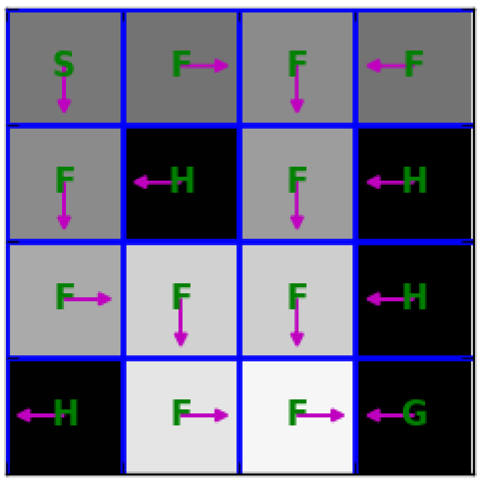
S
F
F
F
F
H
F
H
F
F
F
H
H
F
F
G



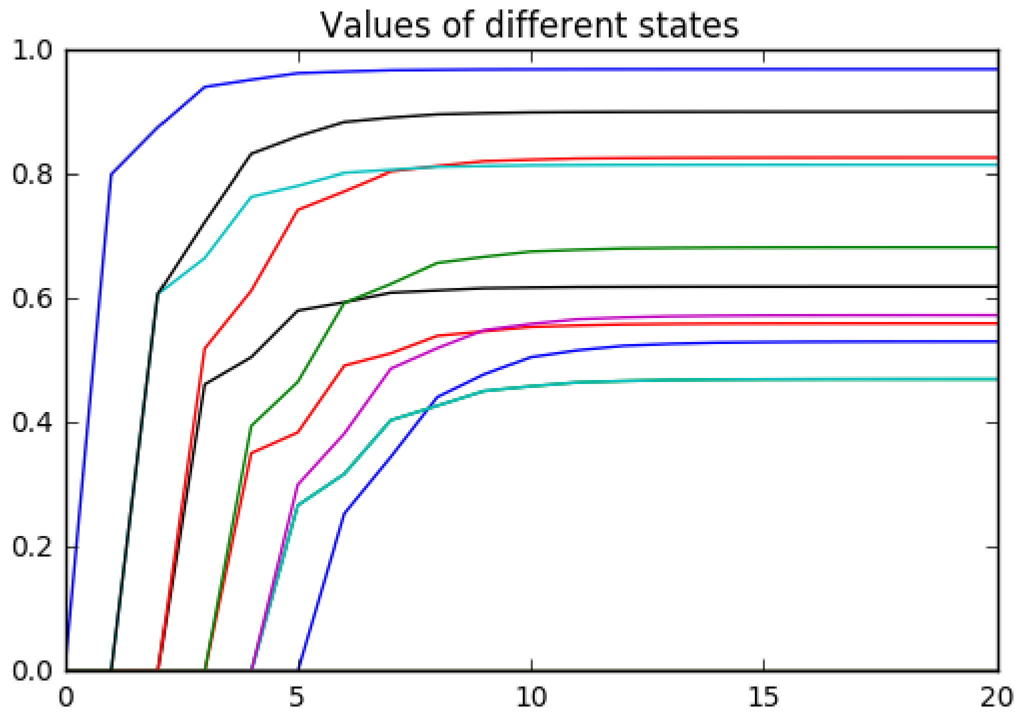
S
F
F
F
F
H
F
H
F
F
F
H
H
F
F
G



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



Values of different states
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20

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 = 70
# YOUR CODE HERE
# Your code will need to define an MDP (mymdp)
# like the frozen lake MDP defined above
# (probability, nextstate, reward)
mymdp = MDP( {
    0: {0: [(1, 1, 1), (0, 0, 0), (2,0,0)],
        1: [(1, 2, 0), (0,0,0), (0,1,0)]},
    1: {0: [(1,0, -100000000000000), (0,1,0), (0,2,0)],
        1: [(1, 2, 0), (0, 1, 0), (0,0,0)]},
    2: {0: [(1, 2, 0), (0, 0, 0), (0,1,0)],
        1: [(1, 1, 0), (0,0,0), (0,2,0)]}
    }, 3, 2)
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          | 1.000
   1      | 1.90000      |    0          | 2.900
   2      | 3.61000      |    0          | 6.510
   3      | 6.85900      |    0          | 13.369
   4      | 13.03210      |    0          | 26.401
   5      | 24.76099      |    0          | 51.162
   6      | 47.04588      |    0          | 98.208
   7      | 89.38717      |    0          | 187.595
   8      | 169.83563      |    0          | 357.431
   9      | 322.68770      |    0          | 680.118
  10      | 613.10663      |    0          | 1293.225
  11      | 1164.90259      |    0          | 2458.128
  12      | 2213.31492      |    0          | 4671.443
  13      | 4205.29835      |    0          | 8876.741
  14      | 7990.06686      |    0          | 16866.808
  15      | 15181.12703      |    0          | 32047.935
  16      | 28844.14136      |    0          | 60892.076
  17      | 54803.86858      |    0          | 115695.945
  18      | 104127.35030      |    0          | 219823.295
  19      | 197841.96557      |    0          | 417665.261
  20      | 375899.73458      |    0          | 793564.995
  21      | 714209.49569      |    0          | 1507774.491
  22      | 1356998.04182      |    0          | 2864772.533
  23      | 2578296.27945      |    0          | 5443068.812
  24      | 4898762.93096      |    0          | 10341831.743
  25      | 9307649.56883      |    0          | 19649481.312
  26      | 17684534.18077      |    0          | 37334015.493
  27      | 33600614.94346      |    0          | 70934630.436
  28      | 63841168.39257      |    0          | 134775798.829
  29      | 121298219.94589      |    0          | 256074018.775
  30      | 230466617.89719      |    0          | 486540636.672
  31      | 437886574.00467      |    0          | 924427210.677
  32      | 831984490.60887      |    0          | 1756411701.285
  33      | 1580770532.15686      |    0          | 3337182233.442
  34      | 3003464011.09803      |    0          | 6340646244.540
  35      | 5706581621.08626      |    0          | 12047227865.627
  36      | 10842505080.06390      |    0          | 22889732945.690
  37      | 20600759652.12141      |    0          | 43490492597.812
  38      | 39141443339.03068      |    0          | 82631935936.843
  39      | 74368742344.15828      |    0          | 157000678281.001
  40      | 141300610453.90073      |    0          | 298301288734.902
  41      | 268471159862.41132      |    0          | 566772448597.313
  42      | 510095203738.58154      |    0          | 1076867652335.894
  43      | 969180887103.30481      |    0          | 2046048539439.199
  44      | 1841443685496.27930      |    0          | 3887492224935.479
  45      | 3498743002442.93066      |    0          | 7386235227378.409
  46      | 6647611704641.56738      |    0          | 14033846932019.977
  47      | 12630462238818.97656      |    0          | 26664309170838.953
  48      | 23997878253756.05469      |    0          | 50662187424595.008
  49      | 45595968682136.50781      |    0          | 96258156106731.516
  50      | 86632340496059.35938      |    0          | 182890496602790.875
  51      | 164601446942512.75000      |    1          | 347491943545303.625
  52      | 382801422374793.00000      |    1          | 730293365920096.625
  53      | 875875508377724.37500      |    0          | 1606168874297821.000
  54      | 2009641749610927.00000      |    0          | 3615810623908748.000
  55      | 4608796970571658.00000      |    0          | 8224607594480406.000
  56      | 10570415923110010.00000      |    0          | 18795023517590416.000
  57      | 24243229519849936.00000      |    0          | 43038253037440352.000
  58      | 55601936458321664.00000      |    0          | 98640189495762016.000
  59      | 127523193912475712.00000      |    0          | 226163383408237728.000
  60      | 292474816087339136.00000      |    0          | 518638199495576832.000
  61      | 670791833071953664.00000      |    0          | 1189430032567530496.000
  62      | 1538463004355535360.00000      |    0          | 2727893036923065856.000
  63      | 3528469337622955520.00000      |    0          | 6256362374546021376.000
  64      | 8092554602914485248.00000      |    0          | 14348916977460506624.000
  65      | 18560297322742239232.00000      |    0          | 32909214300202745856.000
  66      | 42568095442340577280.00000      |    0          | 75477309742543323136.000
  67      | 97630049674221961216.00000      |    0          | 173107359416765284352.000
  68      | 223914800517734039552.00000      |    0          | 397022159934499323904.000
  69      | 513549240814680014848.00000      |    0          | 910571400749179338752.000
  70      | 1177826665015147102208.00000      |    0          | 2088398065764326440960.000
```

Problem 3: Policy Iteration

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

Initialize π_0

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

- 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`.)

Solution:

We'll manually calculate the resultant. Let P be the matrix composed of rows of $p_s = [P(s, \pi(s), \hat{s})]_{\hat{s} \in S}$. Likewise, let R be the matrix composed of rows $r_s = [R(s, \pi(s), \hat{s})]$. Then

In [153]:
```python
def compute_vpi(pi, mdp, gamma):
    P = []
    R = []

    # Make the probability and reward matrices.
    for s in range(mdp.nS):
        prob, sprime, r = zip(*mdp.P[s][pi[s]])
        prob, sprime, r = list(prob), list(sprime), list(r)
        ps = np.zeros(mdp.nS)
        rs = np.zeros(mdp.nS)
        ps[sprime] = prob
        rs[sprime] = r
        P += [ps]
        R += [rs]

    P = np.array(P)
    R = np.array(R)

    # Creating the linear system
    b = - np.einsum('xy,xy->x', P,R)
    A = P*gamma - np.identity(mdp.nS)
    V = np.linalg.solve(A,b)

    return V
```

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

In [154]:
```python
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 [155]:
```python
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 [ 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 [ 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 [  9.580e-04   3.839e-04   2.254e-04   3.839e-04   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 [156]:

```
def compute_qpi(vpi, mdp,  gamma):
    Qpi = np.zeros((mdp.nS, mdp.nA))

    # Make the probability and reward matrices.
    for s in range(mdp.nS):
        for a in range(mdp.nA):
            prob, sprime, r = zip(*mdp.P[s][a])
            prob, sprime, r = np.array(prob), list(sprime), np.array(r)
            Qpi[s,a] = np.dot(prob,(r + gamma*vpi[sprime]))


    return Qpi

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

```
Qpi:
 [[  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 ]]
```

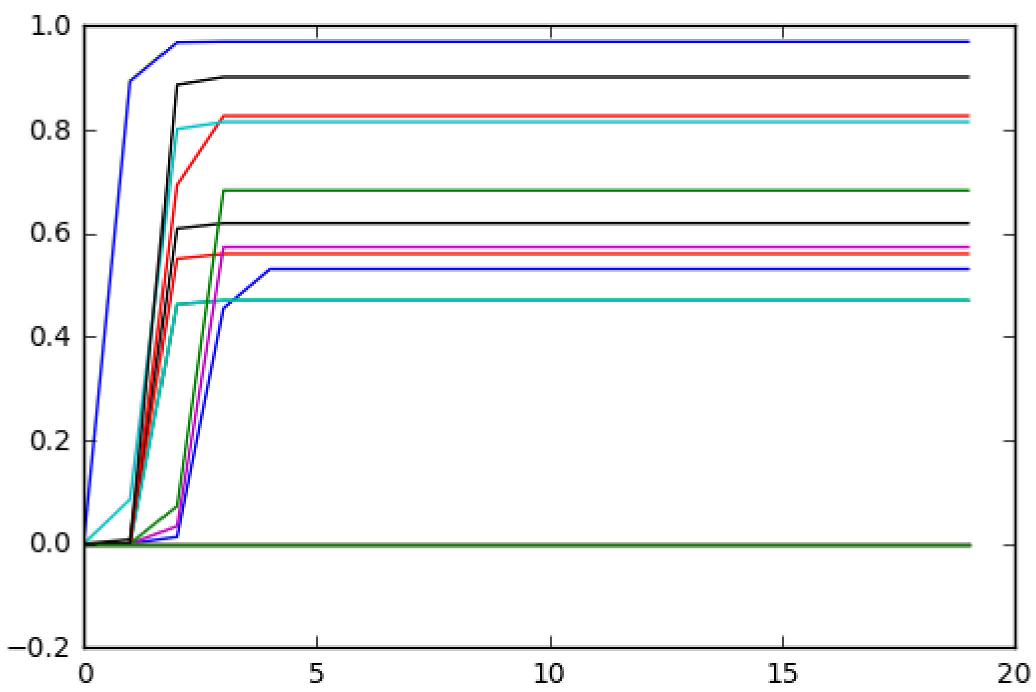
Now we're ready to run policy iteration!

NOTE TO INSTRUCTORS:

I believe there is an error with the following code on Github; that is, all of the ouput for the Q matrix is equivalent but the change in actions differs slightly from the github solution.

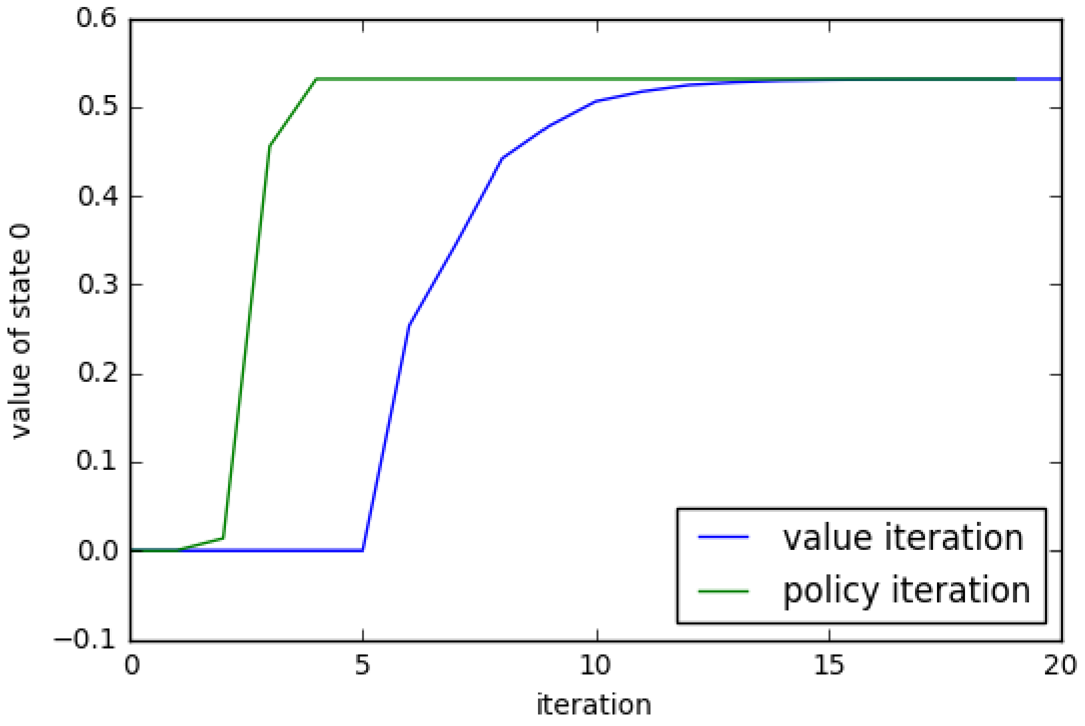
In [158]:
```
def policy_iteration(mdp, gamma, nIt):
    Vs = []
    pis = []
    pi_prev = np.zeros(mdp.nS,dtype='int')
    pis.append(pi_prev)
    print("Iteration | # chg actions | V[0]")
    print("----------+---------------+---------")
    for it in range(nIt):
        vpi = compute_vpi(pi_prev, 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);
```

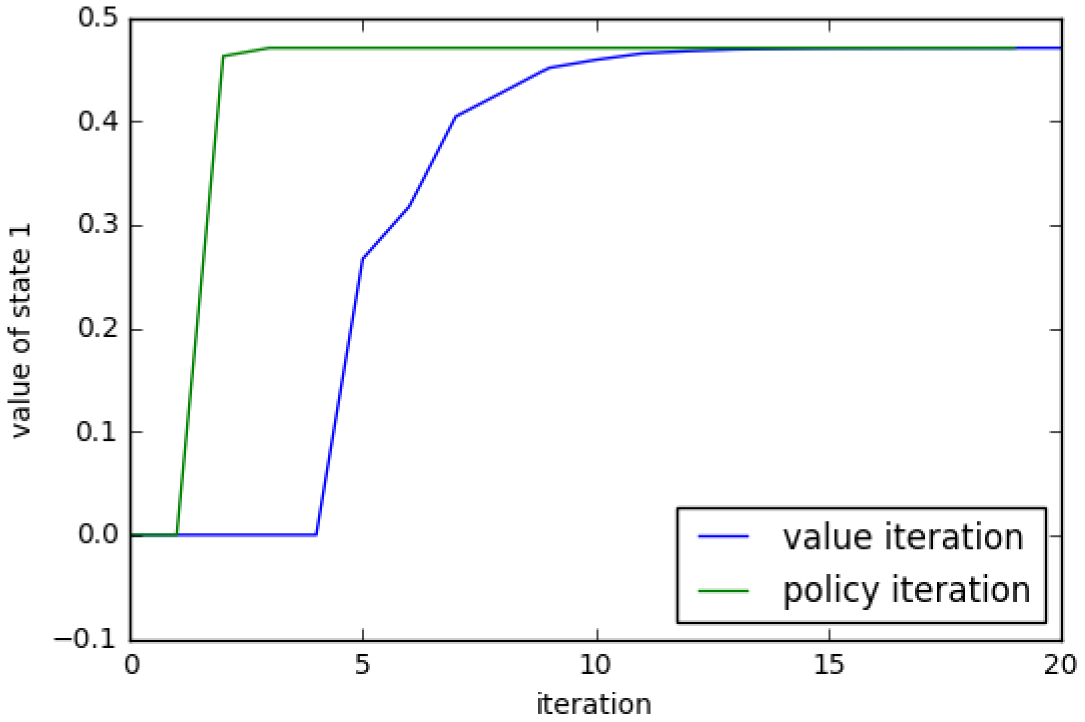
```
Iteration | # chg actions | V[0]
----------+---------------+---------
   0      |      1        | 0.00000
   1      |      7        | -0.00000
   2      |      4        | 0.01352
   3      |      1        | 0.45546
   4      |      0        | 0.53118
   5      |      0        | 0.53118
   6      |      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
```

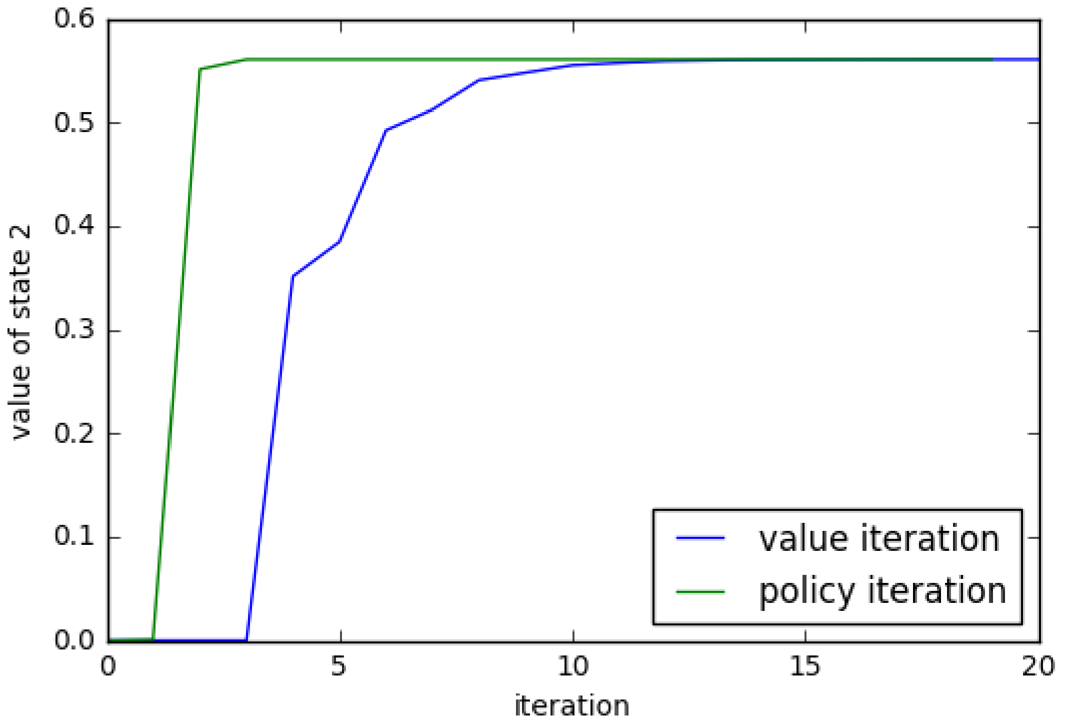


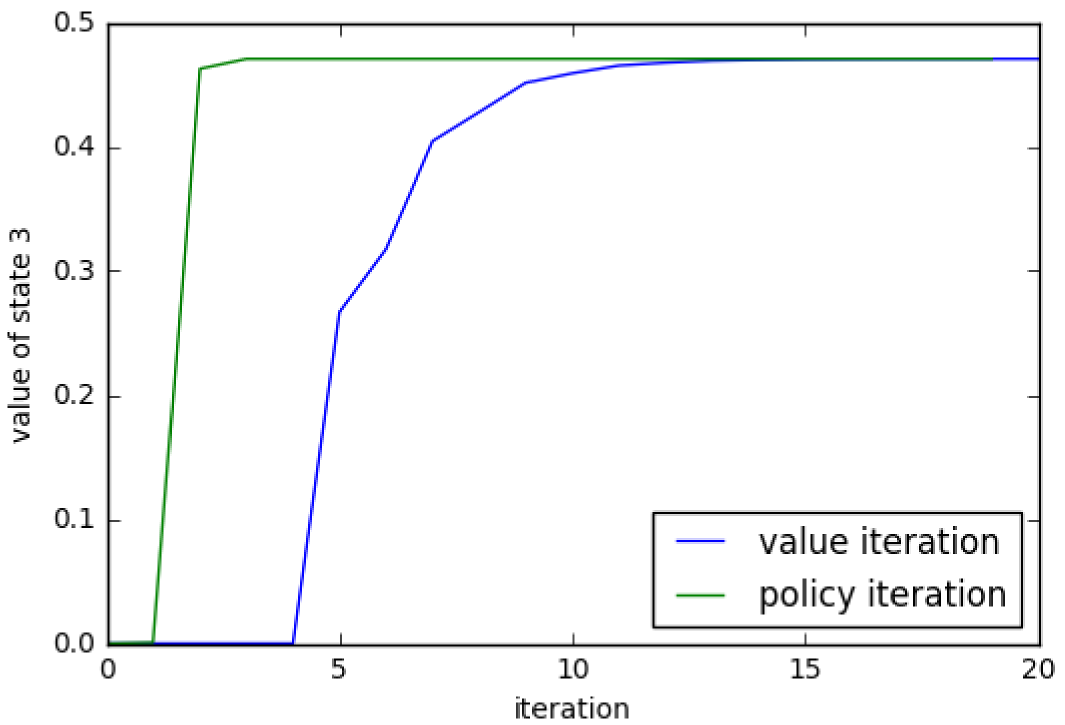
Now we can compare the convergence of value iteration and policy iteration on several states. For fun, you can try adding modified policy iteration.

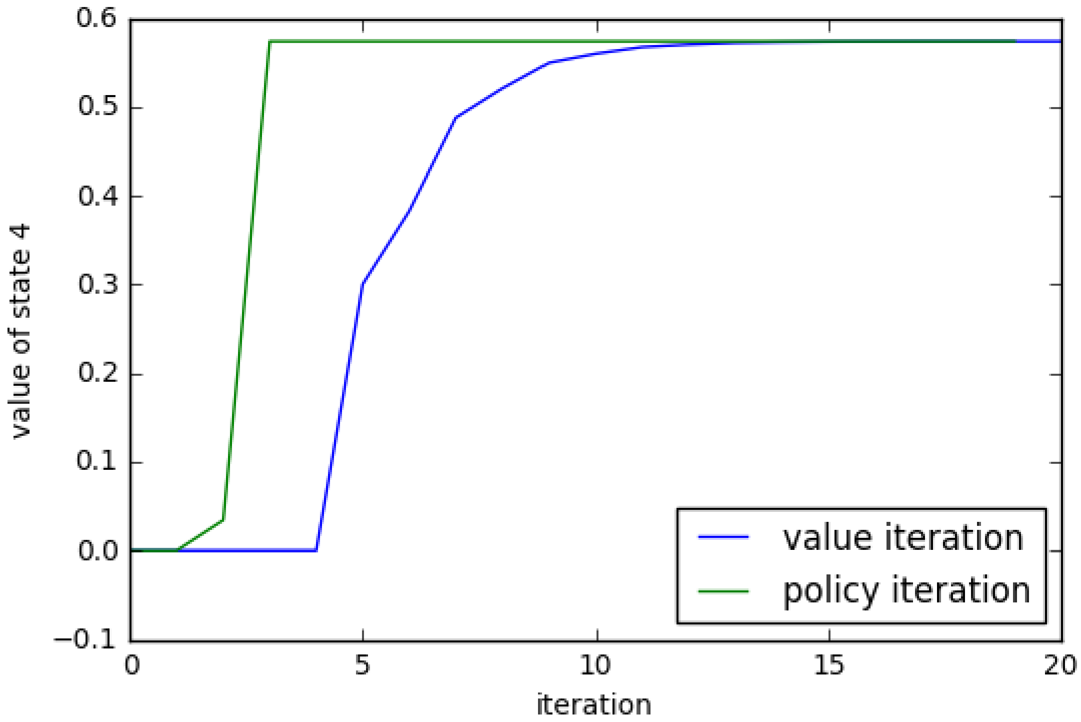
In [159]:
```
for s in range(5):
    plt.figure()
    plt.plot(np.array(Vs_VI)[:,s])
    plt.plot(np.array(Vs_PI)[:,s])
    plt.ylabel("value of state %i"%s)
    plt.xlabel("iteration")
    plt.legend(["value iteration", "policy iteration"], loc='best')
```


value of state 0
iteration
value iteration
policy iteration


value of state 1
iteration
value iteration
policy iteration


value of state 2
iteration
value iteration
policy iteration


value of state 3
iteration
value iteration
policy iteration


value of state 4
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

```
In [ ]:
```