lrl

Release 1.0.0

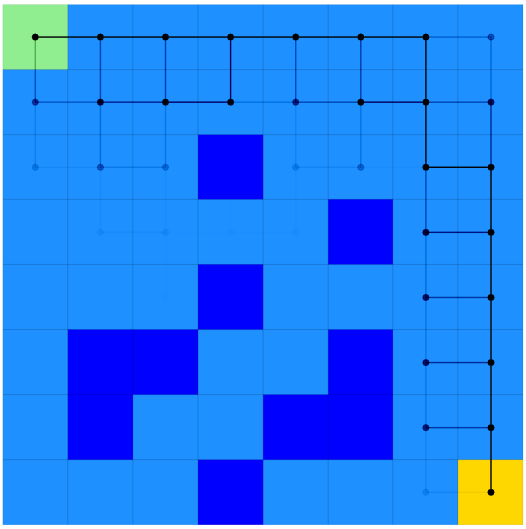
Andrew Scribner

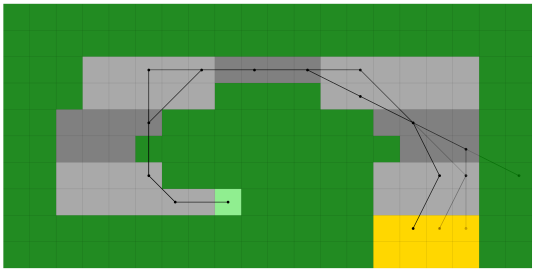
Sep 28, 2019

CONTENTS

lrl is a Python package for applying (and hopefully, learning!) basic Reinforcement Learning algorithms. It is intended to be an early stepping stone for someone trying to understanding the basic concepts of planning and learning, providing out-of-the-box implementations of some simple environments and algorithms in a well documented, readable, and digestible way to give someone platform from which to build understanding.

Within minutes, you'll be able to make fun images of your agent exploring an environment like these!





The source code was written by Andrew Scribner and is available on GitHub.

CHAPTER

ONE

GENERAL

1.1 Overview

lrl is a Python package for applying (and hopefully, learning!) basic Reinforcement Learning algorithms. It is intended to be an early stepping stone for someone trying to understanding the basic concepts of planning and learning, providing out-of-the-box implementations of some simple environments and algorithms in a well documented, readable, and digestible way to give someone platform from which to build understanding.

The overall goal of the author in writing this package was to provide people interested in Reinforcement Learning a starting point and handrail to help them as they began learning. The fastest, most efficient implementation is less important here than code which can be read and learned from by someone new to the topic and with intermediate Python skills.

The source code was written by Andrew Scribner and is available on GitHub.

1.2 Content

Implemented here are two common planning algorithms and one common learning algorithm:

- Value Iteration
- Policy Iteration
- Q-Learning

All three are implemented with a common API so that they can easily be run and compared to one another.

Also provided here for solving are two gridworld environments:

- RewardingFrozenLake
- Racetrack

RewardingFrozenLake is a cautionary tale about why you shouldn't throw a frisbee near a partially frozen lake. The goal of the environment is to traverse a partially frozen lake from a starting position (where you are) to a goal position (where your frisbee is located) without falling through any of the holes in the ice. The state space in this environment is integer (x, y) position and actions are movement (up, down, left, right), but as the ice surface is... well... ice, movements are not always as expected (an action moving in one direction may result in you going in a direction 90 degrees from your choice). The implementation used here is a a slightly modified version of this, which is itself a moderately modified version of this.

Racetrack is a simulation of a car driving on a track from one or more starting positions to one or more finish positions, inspired by Sutton and Barto's Reinforcement Learning (exercise 5.8) and coded by the present author. It incorporates a more complicated state function of integer (x, y) position and velocity, with actions being vehicle acceleration. Stochasticity is introduced through oily grid tiles, and failure through terminal grass tiles surrounding tracks.

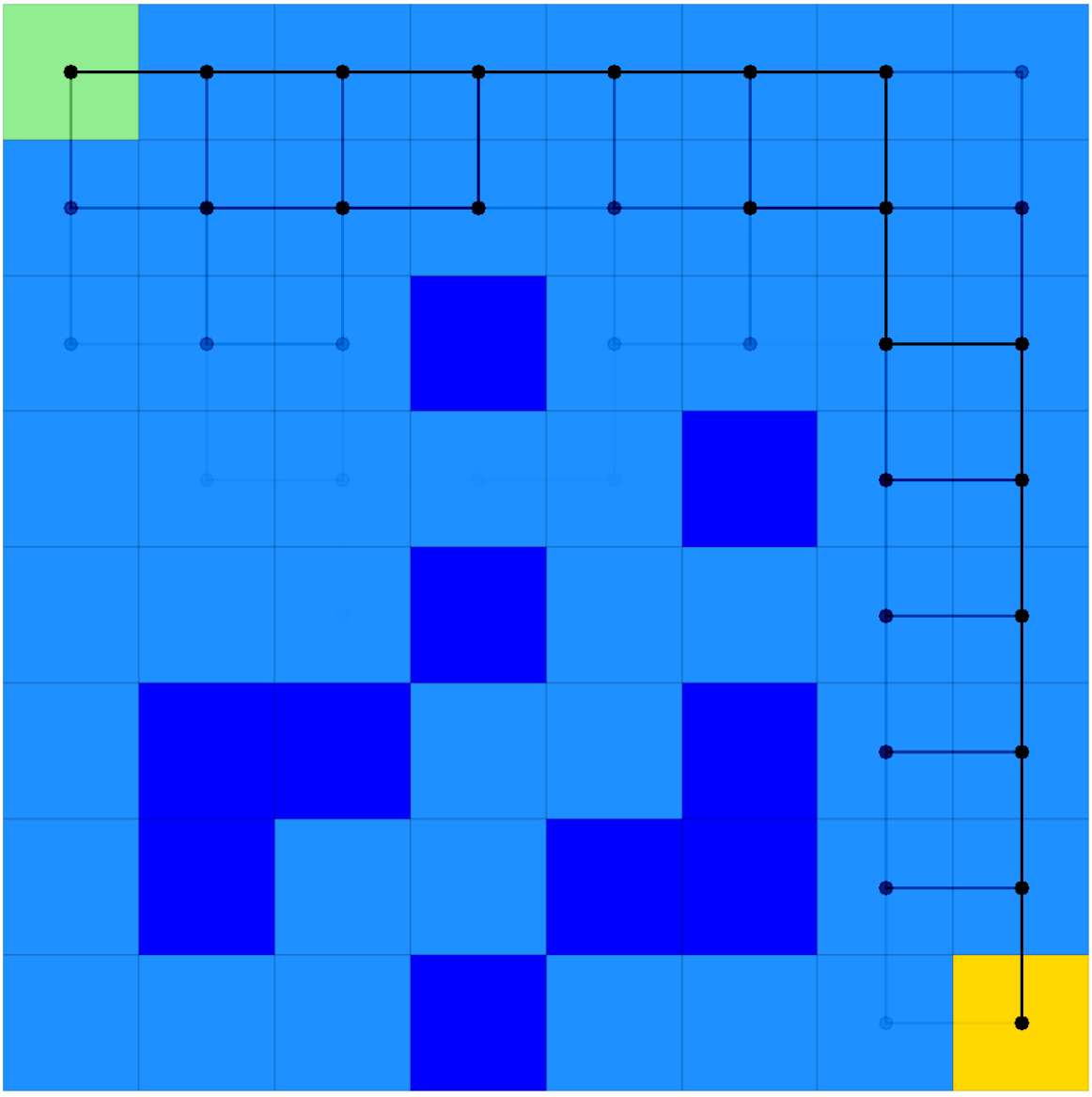


Fig. 1: Example application of an optimal solution to a RewardingFrozenLake map, where the paths obtained are from 100 episodes in the environment following the optimal solution (stochasticity causes many paths to be explored)

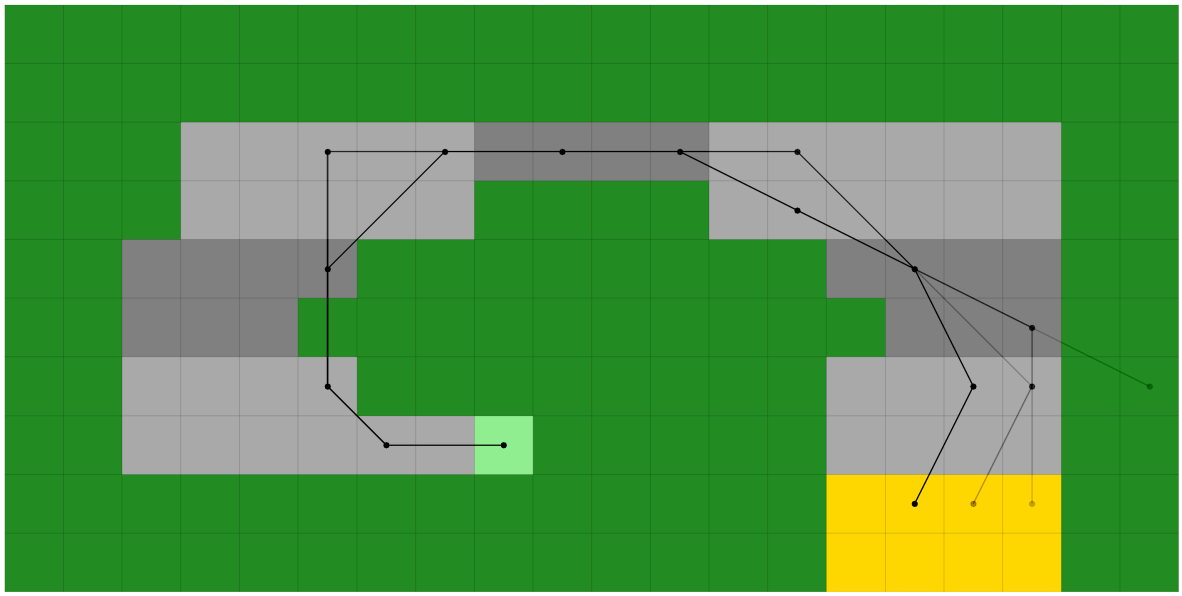


Fig. 2: Example application of an optimal solution to a Racetrack map, where the paths obtained are from 100 episodes in the environment following the optimal solution (stochasticity causes many paths to be explored)

A non-exhaustive set of plotting scripts to display the calculated solutions to the environments is also included, as well as usage examples. The hope is that users will apply the package as written, realize they want to dig deeper into a particular aspect or add some feature they read about in literature, and then add that to the existing codebase.

1.3 Installation Instructions

lrl is accessible using pip

```
` pip install lrl `
```

or, you can pull the source from GitHub to your working directory so you can play along at home

```
` git clone https://github.com/ca-scribner/lrl.git lrl pip install -e lrl `
```

1.4 Acknowledgements and Contributions

Significance guidance and inspiration was taken from two related previous codebases (here, and here, with apologies for access to the second link being restricted). Although the present codebase is nearly original (with exceptions being cited directly in the codebase), the above works were instrumental in inspiring this package and in many ways this package should be seen as an incremental work off these previous projects.

Future contributions are encouraged and hoped for. If you find a bug, build a new environment, implement a new feature, or have a great example that would help others then please contribute. The only requirement for contributions is that they should be well commented and documented (using formatting that Sphinx will understand for the docs, etc.).

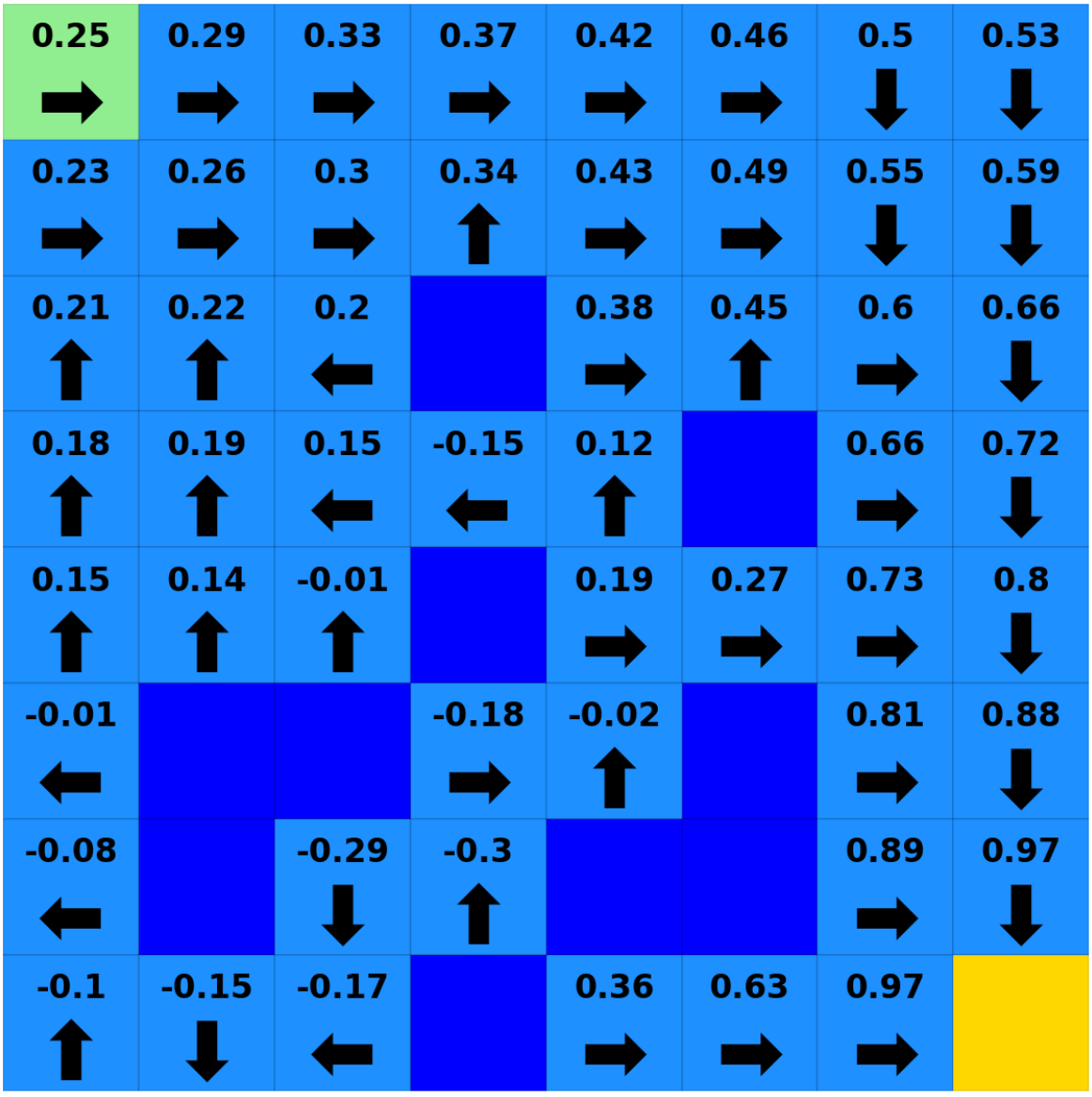


Fig. 3: Example solution to RewardingFrozenLake, where numbers show the value of each location and arrows show the optimal policy

CHAPTER

TWO

TUTORIALS

See the below examples, plus **look at the examples directory in the package** for an example of how to run many cases in batch mode (for running parameter searches, etc.) and for copies of the below examples as notebooks.

2.1 Example Case using Racetrack

Below is an example of how to initialize the Racetrack environment and solve/compare with multiple solvers.

2.1.1 Boilerplate

If you're playing with things under the hood as you run these, autoreload is always useful...

[1]:
```
%load_ext autoreload

%autoreload 2
```

If necessary, add directory containing lrl to path (workaround for if lrl is not installed as a package)

[2]:
```
import sys

# Path to directory containing lrl
sys.path.append('../')
```

[3]:
```
from lrl import environments, solvers
from lrl.utils import plotting

import matplotlib.pyplot as plt
```

Logging is used throughout lrl for basic info and debugging.

[4]:
```
import logging
logging.basicConfig(format='%(asctime)s - %(name)s - %(funcName)s - %(levelname)s -
↪%(message)s',
                    level=logging.INFO, datefmt='%H:%M:%S')
logger = logging.getLogger(__name__)
```

2.1.2 Initialize an Environment

Initialize the 20x10 racetrack that includes some oily (stochastic) surfaces.

Note: Make sure that your velocity limits suit your track. A track must have a grass padding around the entire course that prevents a car from trying to exit the track entirely, so if max(abs(vel))==3, you need 3 grass tiles around the outside perimeter of your map. For track, we have a 2-tile perimeter so velocity must be less than +-2.

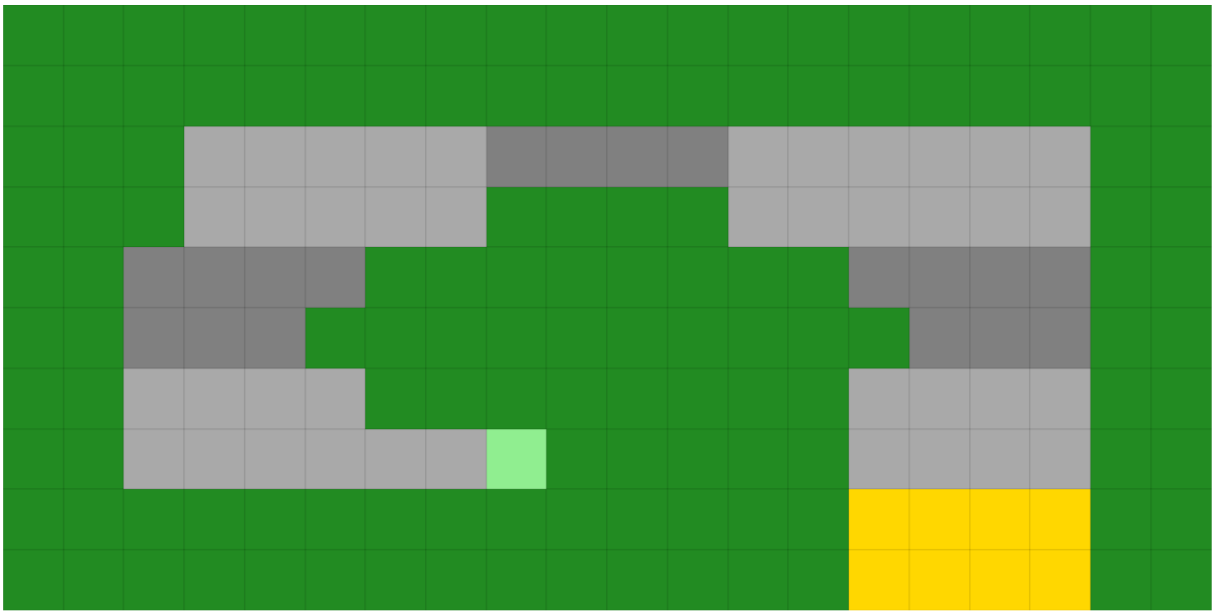
[5]:
```
# This will raise an error due to x_vel max limit
try:
    rt = environments.get_racetrack(track='20x10_U',
                                    x_vel_limits=(-2, 20),  # Note high x_vel upper␣
↪limit
                                    y_vel_limits=(-2, 2),
                                    x_accel_limits=(-2, 2),
                                    y_accel_limits=(-2, 2),
                                    max_total_accel=2,
                                    )
except IndexError as e:
    print("Caught the following error while building a track that shouldn't work:")
    print(e)
    print("")

# This will work
try:
    rt = environments.get_racetrack(track='20x10_U',
                                    x_vel_limits=(-2, 2),
                                    y_vel_limits=(-2, 2),
                                    x_accel_limits=(-2, 2),
                                    y_accel_limits=(-2, 2),
                                    max_total_accel=2,
                                    )
    print("But second track built perfectly!")
except:
    print("Something went wrong, we shouldn't be here")
```

```
Caught the following error while building a track that shouldn't work:
Caught IndexError while building Racetrack.  Likely cause is a max velocity that is␣
↪creater than the wall padding around the track (leading to a car that can exit the␣
↪track entirely)

But second track built perfectly!
```

Take a look at the track using plot_env

[6]:
```
plotting.plot_env(env=rt)
```

[6]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1f32321ae48>
```



There are also additional maps available - see the racetrack code base for more

[7]:
```
print(f'Available tracks: {list(environments.racetrack.TRACKS.keys())}')
```

```
Available tracks: ['3x4_basic', '5x4_basic', '10x10', '10x10_basic', '10x10_all_oil',␣
↪'15x15_basic', '20x20_basic', '20x20_all_oil', '30x30_basic', '20x10_U_all_oil',␣
↪'20x10_U', '10x10_oil', '20x15_risky']
```

Tracks are simply lists of strings using a specific set of characters. See the racetrack code for more detail on how to make your own

[8]:
```
for line in environments.racetrack.TRACKS['20x10_U']:
    print(line)
```

```
GGGGGGGGGGGGGGGGGGGG
GGGGGGGGGGGGGGGGGGGG
GGG      OOOO       GG
GGG      GGGG       GG
GGOOOOGGGGGGGGOOOOGG
GGOOOGGGGGGGGGGOOOGG
GG     GGGGGGGG     GG
GG       SGGGGG     GG
GGGGGGGGGGGGGGFFFFGG
GGGGGGGGGGGGGGFFFFGG
```

We can draw them using character art! For example, here is a custom track with more oil and a different shape than above...

[9]:
```
custom_track = \
"""GGGGGGGGGGGGGGGGGGGG
GGGGGGGGGGGGGGGGGGGG
GGGOOOOOOOOOOOOOOOGG
GGG      GGGG       GG
GG     GGGGGGGG     GG
GGOOOOOOSGGGGGOOOOGG
```

(continues on next page)

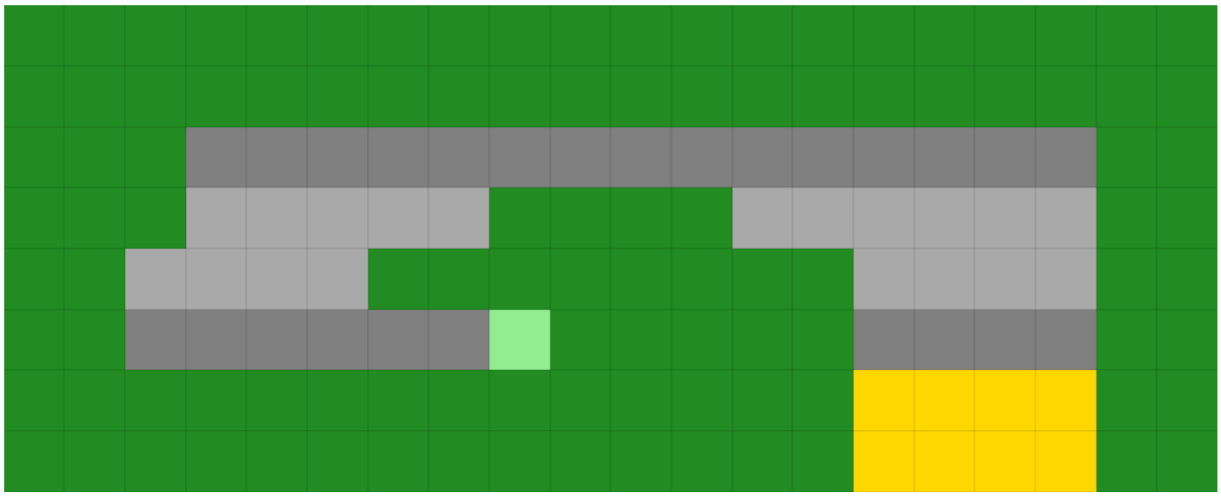
(continued from previous page)

```
GGGGGGGGGGGGGGFFFFGG
GGGGGGGGGGGGGGFFFFGG"""
custom_track = custom_track.split('\n')
```

[10]:
```
rt_custom = environments.get_racetrack(track=custom_track,
                                       x_vel_limits=(-2, 2),
                                       y_vel_limits=(-2, 2),
                                       x_accel_limits=(-2, 2),
                                       y_accel_limits=(-2, 2),
                                       max_total_accel=2,
                                       )
```

[11]:
```
plotting.plot_env(env=rt_custom)
```

[11]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1f325f99ac8>
```



2.1.3 Solve with Value Iteration and Interrogate Solution

[12]:
```
rt_vi = solvers.ValueIteration(env=rt)
rt_vi.iterate_to_convergence()
```

```
16:33:21 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Solver iterating␣
↪to convergence (Max delta in value function < 0.001 or iters>500)
16:33:23 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Solver converged␣
↪to solution in 18 iterations
```

And we can then score our solution by running it multiple times through the environment

[13]:
```
scoring_data = rt_vi.score_policy(iters=500)
```

score_policy returns a EpisodeStatistics object that contains details from each episode taken during the scoring. Easiest way to interact with it is grabbing data as a dataframe

[14]:
```
print(f'type(scoring_data) = {type(scoring_data)}')
scoring_data_df = scoring_data.to_dataframe(include_episodes=True)
scoring_data_df.head(3)
```

```
type(scoring_data) = <class 'lrl.data_stores.data_stores.EpisodeStatistics'>
```

[14]:
```
   episode_index  reward  steps  terminal  reward_mean  reward_median  \
0              0    91.0     11      True         91.0           91.0
1              1    91.0     11      True         91.0           91.0
2              2    91.0     11      True         91.0           91.0

   reward_std  reward_min  reward_max  steps_mean  steps_median  steps_std  \
0         0.0        91.0        91.0        11.0          11.0        0.0
1         0.0        91.0        91.0        11.0          11.0        0.0
2         0.0        91.0        91.0        11.0          11.0        0.0

   steps_min  steps_max  terminal_fraction  \
0         11         11                1.0
1         11         11                1.0
2         11         11                1.0

                                        episodes
0  [(8, 2, 0, 0), (6, 2, -2, 0), (5, 3, -1, 1), (...
1  [(8, 2, 0, 0), (6, 2, -2, 0), (5, 3, -1, 1), (...
2  [(8, 2, 0, 0), (6, 2, -2, 0), (5, 3, -1, 1), (...
```

[15]:
```
scoring_data_df.tail(3)
```

[15]:
```
     episode_index  reward  steps  terminal  reward_mean  reward_median  \
497            497    91.0     11      True    90.473896           91.0
498            498    91.0     11      True    90.474950           91.0
499            499    91.0     11      True    90.476000           91.0

     reward_std  reward_min  reward_max  steps_mean  steps_median  steps_std  \
497    0.880581        89.0        91.0   11.526104          11.0   0.880581
498    0.880013        89.0        91.0   11.525050          11.0   0.880013
499    0.879445        89.0        91.0   11.524000          11.0   0.879445

     steps_min  steps_max  terminal_fraction  \
497         11         13                1.0
498         11         13                1.0
499         11         13                1.0

                                          episodes
497  [(8, 2, 0, 0), (6, 2, -2, 0), (5, 3, -1, 1), (...
498  [(8, 2, 0, 0), (6, 2, -2, 0), (5, 3, -1, 1), (...
499  [(8, 2, 0, 0), (6, 2, -2, 0), (5, 3, -1, 1), (...
```

Reward, Steps, and Terminal columns give data on that specific walk, whereas reward_mean, _median, etc. columns give aggregate scores up until that walk. For example:

[16]:
```
print(f'The reward obtained in the 499th episode was {scoring_data_df.loc[499, "reward
↪"]}')
print(f'The mean reward obtained in the 0-499th episodes (inclusive) was {scoring_
↪data_df.loc[499, "reward_mean"]}')
```

```
The reward obtained in the 499th episode was 91.0
The mean reward obtained in the 0-499th episodes (inclusive) was 90.476
```

And we can access the actual episode path for each episode

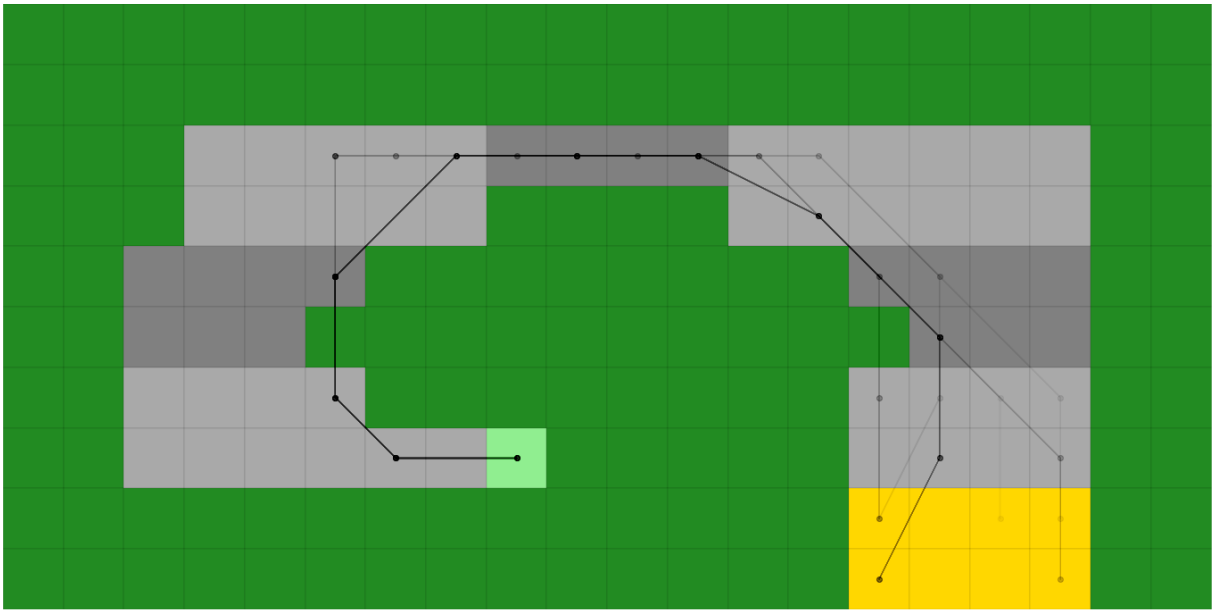
[17]:
```
print(f'Episode 0 (directly)          : {scoring_data.episodes[0]}')
print(f'Episode 0 (from the dataframe): {scoring_data_df.loc[0, "episodes"]}')
```

```
Episode 0 (directly)          : [(8, 2, 0, 0), (6, 2, -2, 0), (5, 3, -1, 1), (5, 5, 0,
↪ 2), (7, 7, 2, 2), (9, 7, 2, 0), (11, 7, 2, 0), (13, 6, 2, -1), (15, 4, 2, -2), (15,
↪ 2, 0, -2), (14, 0, -1, -2)]
Episode 0 (from the dataframe): [(8, 2, 0, 0), (6, 2, -2, 0), (5, 3, -1, 1), (5, 5, 0,
↪ 2), (7, 7, 2, 2), (9, 7, 2, 0), (11, 7, 2, 0), (13, 6, 2, -1), (15, 4, 2, -2), (15,
↪ 2, 0, -2), (14, 0, -1, -2)]
```

2.1.4 Plotting Results

And plot 100 randomly chosen episodes on the map, returned as a matplotlib axes

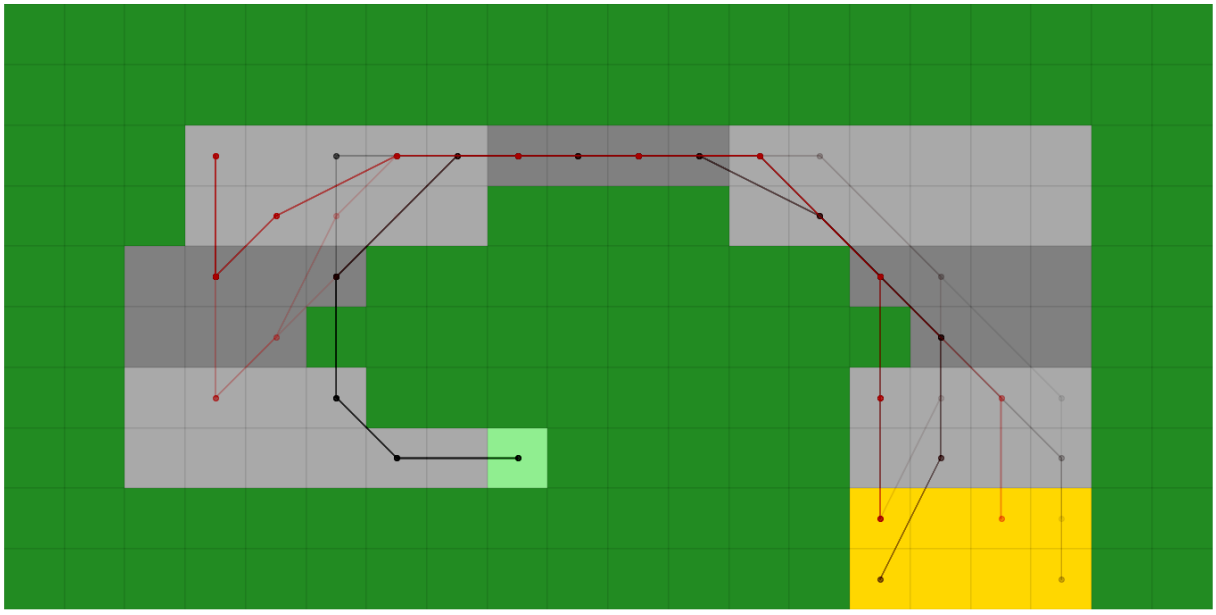
[18]:
```
ax_episodes = plotting.plot_episodes(episodes=scoring_data.episodes, env=rt, max_
↪episodes=100, )
```



score_policy also lets us use hypothetical scenario, such as what if we started in a different starting location. Let's try that by starting in the top left ((x,y) location (3, 7) (x=0 at left, y=0 at bot)) with a velocity of (1, -1), and plot it to our existing axes in red.

[19]:
```
scoring_data_alternate = rt_vi.score_policy(iters=500, initial_state=(3, 7, -2, -2))
ax_episodes_with_alternate = plotting.plot_episodes(episodes=scoring_data_alternate.
↪episodes, env=rt,
                                                    add_env_to_plot=False, color='r',␣
↪ax=ax_episodes
#                                                     savefig='my_figure_file',  # If␣
↪you wanted the figure to save directly
                                                                               # to␣
↪file, use savefig (used throughout
                                                                               # lrl
↪'s plotting scripts)
                                                    )

# Must get_figure because we're reusing the figure from above and jupyter wont␣
↪automatically reshow it
ax_episodes_with_alternate.get_figure()
```

[19]:



Where we can see the optimal policy takes a first action of (+2, 0) resulting in a first velocity of (0, -2) to avoid hitting the grass on step 1, then it redirects back up and around the track (although it sometimes slips on (3, 5) and drives down to (3, 3) before recovering)

We can also look at the Value function and best policy for all states

NOTE: Because state is (x, y, v_x, v_y), it is hard to capture all results on our (x, y) maps. plot_solver_results in this case plots a map for each (v_x, v_y) combination, with the axis title denoting which is the case.

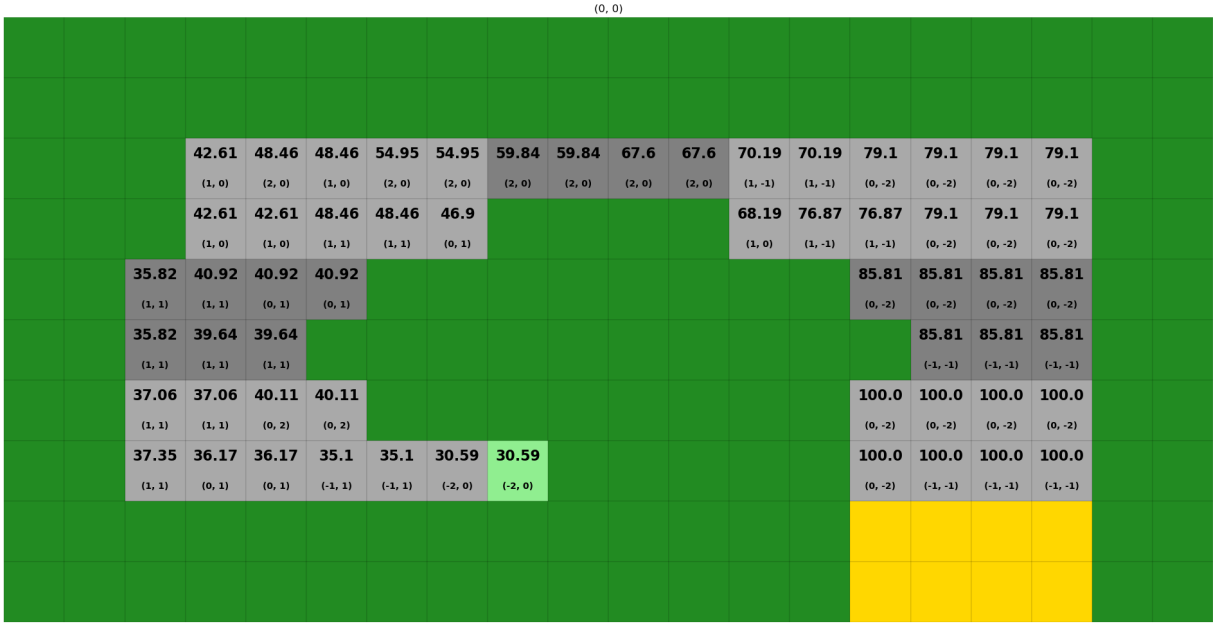
[20]:
```
# Sorry, this will plot 24 plots normally.
# To keep concise, we turn matplotlib inline plotting off then back on and␣
↪selectively plot a few examples
%matplotlib agg
ax_results = plotting.plot_solver_results(env=rt, solver=rt_vi)
%matplotlib inline
```

```
C:\Users\Scribs\Anaconda3\envs\lrl\lib\site-packages\matplotlib\pyplot.py:514:␣
↪RuntimeWarning: More than 20 figures have been opened. Figures created through the␣
↪pyplot interface (`matplotlib.pyplot.figure`) are retained until explicitly closed␣
↪and may consume too much memory. (To control this warning, see the rcParam `figure.
↪max_open_warning`).
  max_open_warning, RuntimeWarning)
```

Plots are indexed by the additional state variables (v_x, v_y), so we can see the policy for v_x=0, v_y=0 like:

[21]:
```
ax_results[(0, 0)].get_figure()
```

[21]:

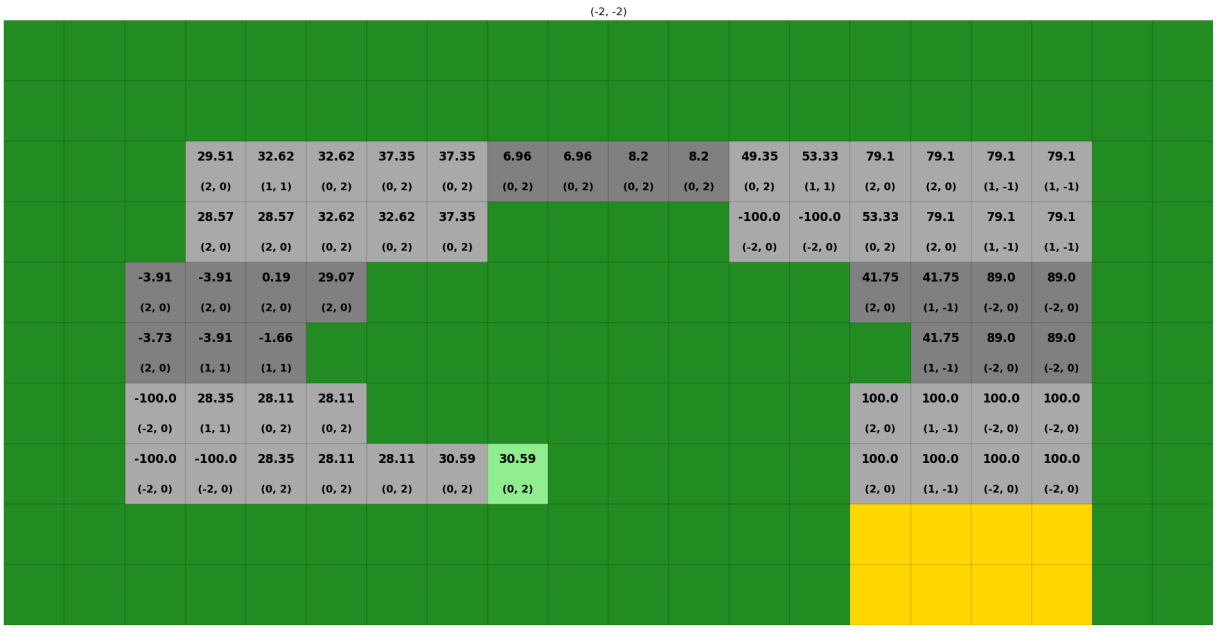


Where we can see the estimated value of the starting (light green) location is 30.59 (including discounting and costs for steps).

We can also look at the policy for (-2, -2) (the velocity for our alternate start used above in red):

[22]:
```
ax_results[(-2, -2)].get_figure()
```

[22]:



Where we in the top left tile (3, 7) the optimal policy is (2, 0), just as we saw in our red path plot above.

We can also see that tile (2, 2) (bottom left) has a -100 value because it is impossible to not crash with a (-2, -2) initialial velocity from tile (2, 2) given that our acceleration limit is 2.

2.1.5 Solving with Policy Iteration and Comparing to Value Iteration

We can also use other solvers

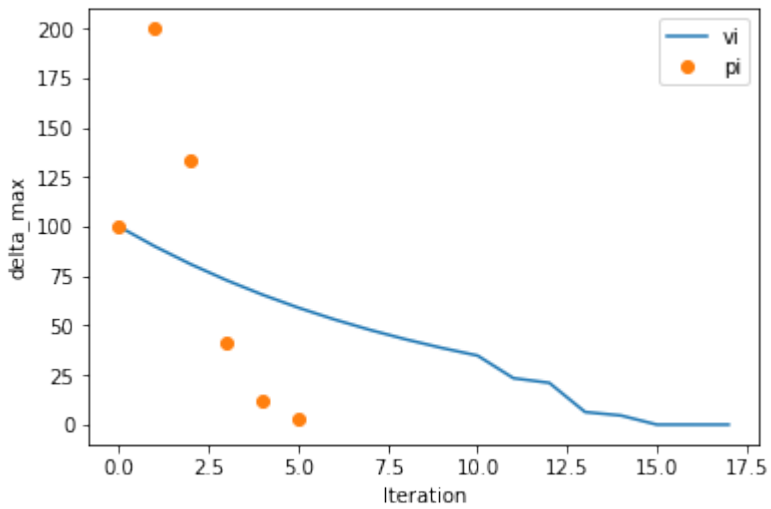
[23]:
```
rt_pi = solvers.PolicyIteration(env=rt)
rt_pi.iterate_to_convergence()
```

```
16:33:37 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Solver iterating␣
↪to convergence (1 iteration without change in policy or iters>500)
16:33:39 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Solver converged␣
↪to solution in 6 iterations
```

We can look at how PI and VI converged relative to each other, comparing the maximum change in value function for each iteration

[24]:
```
# (these are simple convenience functions for plotting, basically just recipes.  See␣
↪the plotting API)
# We can pass the solver..
ax = plotting.plot_solver_convergence(rt_vi, label='vi')

# Or going a little deeper into the API, with style being passed to matplotlib's plot␣
↪function...
ax = plotting.plot_solver_convergence_from_df(rt_pi.iteration_data.to_dataframe(), y=
↪'delta_max', x='iteration', ax=ax, label='pi', ls='', marker='o')

ax.legend()
```

[24]:
```
<matplotlib.legend.Legend at 0x1f33ad329e8>
```



And looking at policy changes per iteration

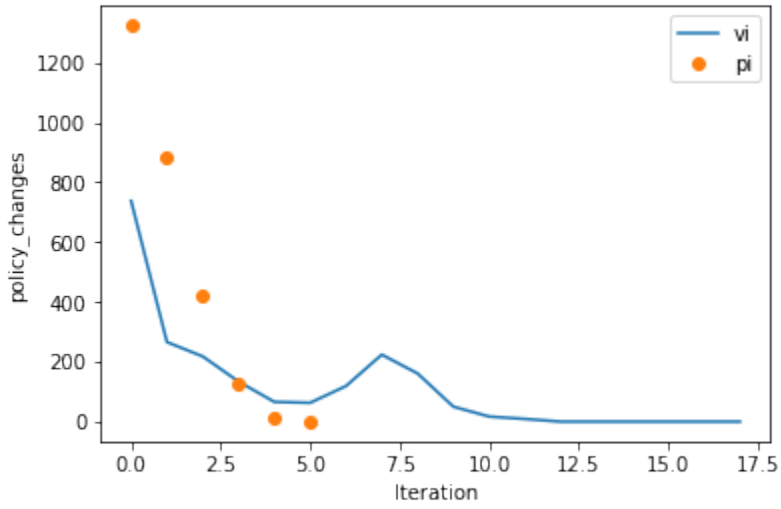
[25]:
```
# (these are simple convenience functions for plotting, basically just recipes.  See␣
↪the plotting API)
# We can pass the solver..
ax = plotting.plot_solver_convergence(rt_vi, y='policy_changes', label='vi')

# Or going a little deeper into the API...
ax = plotting.plot_solver_convergence_from_df(rt_pi.iteration_data.to_dataframe(), y=
↪'policy_changes', x='iteration', ax=ax, label='pi', ls='', marker='o')
```

(continues on next page)

(continued from previous page)

```
ax.legend()
```

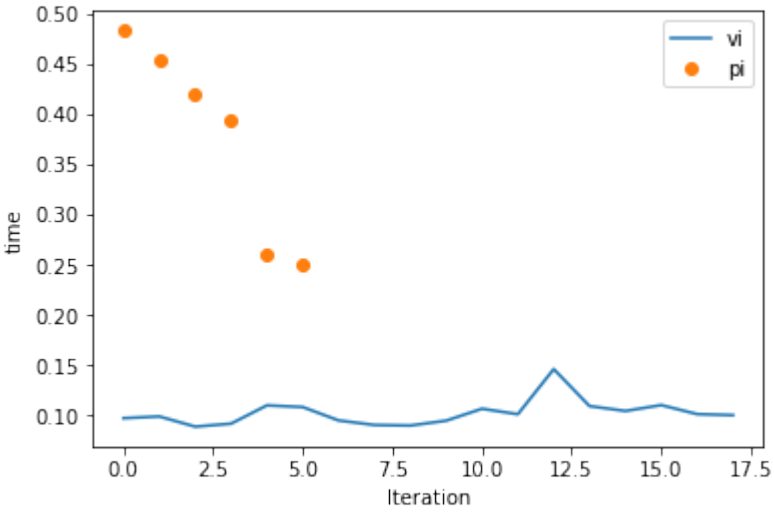
[25]:
```
<matplotlib.legend.Legend at 0x1f33be75780>
```



So we can see PI accomplishes more per iteration. But, is it faster? Let's look at time per iteration

[26]:
```
# (these are simple convenience functions for plotting, basically just recipes.  See␣
↪the plotting API)
# We can pass the solver..
ax = plotting.plot_solver_convergence(rt_vi, y='time', label='vi')

# Or going a little deeper into the API...
ax = plotting.plot_solver_convergence_from_df(rt_pi.iteration_data.to_dataframe(), y=
↪'time', x='iteration', ax=ax, label='pi', ls='', marker='o')

ax.legend()

print(f'Total solution time for Value Iteration (excludes any scoring time):  {rt_vi.
↪iteration_data.to_dataframe().loc[:, "time"].sum():.2f}s')
print(f'Total solution time for Policy Iteration (excludes any scoring time): {rt_pi.
↪iteration_data.to_dataframe().loc[:, "time"].sum():.2f}s')
```

```
Total solution time for Value Iteration (excludes any scoring time):  1.85s
Total solution time for Policy Iteration (excludes any scoring time): 2.26s
```



2.1.6 Solve with Q-Learning

We can also use QLearning, although it needs a few parameters

[27]:
```
# Let's be explicit with our QLearning settings for alpha and epsilon
alpha = 0.1  # Constant alpha during learning

# Decay function for epsilon (see QLearning() and decay_functions() in documentation␣
↪for syntax)
# Decay epsilon linearly from 0.2 at timestep (iteration) 0 to 0.05 at timestep 1500,
# keeping constant at 0.05 for ts>1500
epsilon = {
    'type': 'linear',
    'initial_value': 0.2,
    'initial_timestep': 0,
    'final_value': 0.05,
    'final_timestep': 1500
}

# Above PI/VI used the default gamma, but we will specify one here
gamma = 0.9

# Convergence is kinda tough to interpret automatically for Q-Learning.  One good way␣
↪to monitor convergence is to
# evaluate how good the greedy policy at a given point in the solution is and decide␣
↪if it is still improving.
# We can enable this with score_while_training (available for Value and Policy␣
↪Iteration as well)
# NOTE: During scoring runs, the solver is acting greedily and NOT learning from the␣
↪environment.  These are separate
#       runs solely used to estimate solution progress
# NOTE: Scoring every 50 iterations is probably a bit much, but used to show a nice␣
↪plot below.  The default 500/500
#       is probably a better general guidance
score_while_training = {
    'n_trains_per_eval': 50,  # Number of training episodes we run per attempt to␣
↪score the greedy policy
```

(continues on next page)

(continued from previous page)

```
                                # (eg: Here we do a scoring run after every 500␣
↪training episodes, where training episodes
                                # are the usual epsilon-greedy exploration episodes)
    'n_evals': 250,  # Number of times we run through the env with the greedy policy␣
↪whenever we score
}
# score_while_training = True  # This calls the default settings, which are also 500/
↪500 like above

rt_ql = solvers.QLearning(env=rt, alpha=alpha, epsilon=epsilon, gamma=gamma,
                          max_iters=5000, score_while_training=score_while_training)
```

(note how long Q-Learning takes for this environment versus the planning algorithms)

[28]:
```
rt_ql.iterate_to_convergence()
```

```
16:33:41 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Solver iterating␣
↪to convergence (20 episodes with max delta in Q function < 0.1 or iters>5000)
16:33:41 - lrl.solvers.learners - iterate - INFO - Performing iteration (episode) 0␣
↪of Q-Learning
16:33:41 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 50
16:33:42 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -103.0, r_max = -103.0
16:33:42 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 100
16:33:43 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -103.0, r_max = -103.0
16:33:43 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 150
16:33:43 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -103.0, r_max = -103.0
16:33:44 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 200
16:33:44 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -102.0, r_max = -102.0
16:33:44 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 250
16:33:45 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -103.424, r_max = -102.0
16:33:45 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 300
16:33:46 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -100.0, r_max = -100.0
16:33:46 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 350
16:33:46 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -103.0, r_max = -103.0
16:33:46 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 400
16:33:47 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -104.224, r_max = -104.0
16:33:47 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 450
16:33:47 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -103.0, r_max = -103.0
16:33:48 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 500
```

(continues on next page)

(continued from previous page)

```
16:33:48 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -104.22, r_max = -104.0
16:33:48 - lrl.solvers.learners - iterate - INFO - Performing iteration (episode) 500␣
↪of Q-Learning
16:33:48 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 550
16:33:49 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -104.264, r_max = -104.0
16:33:49 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 600
16:33:50 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -104.0, r_max = -104.0
16:33:50 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 650
16:33:50 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -104.0, r_max = -104.0
16:33:50 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 700
16:33:51 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -104.228, r_max = -104.0
16:33:51 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 750
16:33:52 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -106.0, r_max = -106.0
16:33:52 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 800
16:33:52 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -104.464, r_max = -104.0
16:33:53 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 850
16:33:53 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -105.0, r_max = -105.0
16:33:53 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 900
16:33:54 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -105.0, r_max = -105.0
16:33:54 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 950
16:33:54 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -106.824, r_max = -105.0
16:33:55 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1000
16:33:55 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -104.54, r_max = -104.0
16:33:55 - lrl.solvers.learners - iterate - INFO - Performing iteration (episode)␣
↪1000 of Q-Learning
16:33:55 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1050
16:33:56 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -104.0, r_max = -104.0
16:33:56 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1100
16:33:57 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -100.0, r_max = -100.0
16:33:57 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1150
16:33:57 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -103.796, r_max = -103.0
```

(continues on next page)

(continued from previous page)

```
16:33:58 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1200
16:33:58 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -106.128, r_max = -106.0
16:33:58 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1250
16:33:59 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -104.0, r_max = -104.0
16:33:59 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1300
16:34:00 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -107.852, r_max = -107.0
16:34:00 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1350
16:34:00 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -105.416, r_max = -104.0
16:34:01 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1400
16:34:01 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -100.0, r_max = -100.0
16:34:02 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1450
16:34:02 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -108.08, r_max = -108.0
16:34:02 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1500
16:34:03 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -105.84, r_max = -105.0
16:34:03 - lrl.solvers.learners - iterate - INFO - Performing iteration (episode)␣
↪1500 of Q-Learning
16:34:03 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1550
16:34:03 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -108.224, r_max = -108.0
16:34:04 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1600
16:34:04 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -104.0, r_max = -104.0
16:34:04 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1650
16:34:05 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -106.0, r_max = -106.0
16:34:05 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1700
16:34:06 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -100.0, r_max = -100.0
16:34:06 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1750
16:34:07 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -104.78, r_max = -104.0
16:34:07 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1800
16:34:07 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -112.76, r_max = -111.0
16:34:08 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1850
16:34:08 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -104.732, r_max = -104.0
```

(continues on next page)

(continued from previous page)

```
16:34:08 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1900
16:34:09 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -117.248, r_max = -117.0
16:34:09 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 1950
16:34:10 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -100.0, r_max = -100.0
16:34:10 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2000
16:34:10 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -106.0, r_max = -106.0
16:34:10 - lrl.solvers.learners - iterate - INFO - Performing iteration (episode)␣
↪2000 of Q-Learning
16:34:11 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2050
16:34:11 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -107.12, r_max = -100.0
16:34:12 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2100
16:34:12 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -115.0, r_max = -115.0
16:34:12 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2150
16:34:13 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -102.516, r_max = -100.0
16:34:13 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2200
16:34:14 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -105.72, r_max = -105.0
16:34:14 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2250
16:34:14 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -109.88, r_max = -109.0
16:34:15 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2300
16:34:15 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -105.496, r_max = -105.0
16:34:16 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2350
16:34:16 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -108.12, r_max = -104.0
16:34:16 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2400
16:34:17 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -107.168, r_max = -100.0
16:34:17 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2450
16:34:17 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -107.34, r_max = -106.0
16:34:18 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2500
16:34:18 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -105.568, r_max = -105.0
16:34:18 - lrl.solvers.learners - iterate - INFO - Performing iteration (episode)␣
↪2500 of Q-Learning
16:34:18 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2550
```

(continues on next page)

(continued from previous page)

```
16:34:19 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -106.0, r_max = -106.0
16:34:19 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2600
16:34:20 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -108.0, r_max = -108.0
16:34:20 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2650
16:34:21 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -111.952, r_max = -108.0
16:34:21 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2700
16:34:21 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -107.512, r_max = -106.0
16:34:22 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2750
16:34:22 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -120.848, r_max = -115.0
16:34:22 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2800
16:34:23 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -110.376, r_max = -110.0
16:34:23 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2850
16:34:24 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -112.776, r_max = -111.0
16:34:24 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2900
16:34:24 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -103.8, r_max = -100.0
16:34:25 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 2950
16:34:25 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -107.868, r_max = -106.0
16:34:26 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3000
16:34:26 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -101.668, r_max = -100.0
16:34:26 - lrl.solvers.learners - iterate - INFO - Performing iteration (episode)␣
↪3000 of Q-Learning
16:34:27 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3050
16:34:27 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -49.848, r_max = 89.0
16:34:27 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3100
16:34:28 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -101.392, r_max = -100.0
16:34:28 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3150
16:34:29 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -114.236, r_max = 77.0
16:34:29 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3200
16:34:30 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -106.996, r_max = -106.0
16:34:30 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3250
```

(continues on next page)

(continued from previous page)

```
16:34:30 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -124.144, r_max = -100.0
16:34:31 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3300
16:34:31 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -109.284, r_max = -106.0
16:34:32 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3350
16:34:32 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -102.112, r_max = -100.0
16:34:33 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3400
16:34:33 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -114.396, r_max = -105.0
16:34:34 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3450
16:34:34 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -100.0, r_max = -100.0
16:34:35 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3500
16:34:35 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -100.0, r_max = -100.0
16:34:35 - lrl.solvers.learners - iterate - INFO - Performing iteration (episode)␣
↪3500 of Q-Learning
16:34:36 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3550
16:34:36 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 36.04, r_max = 89.0
16:34:37 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3600
16:34:37 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -109.624, r_max = -107.0
16:34:37 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3650
16:34:38 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -87.048, r_max = 87.0
16:34:38 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3700
16:34:39 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.472, r_max = 91.0
16:34:39 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3750
16:34:40 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.536, r_max = 91.0
16:34:40 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3800
16:34:40 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.488, r_max = 91.0
16:34:41 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3850
16:34:41 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.416, r_max = 91.0
16:34:41 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3900
16:34:42 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.464, r_max = 91.0
16:34:42 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 3950
```

(continues on next page)

(continued from previous page)

```
16:34:42 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.392, r_max = 91.0
16:34:43 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4000
16:34:43 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.432, r_max = 91.0
16:34:43 - lrl.solvers.learners - iterate - INFO - Performing iteration (episode)␣
↪4000 of Q-Learning
16:34:44 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4050
16:34:44 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.52, r_max = 91.0
16:34:44 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4100
16:34:45 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.424, r_max = 91.0
16:34:45 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4150
16:34:45 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.4, r_max = 91.0
16:34:46 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4200
16:34:46 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.504, r_max = 91.0
16:34:46 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4250
16:34:47 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.544, r_max = 91.0
16:34:47 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4300
16:34:48 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.544, r_max = 91.0
16:34:48 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4350
16:34:48 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.424, r_max = 91.0
16:34:49 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4400
16:34:49 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.512, r_max = 91.0
16:34:49 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4450
16:34:50 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.552, r_max = 91.0
16:34:50 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4500
16:34:51 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.536, r_max = 91.0
16:34:51 - lrl.solvers.learners - iterate - INFO - Performing iteration (episode)␣
↪4500 of Q-Learning
16:34:51 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4550
16:34:51 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.544, r_max = 91.0
16:34:52 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4600
16:34:52 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.48, r_max = 91.0
```

(continues on next page)

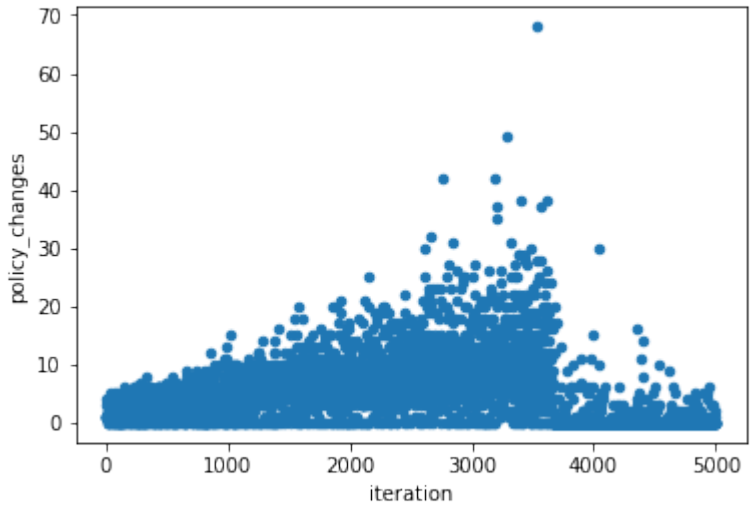
(continued from previous page)

```
16:34:52 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4650
16:34:53 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.504, r_max = 91.0
16:34:53 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4700
16:34:54 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.544, r_max = 91.0
16:34:54 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4750
16:34:54 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.472, r_max = 91.0
16:34:55 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4800
16:34:55 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.728, r_max = 91.0
16:34:55 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4850
16:34:56 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.52, r_max = 91.0
16:34:56 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4900
16:34:56 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.584, r_max = 91.0
16:34:57 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 4950
16:34:57 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.496, r_max = 91.0
16:34:57 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 5000
16:34:58 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 90.408, r_max = 91.0
16:34:58 - lrl.solvers.base_solver - iterate_to_convergence - WARNING - Max␣
↪iterations (5000) reached - solver did not converge
```

Like above, we can plot the number of policy changes per iteration. But this plot looks very different from above and shows one view of why Q-Learning takes many more iterations (each iteration accomplishes a lot less learning than a planning algorithm)

[29]:
```
rt_ql_iter_df = rt_ql.iteration_data.to_dataframe()
rt_ql_iter_df.plot(x='iteration', y='policy_changes', kind='scatter', )
```

[29]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1f33bf52cc0>
```



We can access the intermediate scoring through the scoring_summary (GeneralIterationData) and scoring_episode_statistics (EpisodeStatistics) objects

[30]:
```
rt_ql_intermediate_scoring_df = rt_ql.scoring_summary.to_dataframe()
rt_ql_intermediate_scoring_df
```

[30]:

	iteration	reward_mean	reward_median	reward_std	reward_min	reward_max	\
0	50	-103.000	-103.0	0.000000	-103.0	-103.0	
1	100	-103.000	-103.0	0.000000	-103.0	-103.0	
2	150	-103.000	-103.0	0.000000	-103.0	-103.0	
3	200	-102.000	-102.0	0.000000	-102.0	-102.0	
4	250	-103.424	-104.0	0.905662	-104.0	-102.0	
5	300	-100.000	-100.0	0.000000	-100.0	-100.0	
6	350	-103.000	-103.0	0.000000	-103.0	-103.0	
7	400	-104.224	-104.0	0.416922	-105.0	-104.0	
8	450	-103.000	-103.0	0.000000	-103.0	-103.0	
9	500	-104.220	-104.0	0.414246	-105.0	-104.0	
10	550	-104.264	-104.0	0.440799	-105.0	-104.0	
11	600	-104.000	-104.0	0.000000	-104.0	-104.0	
12	650	-104.000	-104.0	0.000000	-104.0	-104.0	
13	700	-104.228	-104.0	0.419543	-105.0	-104.0	
14	750	-106.000	-106.0	0.000000	-106.0	-106.0	
15	800	-104.464	-104.0	0.844218	-106.0	-104.0	
16	850	-105.000	-105.0	0.000000	-105.0	-105.0	
17	900	-105.000	-105.0	0.000000	-105.0	-105.0	
18	950	-106.824	-107.0	1.302699	-109.0	-105.0	
19	1000	-104.540	-104.0	1.152562	-107.0	-104.0	
20	1050	-104.000	-104.0	0.000000	-104.0	-104.0	
21	1100	-100.000	-100.0	0.000000	-100.0	-100.0	
22	1150	-103.796	-104.0	0.402969	-104.0	-103.0	
23	1200	-106.128	-106.0	0.489506	-108.0	-106.0	
24	1250	-104.000	-104.0	0.000000	-104.0	-104.0	
25	1300	-107.852	-107.0	1.352810	-110.0	-107.0	
26	1350	-105.416	-106.0	0.909365	-106.0	-104.0	
27	1400	-100.000	-100.0	0.000000	-100.0	-100.0	
28	1450	-108.080	-108.0	0.271293	-109.0	-108.0	
29	1500	-105.840	-105.0	1.474585	-112.0	-105.0	
..	...	...	...	...	...	...	

(continues on next page)

(continued from previous page)

70	3550	36.040	89.0	87.544037	-116.0	89.0
71	3600	-109.624	-107.0	4.556602	-131.0	-107.0
72	3650	-87.048	-100.0	53.641902	-113.0	87.0
73	3700	90.472	91.0	0.881599	89.0	91.0
74	3750	90.536	91.0	0.844218	89.0	91.0
75	3800	90.488	91.0	0.872844	89.0	91.0
76	3850	90.416	91.0	0.909365	89.0	91.0
77	3900	90.464	91.0	0.885835	89.0	91.0
78	3950	90.392	91.0	0.919965	89.0	91.0
79	4000	90.432	91.0	0.901874	89.0	91.0
80	4050	90.520	91.0	0.854166	89.0	91.0
81	4100	90.424	91.0	0.905662	89.0	91.0
82	4150	90.400	91.0	0.916515	89.0	91.0
83	4200	90.504	91.0	0.863704	89.0	91.0
84	4250	90.544	91.0	0.839085	89.0	91.0
85	4300	90.544	91.0	0.839085	89.0	91.0
86	4350	90.424	91.0	0.905662	89.0	91.0
87	4400	90.512	91.0	0.858985	89.0	91.0
88	4450	90.552	91.0	0.833844	89.0	91.0
89	4500	90.536	91.0	0.844218	89.0	91.0
90	4550	90.544	91.0	0.839085	89.0	91.0
91	4600	90.480	91.0	0.877268	89.0	91.0
92	4650	90.504	91.0	0.863704	89.0	91.0
93	4700	90.544	91.0	0.839085	89.0	91.0
94	4750	90.472	91.0	0.881599	89.0	91.0
95	4800	90.728	91.0	0.685577	89.0	91.0
96	4850	90.520	91.0	0.854166	89.0	91.0
97	4900	90.584	91.0	0.811754	89.0	91.0
98	4950	90.496	91.0	0.868323	89.0	91.0
99	5000	90.408	91.0	0.912982	89.0	91.0

	steps_mean	steps_median	steps_std	steps_min	steps_max
0	5.000	5.0	0.000000	5	5
1	5.000	5.0	0.000000	5	5
2	5.000	5.0	0.000000	5	5
3	4.000	4.0	0.000000	4	4
4	5.424	6.0	0.905662	4	6
5	101.000	101.0	0.000000	101	101
6	5.000	5.0	0.000000	5	5
7	6.224	6.0	0.416922	6	7
8	5.000	5.0	0.000000	5	5
9	6.220	6.0	0.414246	6	7
10	6.264	6.0	0.440799	6	7
11	6.000	6.0	0.000000	6	6
12	6.000	6.0	0.000000	6	6
13	6.228	6.0	0.419543	6	7
14	8.000	8.0	0.000000	8	8
15	6.464	6.0	0.844218	6	8
16	7.000	7.0	0.000000	7	7
17	7.000	7.0	0.000000	7	7
18	8.824	9.0	1.302699	7	11
19	6.540	6.0	1.152562	6	9
20	6.000	6.0	0.000000	6	6
21	101.000	101.0	0.000000	101	101
22	5.796	6.0	0.402969	5	6
23	8.128	8.0	0.489506	8	10
24	6.000	6.0	0.000000	6	6

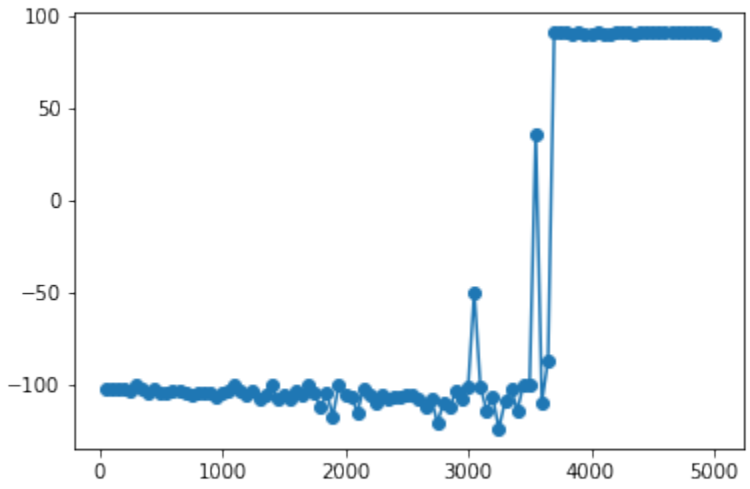
(continues on next page)

(continued from previous page)

```
25       9.852           9.0   1.352810          9         12
26       7.416           8.0   0.909365          6          8
27     101.000         101.0   0.000000        101        101
28      10.080          10.0   0.271293         10         11
29       7.840           7.0   1.474585          7         14
..         ...           ...        ...        ...        ...
70      12.360          13.0   2.076150          9         18
71      11.624           9.0   4.556602          9         33
72      64.860         101.0  43.202042         11        101
73      11.528          11.0   0.881599         11         13
74      11.464          11.0   0.844218         11         13
75      11.512          11.0   0.872844         11         13
76      11.584          11.0   0.909365         11         13
77      11.536          11.0   0.885835         11         13
78      11.608          11.0   0.919965         11         13
79      11.568          11.0   0.901874         11         13
80      11.480          11.0   0.854166         11         13
81      11.576          11.0   0.905662         11         13
82      11.600          11.0   0.916515         11         13
83      11.496          11.0   0.863704         11         13
84      11.456          11.0   0.839085         11         13
85      11.456          11.0   0.839085         11         13
86      11.576          11.0   0.905662         11         13
87      11.488          11.0   0.858985         11         13
88      11.448          11.0   0.833844         11         13
89      11.464          11.0   0.844218         11         13
90      11.456          11.0   0.839085         11         13
91      11.520          11.0   0.877268         11         13
92      11.496          11.0   0.863704         11         13
93      11.456          11.0   0.839085         11         13
94      11.528          11.0   0.881599         11         13
95      11.272          11.0   0.685577         11         13
96      11.480          11.0   0.854166         11         13
97      11.416          11.0   0.811754         11         13
98      11.504          11.0   0.868323         11         13
99      11.592          11.0   0.912982         11         13

[100 rows x 11 columns]
```

[31]:
```
plt.plot(rt_ql_intermediate_scoring_df.loc[:, 'iteration'], rt_ql_intermediate_
↪scoring_df.loc[:, 'reward_mean'], '-o')
```

[31]:
```
[<matplotlib.lines.Line2D at 0x1f33df6ca90>]
```



In this case we see somewhere between the 3500th and 4000th iteration the solver finds a solution and starts building a policy around it. This won't always be the case and learning may be more incremental

And if we wanted to access the actual episodes that went into one of these datapoints, they're available in a dictionary of EpisodeStatistics objects here (keyed by iteration number):

[33]:
```
i = 3750
print(f'EpisodeStatistics for the scoring at iter == {i}:\n')
rt_ql.scoring_episode_statistics[i].to_dataframe().head()
```

```
EpisodeStatistics for the scoring at iter == 3750:
```

[33]:

	episode_index	reward	steps	terminal	reward_mean	reward_median	reward_std	reward_min	reward_max	steps_mean	steps_median	steps_std	steps_min	steps_max	terminal_fraction
0	0	91.0	11	True	91.0	91.0	0.000000	91.0	91.0	11.0	11.0	0.000000	11	11	1.0
1	1	91.0	11	True	91.0	91.0	0.000000	91.0	91.0	11.0	11.0	0.000000	11	11	1.0
2	2	91.0	11	True	91.0	91.0	0.000000	91.0	91.0	11.0	11.0	0.000000	11	11	1.0
3	3	89.0	13	True	90.5	91.0	0.866025	89.0	91.0	11.5	11.0	0.866025	11	13	1.0
4	4	89.0	13	True	90.2	91.0	0.979796	89.0	91.0	11.8	11.0	0.979796	11	13	1.0

2.2 Example Case using RewardingFrozenLake

Below is an example of how to initialize the RewardingFrozenLake environment and solve/compare with multiple solvers.

2.2.1 Boilerplate

If you're playing with things under the hood as you run these, autoreload is always useful...

[1]:
```
%load_ext autoreload

%autoreload 2
```

If necessary, add directory containing lrl to path (workaround for if lrl is not installed as a package)

[2]:
```
import sys

# Path to directory containing lrl
sys.path.append('../')
```

[3]:
```
from lrl import environments, solvers
from lrl.utils import plotting

import matplotlib.pyplot as plt
```

Logging is used throughout lrl for basic info and debugging.

[4]:
```
import logging
logging.basicConfig(format='%(asctime)s - %(name)s - %(funcName)s - %(levelname)s -
↪%(message)s',
                    level=logging.INFO, datefmt='%H:%M:%S')
logger = logging.getLogger(__name__)
```

2.2.2 Initialize an Environment

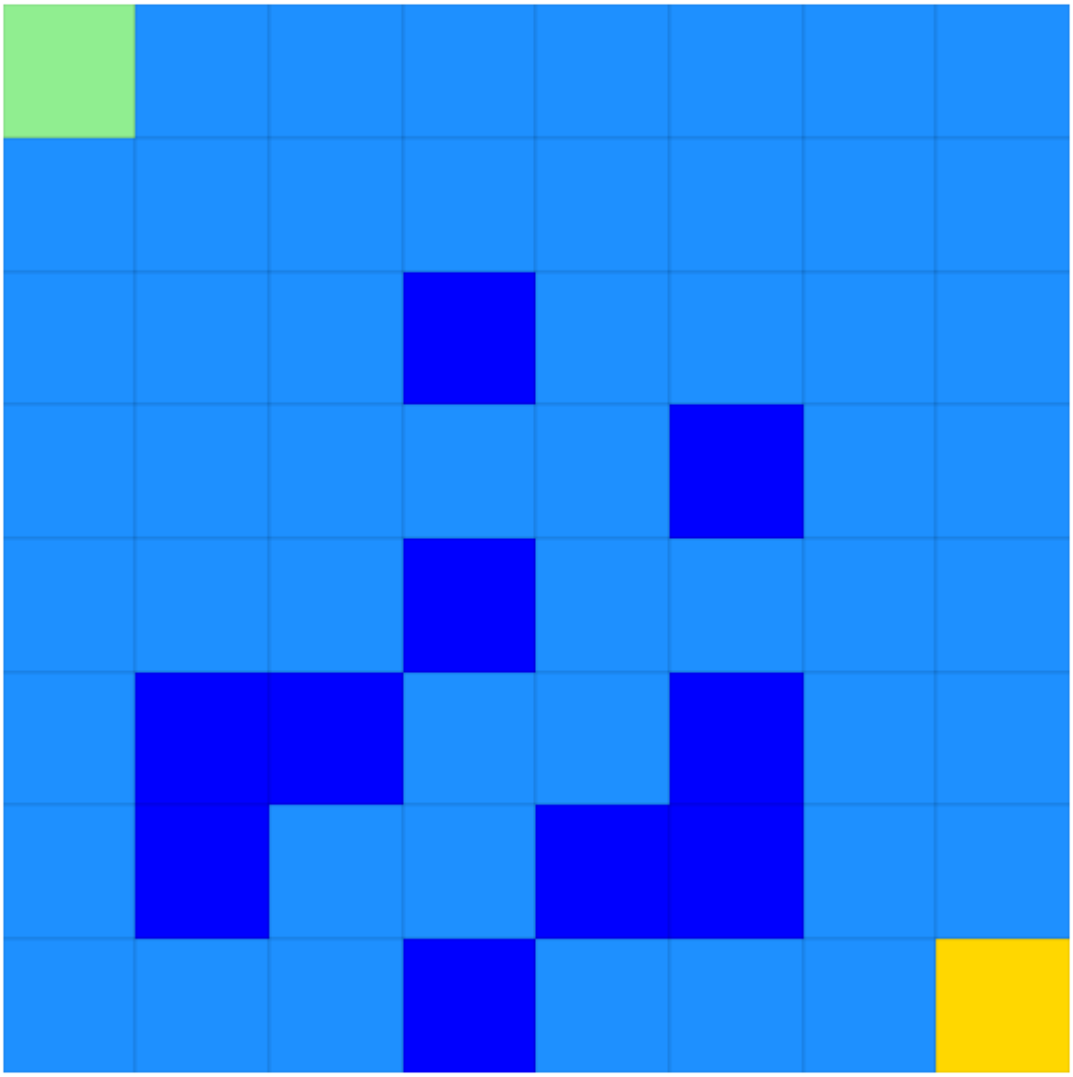
Initialize an 8x8 Frozen Lake (4x4 and other shapes also available - see code and docs)

[5]:
```
lake = environments.frozen_lake.RewardingFrozenLakeEnv(map_name='8x8', is_
↪slippery=True)
```

Take a look at the env using plot_env

[6]:
```
plotting.plot_env(env=lake)
```

[6]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x230f5eb6a90>
```



Ice is light blue - holes are dark blue. Make it from green to yellow without hitting any holes!

2.2.3 Solve with Value Iteration and Interrogate Solution

First, with Value Iteration

[7]:
```
lake_vi = solvers.ValueIteration(env=lake)
lake_vi.iterate_to_convergence()
```

```
16:32:22 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Solver iterating␣
↪to convergence (Max delta in value function < 0.001 or iters>500)
16:32:22 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Solver converged␣
↪to solution in 32 iterations
```

And we can then score our solution by running it multiple times through the environment

[8]:
```
scoring_data = lake_vi.score_policy(iters=500)
```

score_policy returns a EpisodeStatistics object that contains details from each episode taken during the scoring, plus aggregate scores. Easiest way to interact with it is grabbing data as a dataframe

[9]:
```
print(f'type(scoring_data) = {type(scoring_data)}')
scoring_data_df = scoring_data.to_dataframe(include_episodes=True)
scoring_data_df.head(3)
```

```
type(scoring_data) = <class 'lrl.data_stores.data_stores.EpisodeStatistics'>
```

[9]:
```
   episode_index  reward  steps  terminal  reward_mean  reward_median  \
0              0    0.87     15      True     0.870000           0.87
1              1    0.81     21      True     0.840000           0.84
2              2    0.85     17      True     0.843333           0.85

   reward_std  reward_min  reward_max  steps_mean  steps_median  steps_std  \
0    0.000000        0.87        0.87   15.000000          15.0   0.000000
1    0.030000        0.81        0.87   18.000000          18.0   3.000000
2    0.024944        0.81        0.87   17.666667          17.0   2.494438

   steps_min  steps_max  terminal_fraction  \
0         15         15                1.0
1         15         21                1.0
2         15         21                1.0

                                        episodes
0  [0, 1, 2, 3, 4, 5, 6, 7, 15, 23, 31, 39, 47, 5...
1  [0, 0, 1, 2, 10, 11, 3, 4, 12, 13, 14, 22, 23,...
2  [0, 1, 2, 3, 4, 12, 13, 14, 22, 23, 31, 30, 38...
```

[10]:
```
scoring_data_df.tail(3)
```

[10]:
```
     episode_index  reward  steps  terminal  reward_mean  reward_median  \
497            497    0.82     20      True     0.805843           0.81
498            498    0.84     18      True     0.805912           0.81
499            499    0.81     21      True     0.805920           0.81

     reward_std  reward_min  reward_max  steps_mean  steps_median  steps_std  \
497    0.045295        0.58        0.87   21.415663          21.0   4.529457
498    0.045275        0.58        0.87   21.408818          21.0   4.527494
499    0.045230        0.58        0.87   21.408000          21.0   4.523001

     steps_min  steps_max  terminal_fraction  \
497         15         44                1.0
498         15         44                1.0
499         15         44                1.0

                                          episodes
497  [0, 1, 2, 3, 3, 4, 12, 13, 14, 22, 14, 22, 30,...
498  [0, 1, 2, 3, 4, 5, 6, 14, 22, 30, 31, 39, 39, ...
499  [0, 1, 2, 3, 4, 5, 6, 14, 22, 23, 31, 39, 39, ...
```

Reward, Steps, and Terminal columns give data on that specific episode, whereas reward_mean, _median, etc. columns give aggregate scores up until that walk. For example:

[11]:
```
print(f'The reward obtained in the 499th episode was {scoring_data_df.loc[499, "reward
↪"]}')
```

(continues on next page)

(continued from previous page)

```
print(f'The mean reward obtained in the 0-499th episode (inclusive) was {scoring_data_
↪df.loc[499, "reward_mean"]}')
```

```
The reward obtained in the 499th episode was 0.8099999999999999
The mean reward obtained in the 0-499th episode (inclusive) was 0.80592
```

And we can access the actual episode path for each episode

[12]:
```
print(f'Episode 0 (directly)          : {scoring_data.episodes[0]}')
print(f'Episode 0 (from the dataframe): {scoring_data_df.loc[0, "episodes"]}')
```

```
Episode 0 (directly)          : [0, 1, 2, 3, 4, 5, 6, 7, 15, 23, 31, 39, 47, 55, 63]
Episode 0 (from the dataframe): [0, 1, 2, 3, 4, 5, 6, 7, 15, 23, 31, 39, 47, 55, 63]
```

But note that the Frozen Lake indexes its states by an integer, not the (x, y) or (row, col) coordinates. But we can get coordinates easily enough...

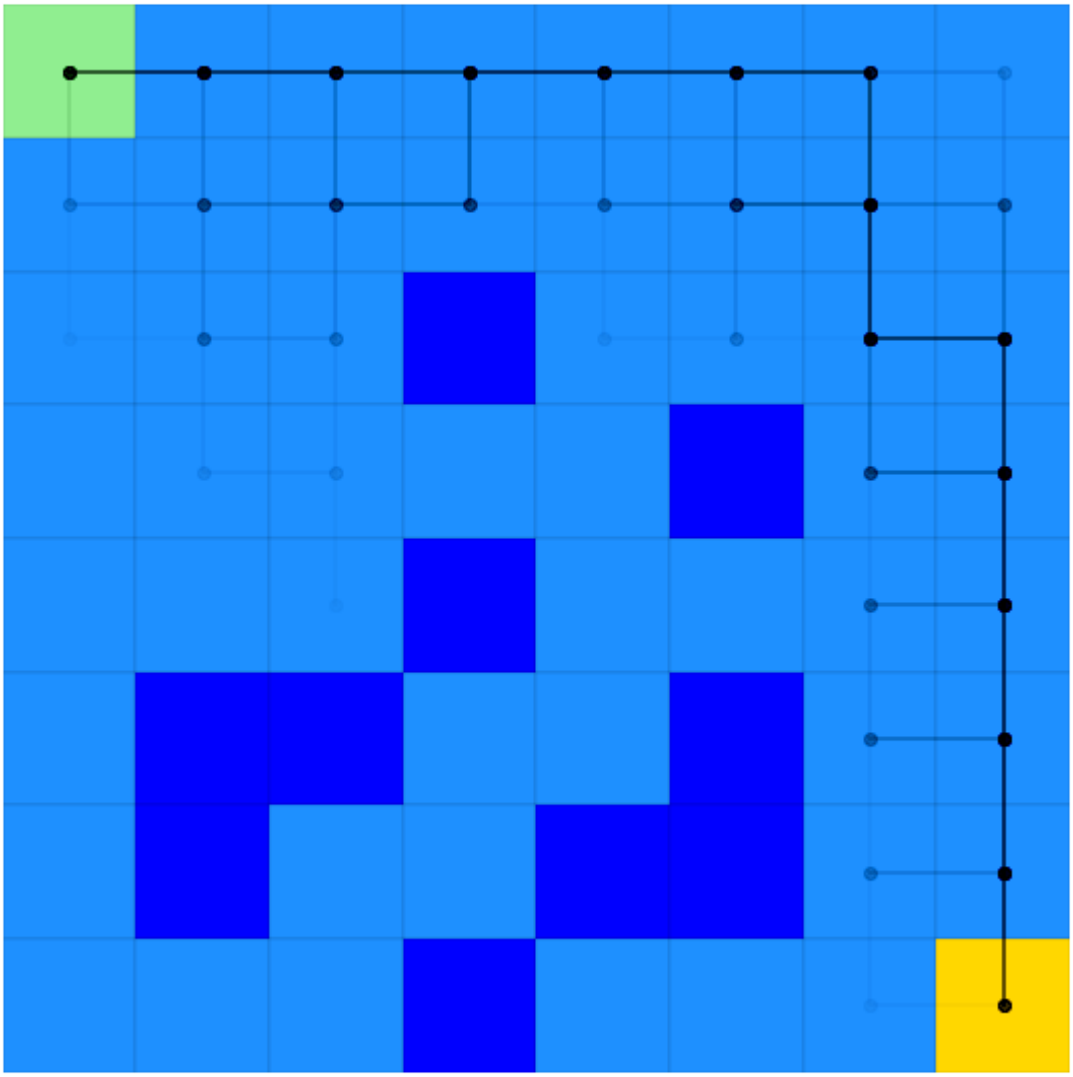
[13]:
```
i = 0
print(f'Location index={i} is at (x, y) location {lake.index_to_state[i]} (where x=0␣
↪is left and y=0 is bot)')
loc = (0, 0)
print(f'And going the other way, location {loc} is index {lake.state_to_index[loc]}')
```

```
Location index=0 is at (x, y) location (0, 7) (where x=0 is left and y=0 is bot)
And going the other way, location (0, 0) is index 56
```

2.2.4 Plotting Results

Plotting 100 randomly chosen episodes on the map, returned as a matplotlib axes, we get:

[14]:
```
ax_episodes = plotting.plot_episodes(episodes=scoring_data.episodes, env=lake, max_
↪episodes=100, )
```



Where we see our strategy tries to stay around the outside edge for the most part, but sometiems slips closer to the middle (although we never actually hit a hole!)

score_policy also lets us use hypothetical scenario, such as what if we started in a different starting location. Let's try that by starting in the middle of some of the holes, plotting this new scenario in red.

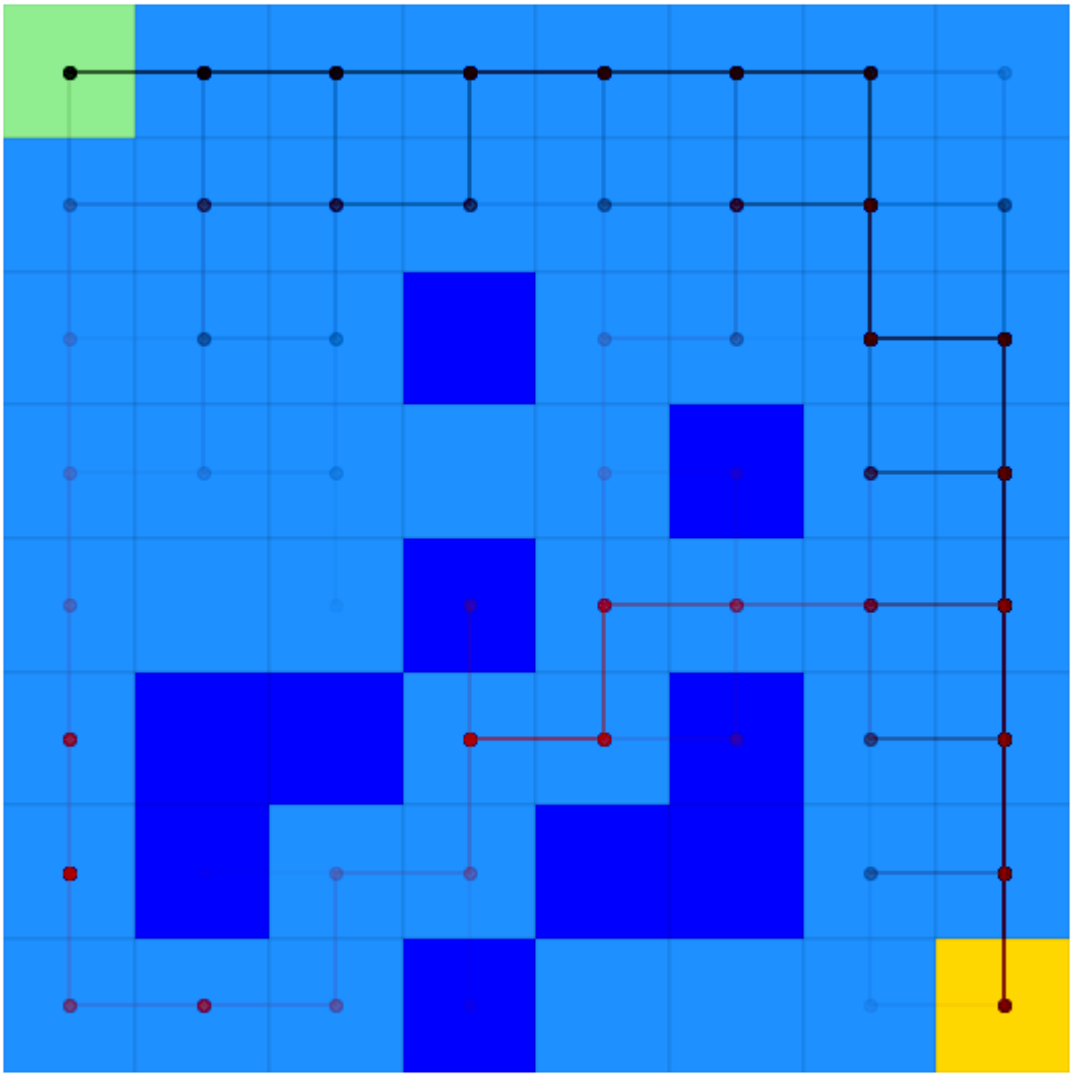
[15]:
```
# Initial state is in (x, y) format
initial_state_tuple = (3, 2)
initial_state_index = lake.state_to_index[initial_state_tuple]
scoring_data_alternate = lake_vi.score_policy(iters=500, initial_state=initial_state_
↪index)
ax_episodes_with_alternate = plotting.plot_episodes(env=lake, episodes=scoring_data_
↪alternate.episodes, add_env_to_plot=False,
                                                    color='r', ax=ax_episodes,
#                                                     savefig='my_figure_file',  # If␣
↪you wanted the figure to save directly
                                                                               # to␣
↪file, use savefig (used throughout
```

(continues on next page)

(continued from previous page)

```
                                                                    # lrl
↪'s plotting scripts)
                                                                    )

# Must get_figure because we're reusing the figure from above and jupyter wont␣
↪automatically reshow it
ax_episodes_with_alternate.get_figure()
```

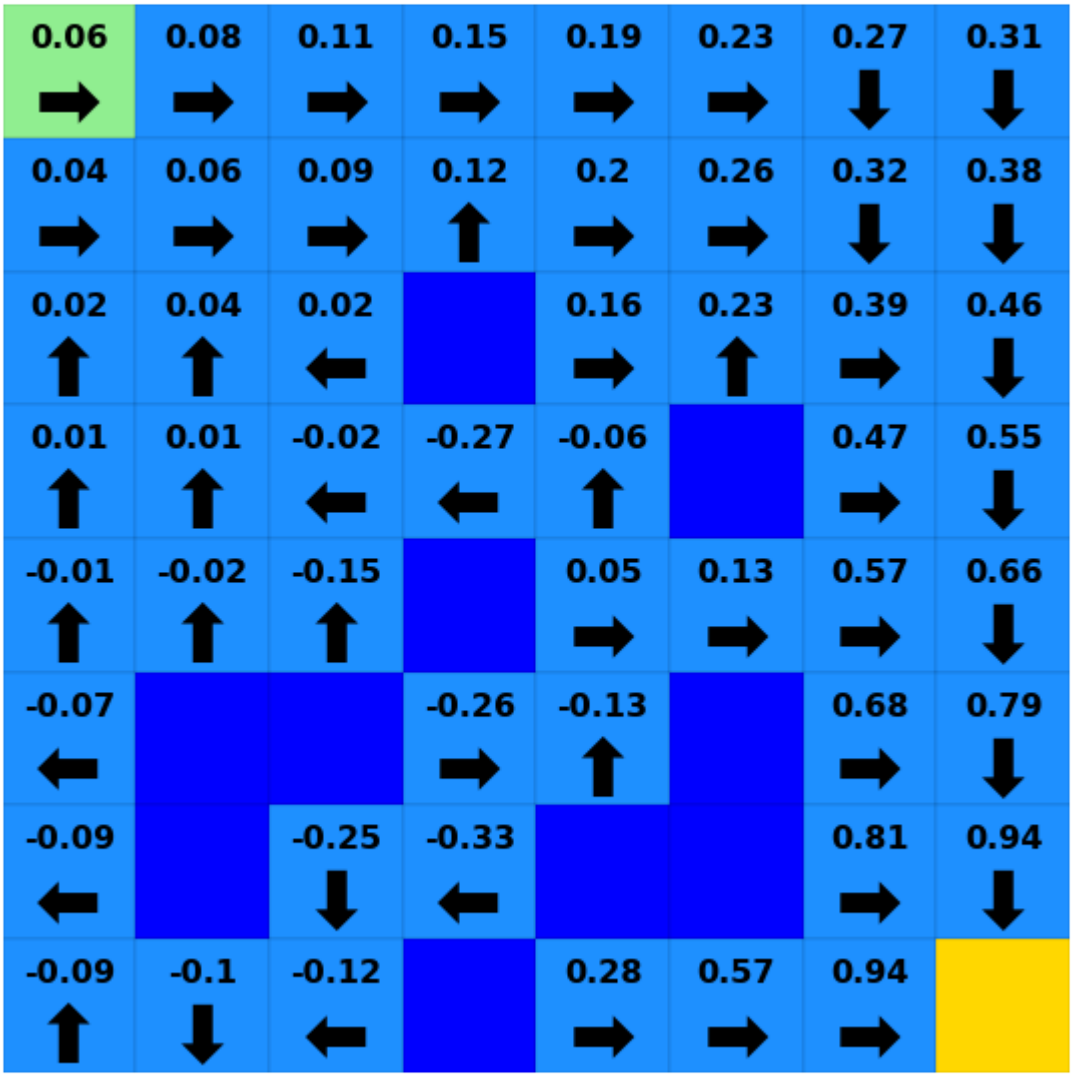
[15]:



Where we can see the optimal policy sometimes manages to win, and sometimes just falls in a hole…

We can also look at the Value function and best policy for all states

```
[16]: ax_results = plotting.plot_solver_results(env=lake, solver=lake_vi)
```



One thing we might want to do with this solution is see how the policy evolved over time. Policies and value (or Q) functions are saved in a dict-like data store that maintains both current value and the history of that value over time. For example, lets look at the value for index 0 (bottom left)

We can grab the most recent value at location 0 by:

[17]:
```
iteration = 31  # Could also be -1 to get the last item, like normal list syntax
lake_vi.value.get_value_at_timepoint(key=0, timepoint=iteration)
```

[17]:
```
0.05538280537023101
```

And we can also get the history of all values for location 0

[18]:
```
lake_vi.value.get_value_history(0)
```

[18]:
```
[(0, -0.009999999999999998),
 (1, -0.018999999999999996),
 (2, -0.0271),
 (3, -0.03439),
```

(continues on next page)

(continued from previous page)

```
 (4, -0.040951),
 (5, -0.046855900000000006),
 (6, -0.05217031000000001),
 (7, -0.056953279000000016),
 (8, -0.06125795110000003),
 (9, -0.06513215599000002),
 (10, -0.06861894039100003),
 (11, -0.07175704635190003),
 (12, -0.07458134171671002),
 (13, -0.06620112263383392),
 (14, -0.05606391032872242),
 (15, -0.03814365080870915),
 (16, -0.02193630467499485),
 (17, -0.004533717206715425),
 (18, 0.009321649875289119),
 (19, 0.021602026276737636),
 (20, 0.030685599534352126),
 (21, 0.037967128156801665),
 (22, 0.04311442211812146),
 (23, 0.04700668910612634),
 (24, 0.04968251379686211),
 (25, 0.05163476331362879),
 (26, 0.0529522810474764),
 (27, 0.05389380710935053),
 (28, 0.054507344360609306),
 (29, 0.05495173816104297),
 (30, 0.05519127451645814),
 (31, 0.05538280537023101)]
```

We see that the value changes each iteration, so we have one entry per iteration. But, if we look at policy, we see a more efficient storage:

[19]:
```
iteration = 31  # Could also be -1 to get the last item, like normal list syntax
print(f'Most recent policy at key=0: {lake_vi.policy.get_value_at_timepoint(key=0,␣
↪timepoint=iteration)}')
print('History of all changes:')
lake_vi.policy.get_value_history(0)
```

```
Most recent policy at key=0: 2
History of all changes:
```

[19]:
```
[(0, 0), (13, 2)]
```

Where we see policy was initialized at iteration 0 (to value of 0), and changed at iteration 13 to a value of 2. What is action 2?

[20]:
```
action = 2
print(f'Using lake environment, action {action} is {lake.action_as_char[2]}')
```

```
Using lake environment, action 2 is
```

We can also access the entire policy at a given timepoint by grabbing it from the data store as a dict:

[21]:
```
p_12 = lake_vi.policy.to_dict(timepoint=12)
p_13 = lake_vi.policy.to_dict(timepoint=13)
i = 0
print(f'policy at index {i} was {lake.action_as_char[p_12[i]]} at iteration 12 and
↪{lake.action_as_char[p_13[i]]} at iteration 13')
```

(continues on next page)

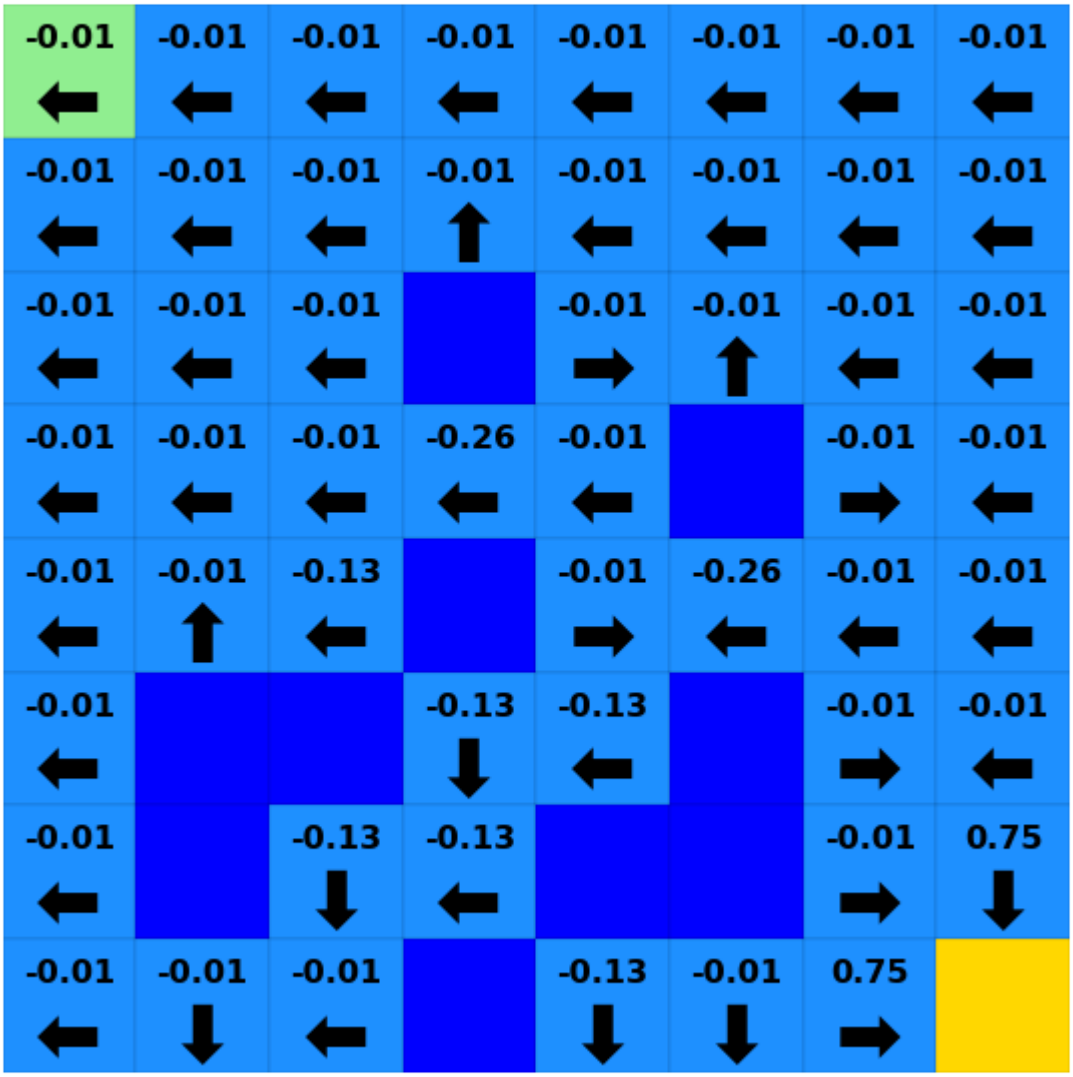
(continued from previous page)

```
policy at index 0 was  at iteration 12 and  at iteration 13
```

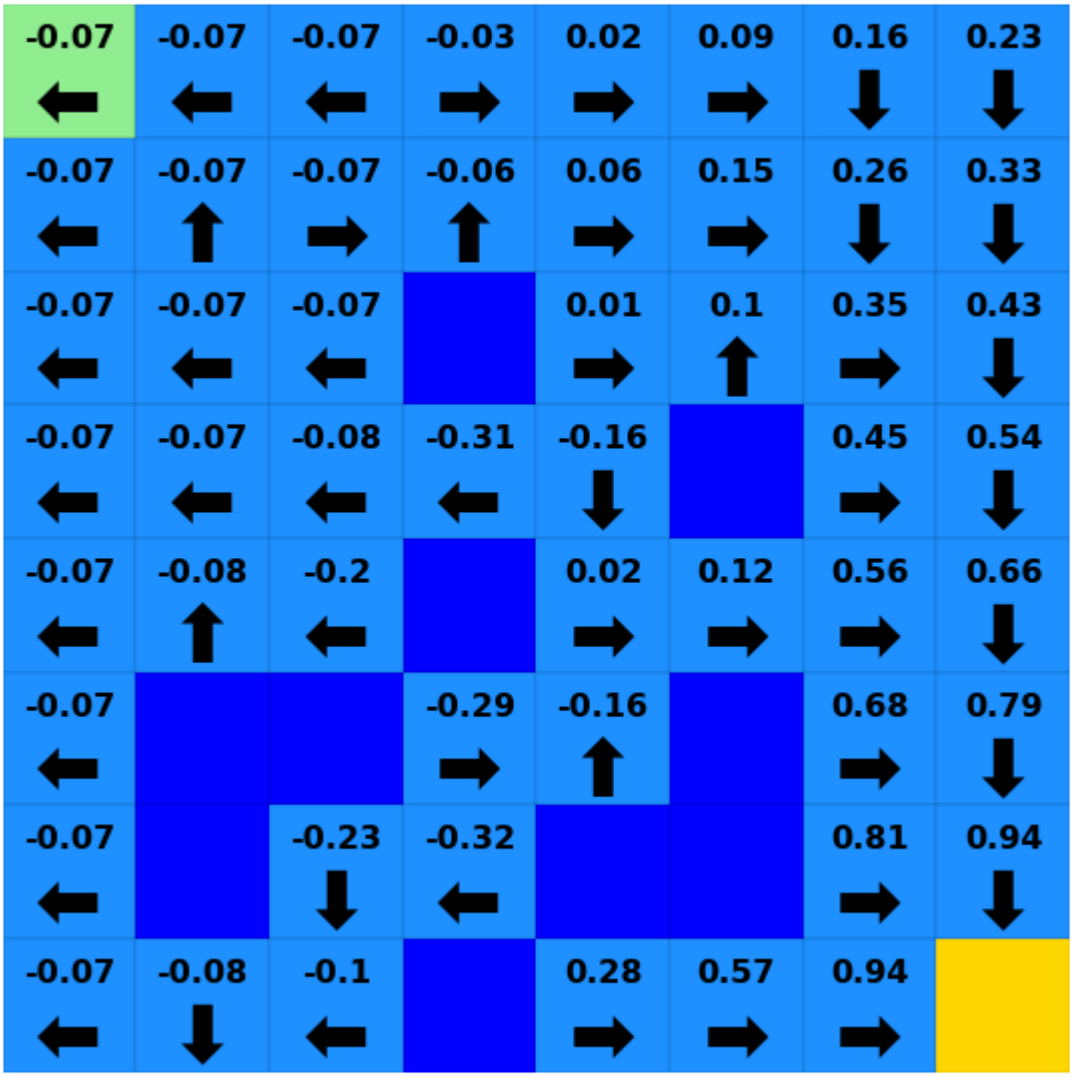
And even plot everything over time...

[22]:
```
for i in [0, 10, 20, 30]:
    print(f"Iteration {i}")
    ax = plotting.plot_solver_results(env=lake,
                                      policy=lake_vi.policy.to_dict(timepoint=i),
                                      value=lake_vi.value.to_dict(timepoint=i),
                                      )
    plt.show()
```

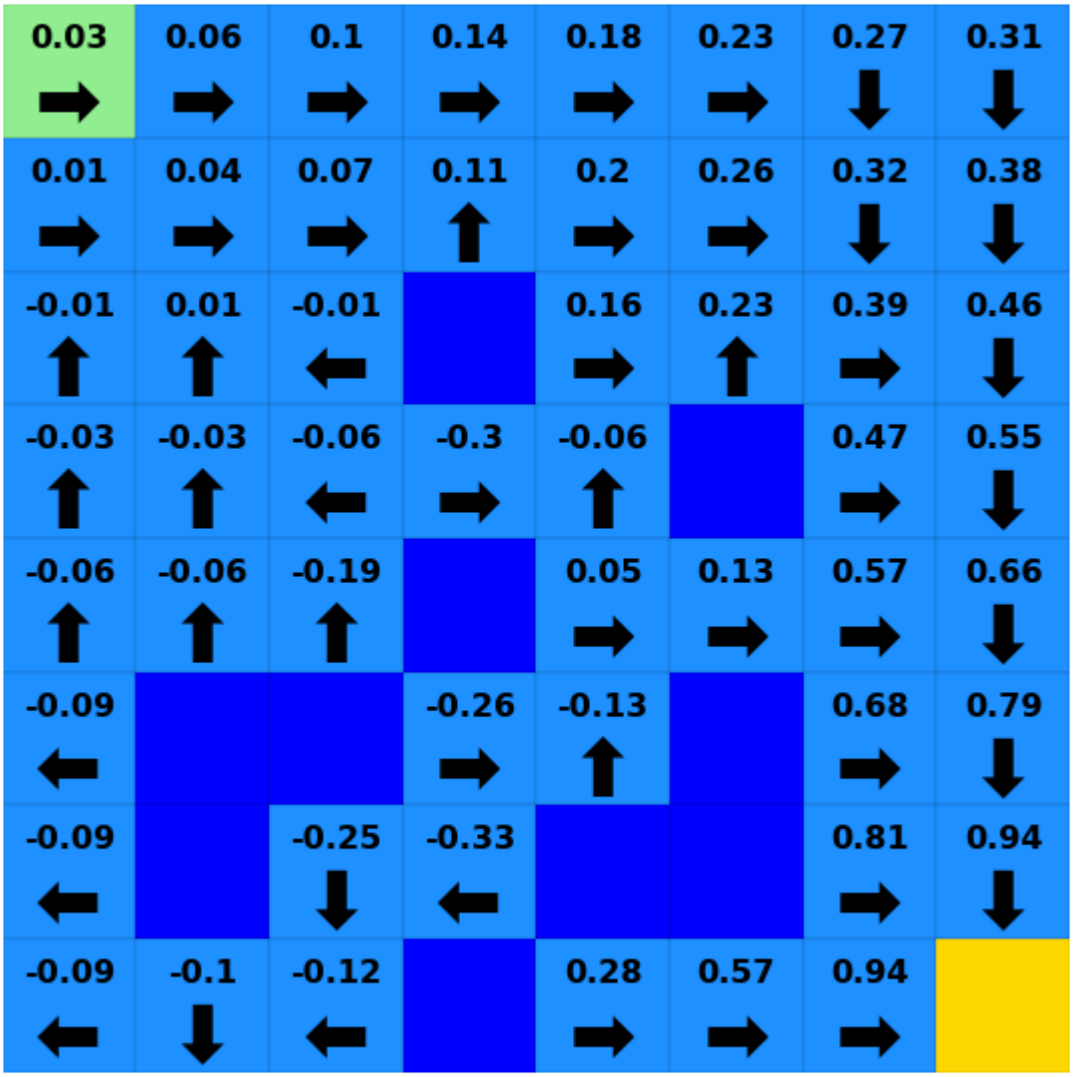
```
Iteration 0
```



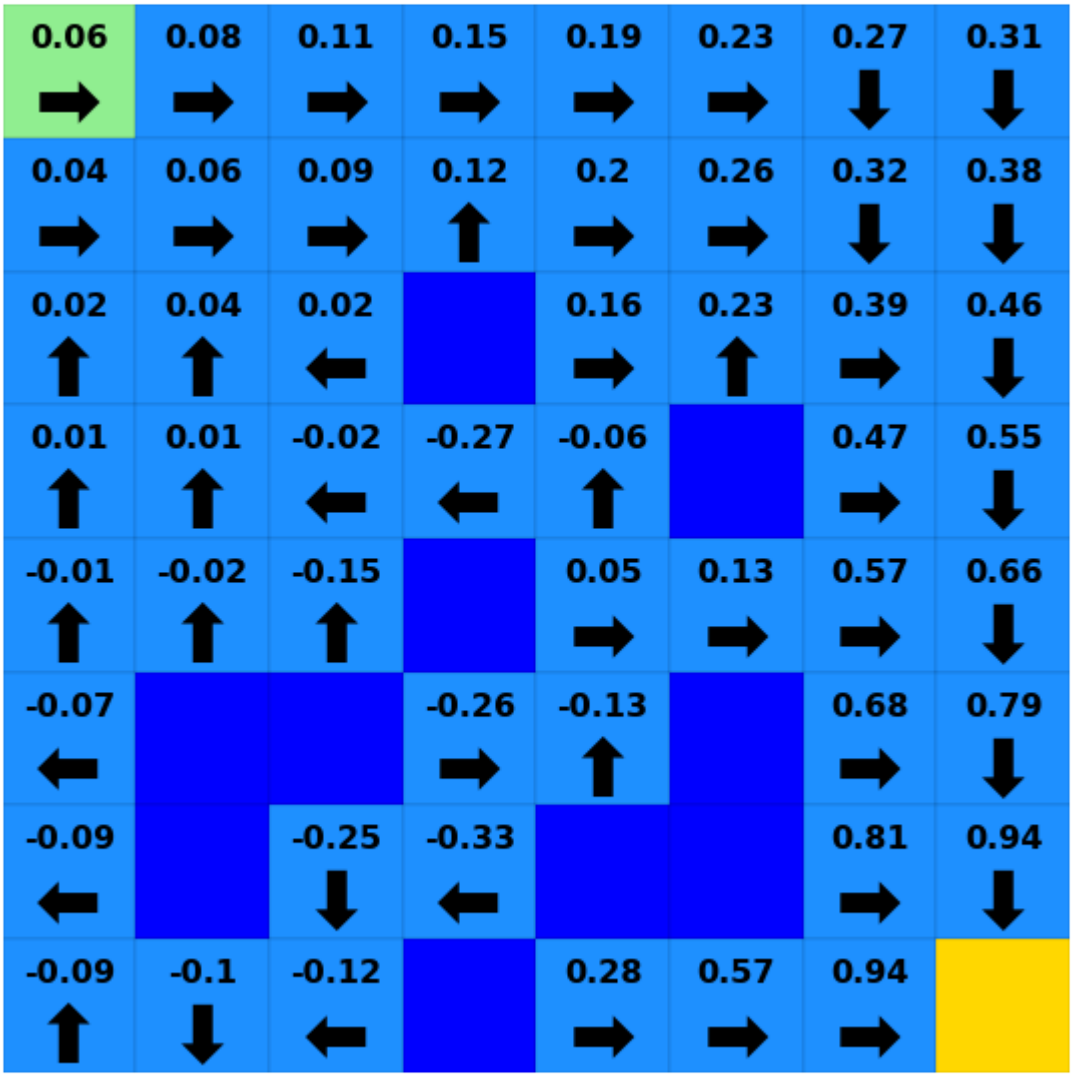

```
Iteration 10
```



Iteration 20



Iteration 30



2.2.5 Solving with Policy Iteration and Comparing to Value Iteration

We can also use other solvers

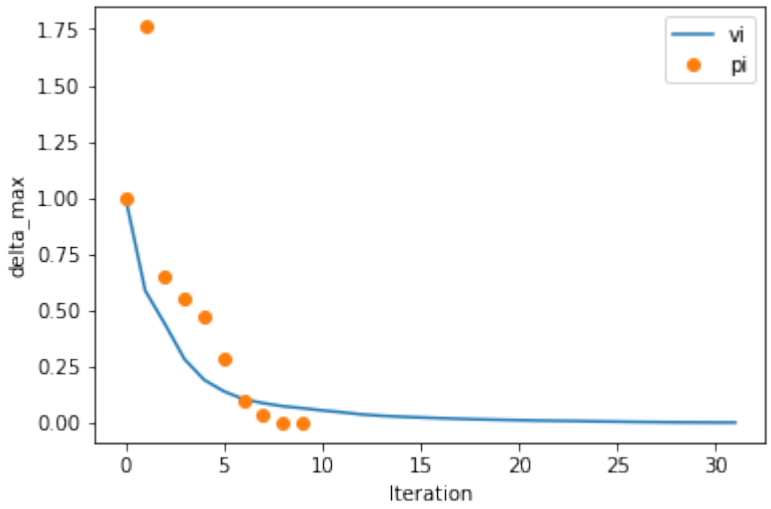
[23]:
```
lake_pi = solvers.PolicyIteration(env=lake)
lake_pi.iterate_to_convergence()
```

```
16:32:36 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Solver iterating␣
↪to convergence (1 iteration without change in policy or iters>500)
16:32:36 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Solver converged␣
↪to solution in 10 iterations
```

We can look at how PI and VI converged relative to each other, comparing the maximum change in value function for each iteration

[24]:
```
# (these are simple convenience functions for plotting, basically just recipes.  See
↪the plotting API)
# We can pass the solver..
ax = plotting.plot_solver_convergence(lake_vi, label='vi')

# Or going a little deeper into the API, with style being passed to matplotlib's plot
↪function...
ax = plotting.plot_solver_convergence_from_df(lake_pi.iteration_data.to_dataframe(),
↪y='delta_max', x='iteration', ax=ax, label='pi', ls='', marker='o')

ax.legend()
```

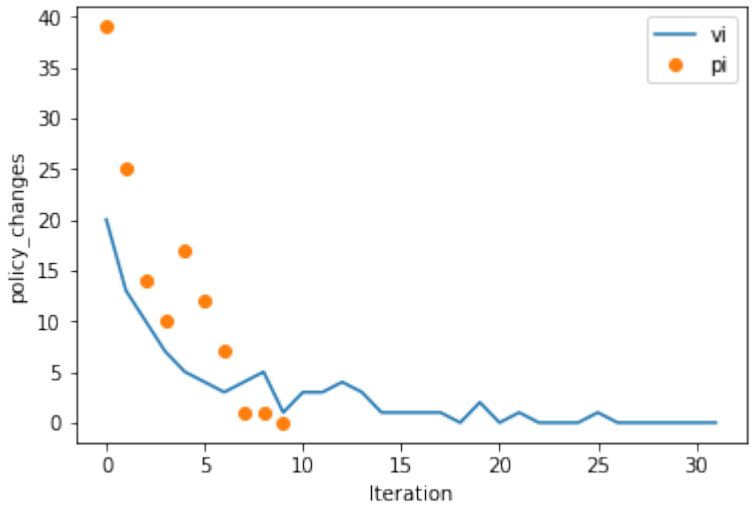
[24]:
```
<matplotlib.legend.Legend at 0x230f8efd048>
```



And looking at policy changes per iteration

[25]:
```
# (these are simple convenience functions for plotting, basically just recipes.  See
↪the plotting API)
# We can pass the solver..
ax = plotting.plot_solver_convergence(lake_vi, y='policy_changes', label='vi')

# Or going a little deeper into the API...
ax = plotting.plot_solver_convergence_from_df(lake_pi.iteration_data.to_dataframe(),
↪y='policy_changes', x='iteration', ax=ax, label='pi', ls='', marker='o')

ax.legend()
```

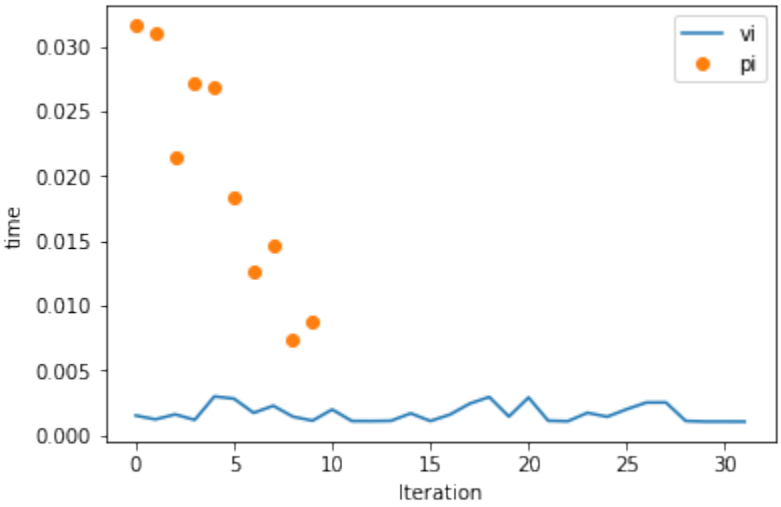
[25]:
```
<matplotlib.legend.Legend at 0x230f909fb38>
```



So we can see PI accomplishes a little more per iteration. But, is it faster? Let's look at time per iteration

[26]:
```
# (these are simple convenience functions for plotting, basically just recipes.  See␣
↪the plotting API)
# We can pass the solver..
ax = plotting.plot_solver_convergence(lake_vi, y='time', label='vi')

# Or going a little deeper into the API...
ax = plotting.plot_solver_convergence_from_df(lake_pi.iteration_data.to_dataframe(),␣
↪y='time', x='iteration', ax=ax, label='pi', ls='', marker='o')

ax.legend()

print(f'Total solution time for Value Iteration (excludes any scoring time):  {lake_
↪vi.iteration_data.to_dataframe().loc[:, "time"].sum():.2f}s')
print(f'Total solution time for Policy Iteration (excludes any scoring time): {lake_
↪pi.iteration_data.to_dataframe().loc[:, "time"].sum():.2f}s')
```

```
Total solution time for Value Iteration (excludes any scoring time):  0.05s
Total solution time for Policy Iteration (excludes any scoring time): 0.20s
```



2.2.6 Solve with Q-Learning

We can also use QLearning, although it needs a few parameters

[27]:
```
# Let's be explicit with our QLearning settings for alpha and epsilon
alpha = 0.1  # Constant alpha during learning

# Decay function for epsilon (see QLearning() and decay_functions() in documentation␣
↪for syntax)
# Decay epsilon linearly from 0.2 at timestep (iteration) 0 to 0.05 at timestep 1500,
# keeping constant at 0.05 for ts>1500
epsilon = {
    'type': 'linear',
    'initial_value': 0.2,
    'initial_timestep': 0,
    'final_value': 0.05,
    'final_timestep': 1500
}

# Above PI/VI used the default gamma, but we will specify one here
gamma = 0.9

# Convergence is kinda tough to interpret automatically for Q-Learning.  One good way␣
↪to monitor convergence is to
# evaluate how good the greedy policy at a given point in the solution is and decide␣
↪if it is still improving.
# We can enable this with score_while_training (available for Value and Policy␣
↪Iteration as well)
# NOTE: During scoring runs, the solver is acting greedily and NOT learning from the␣
↪environment.  These are separate
#       runs solely used to estimate solution progress
# NOTE: Scoring every 50 iterations is probably a bit much, but used to show a nice␣
↪plot below.  The default 500/500
#       is probably a better general guidance
score_while_training = {
    'n_trains_per_eval': 50,  # Number of training episodes we run per attempt to␣
↪score the greedy policy
                              # (eg: Here we do a scoring run after every 500␣
↪training episodes, where training episodes
                              # are the usual epsilon-greedy exploration episodes)
    'n_evals': 250,  # Number of times we run through the env with the greedy policy␣
↪whenever we score
}
# score_while_training = True  # This calls the default settings, which are also 500/
↪500 like above

lake_ql = solvers.QLearning(env=lake, alpha=alpha, epsilon=epsilon, gamma=gamma,
                            max_iters=5000, score_while_training=score_while_training)
```

(note how long Q-Learning takes for this environment versus the planning algorithms)

[28]:
```
lake_ql.iterate_to_convergence()
```

```
16:32:38 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Solver iterating␣
↪to convergence (20 episodes with max delta in Q function < 0.1 or iters>5000)
16:32:38 - lrl.solvers.learners - iterate - INFO - Performing iteration (episode) 0␣
↪of Q-Learning
16:32:39 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 50
```

(continues on next page)

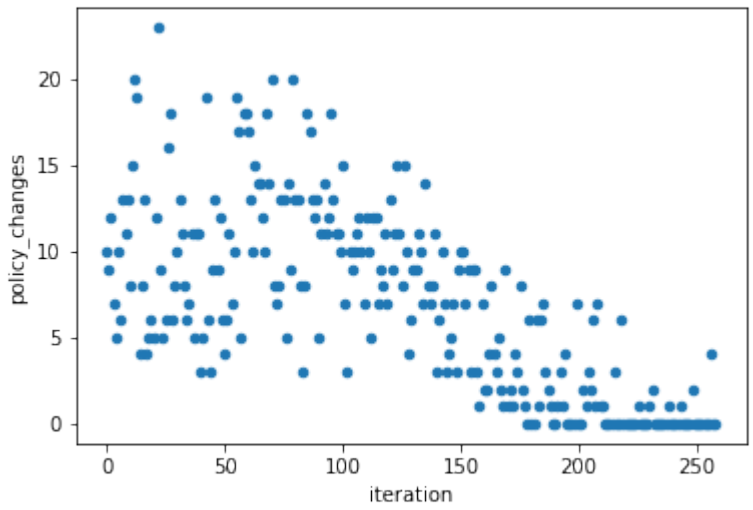
(continued from previous page)

```
16:32:39 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -1.2661200000000001, r_max = -1.0000000000000007
16:32:39 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 100
16:32:40 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = -1.2866800000000003, r_max = -1.0000000000000007
16:32:40 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 150
16:32:40 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 0.013879999999999683, r_max = 0.83
16:32:41 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 200
16:32:41 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 0.7806399999999999, r_max = 0.87
16:32:41 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy being scored 250 times at iteration 250
16:32:41 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Current greedy␣
↪policy achieved: r_mean = 0.67704, r_max = 0.87
16:32:41 - lrl.solvers.base_solver - iterate_to_convergence - INFO - Solver converged␣
↪to solution in 259 iterations
```

Like above, we can plot the number of policy changes per iteration. But this plot looks very different from above and shows one view of why Q-Learning takes many more iterations (each iteration accomplishes a lot less learning than a planning algorithm)

[29]:
```
lake_ql_iter_df = lake_ql.iteration_data.to_dataframe()
lake_ql_iter_df.plot(x='iteration', y='policy_changes', kind='scatter', )
```

[29]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x230f8c96c88>
```



We can access the intermediate scoring through the scoring_summary (GeneralIterationData) and scoring_episode_statistics (EpisodeStatistics) objects

[30]:
```
lake_ql_intermediate_scoring_df = lake_ql.scoring_summary.to_dataframe()
lake_ql_intermediate_scoring_df
```

[30]:
```
   iteration  reward_mean  reward_median  reward_std  reward_min  reward_max  \
0         50     -1.26612          -1.25    0.180753       -1.97       -1.00
```

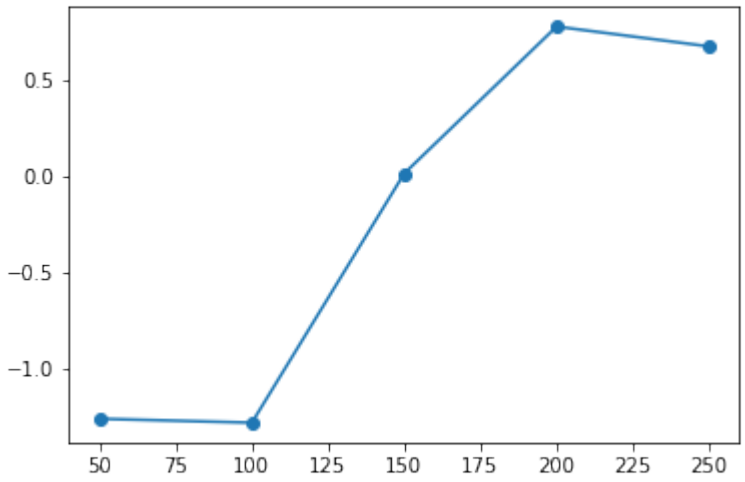
(continues on next page)

(continued from previous page)

```
1         100     -1.28668          -1.00     0.328476       -1.99       -1.00
2         150      0.01388           0.42     0.821986       -1.96        0.83
3         200      0.78064           0.80     0.145922       -1.19        0.87
4         250      0.67704           0.81     0.497950       -1.77        0.87

   steps_mean  steps_median  steps_std  steps_min  steps_max
0      38.512          31.0  26.113021          6        101
1      80.564         101.0  25.849988         15        101
2      61.600          55.0  27.831349         12        101
3      23.136          22.0   7.549139         15         77
4      23.872          21.0  13.893006         14        101
```

[31]:
```
plt.plot(lake_ql_intermediate_scoring_df.loc[:, 'iteration'], lake_ql_intermediate_
↪scoring_df.loc[:, 'reward_mean'], '-o')
```

[31]:
```
[<matplotlib.lines.Line2D at 0x230f7e76da0>]
```



In this case we see past ~200 iterations we're consistently getting good scores, and the solver converges shortly thereafter (no improvement over many iterations)

And if we wanted to access the actual episodes that went into one of these datapoints, they're available in a dictionary of EpisodeStatistics objects here (keyed by iteration number):

[32]:
```
i = 200
print(f'EpisodeStatistics for the scoring at iter == {i}:\n')
lake_ql.scoring_episode_statistics[i].to_dataframe().head()
```

```
EpisodeStatistics for the scoring at iter == 200:
```

[32]:
```
   episode_index  reward  steps  terminal  reward_mean  reward_median  \
0              0    0.69     33      True     0.690000           0.69
1              1    0.83     19      True     0.760000           0.76
2              2    0.87     15      True     0.796667           0.83
3              3    0.83     19      True     0.805000           0.83
4              4    0.71     31      True     0.786000           0.83

   reward_std  reward_min  reward_max  steps_mean  steps_median  steps_std  \
0    0.000000        0.69        0.69   33.000000          33.0   0.000000
1    0.070000        0.69        0.83   26.000000          26.0   7.000000
```

(continues on next page)

(continued from previous page)

```
2    0.077172        0.69        0.87   22.333333          19.0   7.717225
3    0.068374        0.69        0.87   21.500000          19.0   6.837397
4    0.072000        0.69        0.87   23.400000          19.0   7.200000

   steps_min  steps_max  terminal_fraction
0         33         33                1.0
1         19         33                1.0
2         15         33                1.0
3         15         33                1.0
4         15         33                1.0
```

CHAPTER

THREE

API

3.1 Solvers

class lrl.solvers.**PolicyIteration**(*env*, *value_function_initial_value=0.0*, *max_policy_eval_iters_per_improvement=10*, *policy_evaluation_type='on-policy-iterative'*, ***kwargs*)

Bases: lrl.solvers.base_solver.BaseSolver

Solver for policy iteration

Implemented as per Sutton and Barto's Reinforcement Learning (http://www.incompleteideas.net/book/RLbook2018.pdf, page 80).

Notes

See also BaseSolver for additional attributes, members, and arguments (missing here due to Sphinx bug with inheritance in docs)

Examples

See examples directory

Parameters

- **value_function_initial_value** (*float*) – Value to initialize all elements of the value function to
- **max_policy_eval_iters_per_improvement** –
- **policy_evaluation_type** (*str*) – Type of solution method for calculating policy (see policy_evaluation() for more details. Typical usage should not need to change this as it will make calculations slower and more memory intensive)
- **BaseSolver class for additional** (*See*) –

Returns None

value = None

Space-efficient dict-like storage of the current and all former value functions

Type *DictWithHistory*

iterate()

Perform a single iteration of policy iteration, updating self.value and storing metadata about the iteration.

Side Effects:

- self.value: Updated to the newest estimate of the value function
- self.policy: Updated to the greedy policy according to the value function estimate
- self.iteration: Increment iteration counter by 1
- self.iteration_data: Add new record to iteration data store

Returns None

converged()
Returns True if solver is converged.

Judge convergence by checking whether the most recent policy iteration resulted in any changes in policy

Returns Convergence status (True=converged)

Return type bool

_policy_evaluation(*max_iters=None*)
Compute an estimate of the value function for the current policy to within self.tolerance

Side Effects: self.value: Updated to the newest estimate of the value function

Returns None

_policy_improvement(*return_differences=True*)
Update the policy to be greedy relative to the most recent value function

Side Effects: self.policy: Updated to be greedy relative to self.value

Parameters **return_differences** – If True, return number of differences between old and new policies

Returns (if return_differences==True) Number of differences between the old and new policies

Return type int

init_policy(*init_type=None*)
Initialize self.policy, which is a dictionary-like DictWithHistory object for storing current and past policies

Parameters **init_type** (*None, str*) – Method used for initializing policy. Can be any of:

- None: Uses value in self.policy_init_type
- zeros: Initialize policy to all 0's (first action)
- **random: Initialize policy to a random action (action indices are random integer from** [0, len(self.env.P[this_state])], where P is the transition matrix and P[state] is a list of all actions available in the state)

Side Effects: If init_type is specified as argument, it is also stored to self.policy_init_type (overwriting previous value)

Returns None

iterate_to_convergence(*raise_if_not_converged=None*, *score_while_training=None*)
Perform self.iterate repeatedly until convergence, optionally scoring the current policy periodically

Side Effects: Many, but depends on the subclass of the solver's .iterate()

Parameters

- **raise_if_not_converged** (*bool*) – If true, will raise an exception if convergence is not reached before hitting maximum number of iterations. If None, uses self.raise_if_not_converged
- **score_while_training** (*bool, dict, None*) – If None, use self.score_while_training. Else, accepts inputs of same format as accepted for score_while_training solver inputs

Returns None

run_policy(*max_steps=None*, *initial_state=None*)

Perform a walk (episode) through the environment using the current policy

Side Effects:

- self.env will be reset and optionally then forced into initial_state

Parameters

- **max_steps** – Maximum number of steps to be taken in the walk (step 0 is taken to be entering initial state) If None, defaults to self.max_steps_per_episode
- **initial_state** – State for the environment to be placed in to start the walk (used to force a deterministic start from anywhere in the environment rather than the typical start position)

Returns

tuple containing:

- **states** (*list*): boolean indicating if the episode was terminal according to the environment
- **rewards** (*list*): list of rewards obtained during the episode (rewards[0] == 0 as step 0 is simply starting the game)
- **is_terminal** (*bool*): Boolean denoting whether the environment returned that the episode terminated naturally

Return type (tuple)

score_policy(*iters=500*, *max_steps=None*, *initial_state=None*)

Score the current policy by performing iters greedy episodes in the environment and returning statistics

Side Effects: self.env will be reset more side effects

more side effects

Parameters

- **iters** – Number of episodes in the environment
- **max_steps** – Maximum number of steps allowed per episode. If None, defaults to self.max_steps_per_episode
- **initial_state** – State for the environment to be placed in to start the episode (used to force a deterministic start from anywhere in the environment rather than the typical start position)

Returns Object containing statistics about the episodes (rewards, number of steps, etc.)

Return type *EpisodeStatistics*

class lrl.solvers.**ValueIteration**(*env*, *value_function_initial_value=0.0*, ***kwargs*)
Bases: lrl.solvers.base_solver.BaseSolver

Solver for value iteration

Implemented as per Sutton and Barto's Reinforcement Learning (http://www.incompleteideas.net/book/RLbook2018.pdf, page 82).

Notes

See also BaseSolver for additional attributes, members, and arguments (missing here due to Sphinx bug with inheritance in docs)

Examples

See examples directory

Parameters

- **value_function_initial_value** (*float*) – Value to initialize all elements of the value function to
- **BaseSolver class for additional** (*See*) –

Returns None

value = None
Space-efficient dict-like storage of the current and all former value functions

Type *DictWithHistory*

iterate()
Perform a single iteration of value iteration, updating self.value and storing metadata about the iteration.

Side Effects:

- self.value: Updated to the newest estimate of the value function
- self.policy: Updated to the greedy policy according to the value function estimate
- self.iteration: Increment iteration counter by 1
- self.iteration_data: Add new record to iteration data store

Returns None

converged()
Returns True if solver is converged.

Test convergence by comparing the latest value function delta_max to the convergence tolerance

Returns Convergence status (True=converged)

Return type bool

init_policy(*init_type=None*)
Initialize self.policy, which is a dictionary-like DictWithHistory object for storing current and past policies

Parameters **init_type** (*None, str*) – Method used for initializing policy. Can be any of:

- None: Uses value in self.policy_init_type
- zeros: Initialize policy to all 0's (first action)

- **random: Initialize policy to a random action (action indices are random integer from** [0, len(self.env.P[this_state])], where P is the transition matrix and P[state] is a list of all actions available in the state)

Side Effects: If init_type is specified as argument, it is also stored to self.policy_init_type (overwriting previous value)

Returns None

iterate_to_convergence(*raise_if_not_converged=None*, *score_while_training=None*)

Perform self.iterate repeatedly until convergence, optionally scoring the current policy periodically

Side Effects: Many, but depends on the subclass of the solver's .iterate()

Parameters

- **raise_if_not_converged** (*bool*) – If true, will raise an exception if convergence is not reached before hitting maximum number of iterations. If None, uses self.raise_if_not_converged
- **score_while_training** (*bool, dict, None*) – If None, use self.score_while_training. Else, accepts inputs of same format as accepted for score_while_training solver inputs

Returns None

run_policy(*max_steps=None*, *initial_state=None*)

Perform a walk (episode) through the environment using the current policy

Side Effects:

- self.env will be reset and optionally then forced into initial_state

Parameters

- **max_steps** – Maximum number of steps to be taken in the walk (step 0 is taken to be entering initial state) If None, defaults to self.max_steps_per_episode
- **initial_state** – State for the environment to be placed in to start the walk (used to force a deterministic start from anywhere in the environment rather than the typical start position)

Returns

tuple containing:

- **states** (*list*): boolean indicating if the episode was terminal according to the environment
- **rewards** (*list*): list of rewards obtained during the episode (rewards[0] == 0 as step 0 is simply starting the game)
- **is_terminal** (*bool*): Boolean denoting whether the environment returned that the episode terminated naturally

Return type (tuple)

score_policy(*iters=500*, *max_steps=None*, *initial_state=None*)

Score the current policy by performing iters greedy episodes in the environment and returning statistics

Side Effects: self.env will be reset more side effects

more side effects

Parameters

- **iters** – Number of episodes in the environment
- **max_steps** – Maximum number of steps allowed per episode. If None, defaults to self.max_steps_per_episode
- **initial_state** – State for the environment to be placed in to start the episode (used to force a deterministic start from anywhere in the environment rather than the typical start position)

Returns Object containing statistics about the episodes (rewards, number of steps, etc.)

Return type *EpisodeStatistics*

class lrl.solvers.**QLearning**(*env*, *value_function_tolerance=0.1*, *alpha=None*, *epsilon=None*, *max_iters=2000*, *min_iters=250*, *num_episodes_for_convergence=20*, ***kwargs*)

Bases: lrl.solvers.base_solver.BaseSolver

Solver class for Q-Learning

Notes

See also BaseSolver for additional attributes, members, and arguments (missing due here to Sphinx bug with inheritance in docs)

Examples

See examples directory

Parameters

- **alpha** (*float, dict*) – (OPTIONAL)
  - If None, default linear decay schedule applied, decaying from 0.1 at iter 0 to 0.025 at max iter
  - If float, interpreted as a constant alpha value
  - If dict, interpreted as specifications to a decay function as defined in decay_functions()
- **epsilon** (*float, dict*) – (OPTIONAL)
  - If None, default linear decay schedule applied, decaying from 0.25 at iter 0 to 0.05 at max iter
  - If float, interpreted as a constant epsilon value
  - If dict, interpreted as specifications to a decay function as defined in decay_functions()
- **num_episodes_for_convergence** (*int*) – Number of consecutive episodes with delta_Q < tolerance to say a solution is converged
- ****kwargs** – Other arguments passed to BaseSolver

Returns None

transitions = None

Counter for number of transitions experienced during all learning

Type int

q = None
Space-efficient dict-like storage of the current and all former q functions

Type *DictWithHistory*

iteration_data = None
Data store for iteration data

Overloads BaseSolver's iteration_data attribute with one that includes more fields

Type *GeneralIterationData*

episode_statistics = None
Data store for statistics from training episodes

Type *EpisodeStatistics*

num_episodes_for_convergence = None
Number of consecutive episodes with delta_Q < tolerance to say a solution is converged

Type int

_policy_improvement(*states=None*)
Update the policy to be greedy relative to the most recent q function

Side Effects: self.policy: Updated to be greedy relative to self.q

Parameters **states** – List of states to update. If None, all states will be updated

Returns None

step(*count_transition=True*)
Take and learn from a single step in the environment.

Applies the typical Q-Learning approach to learn from the experienced transition

Parameters **count_transition** (*bool*) – If True, increment transitions counter self.transitions. Else, do not.

Returns

tuple containing:

- **transition** (*tuple*): Tuple of (state, reward, next_state, is_terminal)
- **delta_q** (*float*): The (absolute) change in q caused by this step

Return type (tuple)

iterate()
Perform and learn from a single episode in the environment (one walk from start to finish)

Side Effects:

- self.value: Updated to the newest estimate of the value function
- self.policy: Updated to the greedy policy according to the value function estimate
- self.iteration: Increment iteration counter by 1
- self.iteration_data: Add new record to iteration data store
- self.env: Reset and then walked through

Returns None

choose_epsilon_greedy_action(*state*, *epsilon=None*)
Return an action chosen by epsilon-greedy scheme based on the current estimate of Q

Parameters

- **state** (*int, tuple*) – Descriptor of current state in environment
- **epsilon** – Optional. If None, self.epsilon is used

Returns action chosen

Return type int or tuple

converged()
Returns True if solver is converged.

Returns Convergence status (True=converged)

Return type bool

get_q_at_state(*state*)
Returns a numpy array of q values at the current state in the same order as the standard action indexing :param state: Descriptor of current state in environment :type state: int, tuple

Returns Numpy array of q for all actions

Return type np.array

init_policy(*init_type=None*)
Initialize self.policy, which is a dictionary-like DictWithHistory object for storing current and past policies

Parameters **init_type** (*None, str*) – Method used for initializing policy. Can be any of:

- None: Uses value in self.policy_init_type
- zeros: Initialize policy to all 0's (first action)
- **random: Initialize policy to a random action (action indices are random integer from** [0, len(self.env.P[this_state])], where P is the transition matrix and P[state] is a list of all actions available in the state)

Side Effects: If init_type is specified as argument, it is also stored to self.policy_init_type (overwriting previous value)

Returns None

init_q(*init_val=0.0*)
Initialize self.q, a dict-like DictWithHistory object for storing the state-action value function q

Parameters **init_val** (*float*) – Value to give all states in the initialized q

Returns None

iterate_to_convergence(*raise_if_not_converged=None*, *score_while_training=None*)
Perform self.iterate repeatedly until convergence, optionally scoring the current policy periodically

Side Effects: Many, but depends on the subclass of the solver's .iterate()

Parameters

- **raise_if_not_converged** (*bool*) – If true, will raise an exception if convergence is not reached before hitting maximum number of iterations. If None, uses self.raise_if_not_converged

- **score_while_training** (*bool, dict, None*) – If None, use self.score_while_training. Else, accepts inputs of same format as accepted for score_while_training solver inputs

Returns None

run_policy(*max_steps=None*, *initial_state=None*)

Perform a walk (episode) through the environment using the current policy

Side Effects:

- self.env will be reset and optionally then forced into initial_state

Parameters

- **max_steps** – Maximum number of steps to be taken in the walk (step 0 is taken to be entering initial state) If None, defaults to self.max_steps_per_episode
- **initial_state** – State for the environment to be placed in to start the walk (used to force a deterministic start from anywhere in the environment rather than the typical start position)

Returns

tuple containing:

- **states** (*list*): boolean indicating if the episode was terminal according to the environment
- **rewards** (*list*): list of rewards obtained during the episode (rewards[0] == 0 as step 0 is simply starting the game)
- **is_terminal** (*bool*): Boolean denoting whether the environment returned that the episode terminated naturally

Return type (tuple)

score_policy(*iters=500*, *max_steps=None*, *initial_state=None*)

Score the current policy by performing iters greedy episodes in the environment and returning statistics

Side Effects: self.env will be reset more side effects

more side effects

Parameters

- **iters** – Number of episodes in the environment
- **max_steps** – Maximum number of steps allowed per episode. If None, defaults to self.max_steps_per_episode
- **initial_state** – State for the environment to be placed in to start the episode (used to force a deterministic start from anywhere in the environment rather than the typical start position)

Returns Object containing statistics about the episodes (rewards, number of steps, etc.)

Return type *EpisodeStatistics*

property alpha

Returns value of alpha at current iteration

property epsilon

Returns value of epsilon at current iteration

class lrl.solvers.**BaseSolver**(*env*, *gamma=0.9*, *value_function_tolerance=0.001*, *policy_init_mode='zeros'*, *max_iters=500*, *min_iters=2*, *max_steps_per_episode=100*, *score_while_training=False*, *raise_if_not_converged=False*)

Bases: object

Base class for solvers

Examples

See examples directory

Parameters

- **env** – Environment instance, such as from RaceTrack() or RewardingFrozenLake()
- **gamma** (*float*) – Discount factor
- **value_function_tolerance** (*float*) – Tolerance for convergence of value function during solving (also used for Q (state-action) value function tolerance
- **policy_init_mode** (*str*) – Initialization mode for policy. See init_policy() for more detail
- **max_iters** (*int*) – Maximum number of iterations to solve environment
- **min_iters** (*int*) – Minimum number of iterations before checking for solver convergence
- **raise_if_not_converged** (*bool*) – If True, will raise exception when environment hits max_iters without convergence. If False, a warning will be logged.
- **max_steps_per_episode** (*int*) – Maximum number of steps allowed per episode (helps when evaluating policies that can lead to infinite walks)
- **score_while_training** (*dict, bool*) – Dict specifying whether the policy should be scored during training (eg: test how well a policy is doing every N iterations).

  If dict, must be of format:

  – n_trains_per_eval (int): Number of training iters between evaluations

  – n_evals (int): Number of episodes for a given policy evaluation

  If True, score with default settings of:

  – n_trains_per_eval: 500

  – n_evals: 500

  If False, do not score during training.

Returns None

env = None

Environment being solved

Type *Racetrack*, RewardingFrozenLakeEnv

policy = None

Space-efficient dict-like storage of the current and all former policies.

Type *DictWithHistory*

iteration_data = None
Data describing iteration results during solving of the environment.

Fields include:

- time: time for this iteration
- delta_max: maximum change in value function for this iteration
- policy_changes: number of policy changes this iteration
- converged: boolean denoting if solution is converged after this iteration

Type *GeneralIterationData*

scoring_summary = None
Summary data from scoring runs computed during training if score_while_training == True

Fields include:

- reward_mean: mean reward obtained during a given scoring run

Type *GeneralIterationData*

scoring_episode_statistics = None
Detailed scoring data from scoring runs held as a dict of EpisodeStatistics objects.

Data is indexed by iteration number (from scoring_summary)

Type dict, *EpisodeStatistics*

init_policy(*init_type=None*)
Initialize self.policy, which is a dictionary-like DictWithHistory object for storing current and past policies

Parameters **init_type** (*None, str*) – Method used for initializing policy. Can be any of:

- None: Uses value in self.policy_init_type
- zeros: Initialize policy to all 0's (first action)
- **random: Initialize policy to a random action (action indices are random integer from** [0, len(self.env.P[this_state])], where P is the transition matrix and P[state] is a list of all actions available in the state)

Side Effects: If init_type is specified as argument, it is also stored to self.policy_init_type (overwriting previous value)

Returns None

iterate()
Perform the a single iteration of the solver.

This may be an iteration through all states in the environment (like in policy iteration) or obtaining and learning from a single experience (like in Q-Learning)

This method should update self.value and may update self.policy, and also commit iteration statistics to self.iteration_data. Unless the subclass implements a custom self.converged, self.iteration_data should include a boolean entry for "converged", which is used by the default converged() function.

Returns None

iterate_to_convergence(*raise_if_not_converged=None*, *score_while_training=None*)
Perform self.iterate repeatedly until convergence, optionally scoring the current policy periodically

Side Effects: Many, but depends on the subclass of the solver's .iterate()

Parameters

- **raise_if_not_converged** (*bool*) – If true, will raise an exception if convergence is not reached before hitting maximum number of iterations. If None, uses self.raise_if_not_converged
- **score_while_training** (*bool, dict, None*) – If None, use self.score_while_training. Else, accepts inputs of same format as accepted for score_while_training solver inputs

Returns None

converged()
Returns True if solver is converged.

This may be custom for each solver, but as a default it checks whether the most recent iteration_data entry has converged==True

Returns Convergence status (True=converged)

Return type bool

run_policy(*max_steps=None*, *initial_state=None*)
Perform a walk (episode) through the environment using the current policy

Side Effects:

- self.env will be reset and optionally then forced into initial_state

Parameters

- **max_steps** – Maximum number of steps to be taken in the walk (step 0 is taken to be entering initial state) If None, defaults to self.max_steps_per_episode
- **initial_state** – State for the environment to be placed in to start the walk (used to force a deterministic start from anywhere in the environment rather than the typical start position)

Returns

tuple containing:

- **states** (*list*): boolean indicating if the episode was terminal according to the environment
- **rewards** (*list*): list of rewards obtained during the episode (rewards[0] == 0 as step 0 is simply starting the game)
- **is_terminal** (*bool*): Boolean denoting whether the environment returned that the episode terminated naturally

Return type (tuple)

score_policy(*iters=500*, *max_steps=None*, *initial_state=None*)
Score the current policy by performing iters greedy episodes in the environment and returning statistics

Side Effects: self.env will be reset more side effects

more side effects

Parameters

- **iters** – Number of episodes in the environment

- **max_steps** – Maximum number of steps allowed per episode. If None, defaults to self.max_steps_per_episode
- **initial_state** – State for the environment to be placed in to start the episode (used to force a deterministic start from anywhere in the environment rather than the typical start position)

Returns Object containing statistics about the episodes (rewards, number of steps, etc.)

Return type *EpisodeStatistics*

class lrl.solvers.**BaseSolver**(*env*, *gamma=0.9*, *value_function_tolerance=0.001*, *policy_init_mode='zeros'*, *max_iters=500*, *min_iters=2*, *max_steps_per_episode=100*, *score_while_training=False*, *raise_if_not_converged=False*)

Bases: `object`

Base class for solvers

Examples

See examples directory

Parameters

- **env** – Environment instance, such as from RaceTrack() or RewardingFrozenLake()
- **gamma** (*float*) – Discount factor
- **value_function_tolerance** (*float*) – Tolerance for convergence of value function during solving (also used for Q (state-action) value function tolerance
- **policy_init_mode** (*str*) – Initialization mode for policy. See init_policy() for more detail
- **max_iters** (*int*) – Maximum number of iterations to solve environment
- **min_iters** (*int*) – Minimum number of iterations before checking for solver convergence
- **raise_if_not_converged** (*bool*) – If True, will raise exception when environment hits max_iters without convergence. If False, a warning will be logged.
- **max_steps_per_episode** (*int*) – Maximum number of steps allowed per episode (helps when evaluating policies that can lead to infinite walks)
- **score_while_training** (*dict, bool*) – Dict specifying whether the policy should be scored during training (eg: test how well a policy is doing every N iterations).

  If dict, must be of format:

  – n_trains_per_eval (int): Number of training iters between evaluations

  – n_evals (int): Number of episodes for a given policy evaluation

  If True, score with default settings of:

  – n_trains_per_eval: 500

  – n_evals: 500

  If False, do not score during training.

Returns None

env = None
Environment being solved

Type *Racetrack*, RewardingFrozenLakeEnv

policy = None
Space-efficient dict-like storage of the current and all former policies.

Type *DictWithHistory*

iteration_data = None
Data describing iteration results during solving of the environment.

Fields include:

- time: time for this iteration
- delta_max: maximum change in value function for this iteration
- policy_changes: number of policy changes this iteration
- converged: boolean denoting if solution is converged after this iteration

Type *GeneralIterationData*

scoring_summary = None
Summary data from scoring runs computed during training if score_while_training == True

Fields include:

- reward_mean: mean reward obtained during a given scoring run

Type *GeneralIterationData*

scoring_episode_statistics = None
Detailed scoring data from scoring runs held as a dict of EpisodeStatistics objects.

Data is indexed by iteration number (from scoring_summary)

Type dict, *EpisodeStatistics*

init_policy(*init_type=None*)
Initialize self.policy, which is a dictionary-like DictWithHistory object for storing current and past policies

Parameters **init_type** (*None, str*) – Method used for initializing policy. Can be any of:

- None: Uses value in self.policy_init_type
- zeros: Initialize policy to all 0's (first action)
- **random: Initialize policy to a random action (action indices are random integer from** [0, len(self.env.P[this_state])], where P is the transition matrix and P[state] is a list of all actions available in the state)

Side Effects: If init_type is specified as argument, it is also stored to self.policy_init_type (overwriting previous value)

Returns None

iterate()
Perform the a single iteration of the solver.

This may be an iteration through all states in the environment (like in policy iteration) or obtaining and learning from a single experience (like in Q-Learning)

This method should update self.value and may update self.policy, and also commit iteration statistics to self.iteration_data. Unless the subclass implements a custom self.converged, self.iteration_data should include a boolean entry for "converged", which is used by the default converged() function.

Returns None

iterate_to_convergence(*raise_if_not_converged=None*, *score_while_training=None*)

Perform self.iterate repeatedly until convergence, optionally scoring the current policy periodically

Side Effects: Many, but depends on the subclass of the solver's .iterate()

Parameters

- **raise_if_not_converged** (*bool*) – If true, will raise an exception if convergence is not reached before hitting maximum number of iterations. If None, uses self.raise_if_not_converged
- **score_while_training** (*bool, dict, None*) – If None, use self.score_while_training. Else, accepts inputs of same format as accepted for score_while_training solver inputs

Returns None

converged()

Returns True if solver is converged.

This may be custom for each solver, but as a default it checks whether the most recent iteration_data entry has converged==True

Returns Convergence status (True=converged)

Return type bool

run_policy(*max_steps=None*, *initial_state=None*)

Perform a walk (episode) through the environment using the current policy

Side Effects:

- self.env will be reset and optionally then forced into initial_state

Parameters

- **max_steps** – Maximum number of steps to be taken in the walk (step 0 is taken to be entering initial state) If None, defaults to self.max_steps_per_episode
- **initial_state** – State for the environment to be placed in to start the walk (used to force a deterministic start from anywhere in the environment rather than the typical start position)

Returns

tuple containing:

- **states** (*list*): boolean indicating if the episode was terminal according to the environment
- **rewards** (*list*): list of rewards obtained during the episode (rewards[0] == 0 as step 0 is simply starting the game)
- **is_terminal** (*bool*): Boolean denoting whether the environment returned that the episode terminated naturally

Return type (tuple)

score_policy(*iters=500*, *max_steps=None*, *initial_state=None*)
Score the current policy by performing iters greedy episodes in the environment and returning statistics

Side Effects: self.env will be reset more side effects

more side effects

Parameters

- **iters** – Number of episodes in the environment
- **max_steps** – Maximum number of steps allowed per episode. If None, defaults to self.max_steps_per_episode
- **initial_state** – State for the environment to be placed in to start the episode (used to force a deterministic start from anywhere in the environment rather than the typical start position)

Returns Object containing statistics about the episodes (rewards, number of steps, etc.)

Return type *EpisodeStatistics*

3.2 Environments

class lrl.environments.**Racetrack**(*track=None*, *x_vel_limits=None*, *y_vel_limits=None*, *x_accel_limits=None*, *y_accel_limits=None*, *max_total_accel=2*)
Bases: gym.envs.toy_text.discrete.DiscreteEnv

A car-race-like environment that uses location and velocity for state and acceleration for actions, in 2D

Loosely inspired by the Racetrack example of Sutton and Barto's Reinforcement Learning (Exercise 5.8, http://www.incompleteideas.net/book/the-book.html)

The objective of this environment is to traverse a racetrack from a start location to any goal location. Reaching a goal location returns a large reward and terminates the episode, whereas landing on a grass location returns a large negative reward and terminates the episode. All non-terminal transitions return a small negative reward. Oily road surfaces are non-terminal but also react to an agent's action stochastically, sometimes causing an Agent to "slip" whereby their requested action is ignored (interpreted as if a=(0,0)).

The tiles in the environment are:

- (blank): Clean open (deterministic) road
- O: Oily (stochastic) road
- G: (terminal) grass
- S: Starting location (agent starts at a random starting location). After starting, S tiles behave like open road
- F: Finish location(s) (agent must reach any of these tiles to receive positive reward

The state space of the environment is described by xy location and xy velocity (with maximum velocity being a user-specified parameter). For example, s=(3, 5, 1, -1) means the Agent is currently in the x=3, y=5 location with Vx=1, Vy=-1.

The action space of the environment is xy acceleration (with maximum acceleration being a user-specified parameter). For example, a=(-2, 1) means ax=-2, ay=-1. Transitions are determined by the current velocity as well as the requested acceleration (with a cap set by Vmax of the environment), for example:

- s=(3, 5, 1, -1), a=(-3, 1) –> s_prime=(1, 5, -2, 0)

But if vx_max == +-1 then:

- s=(3, 5, 1, -1), a=(-3, 1) –> s_prime=(2, 5, -1, 0)

Note that sign conventions for location are:

- x: 0 at leftmost column, positive to the right
- y: 0 at bottommost row, positive up

Parameters

- **track** (*list*) – List of strings describing the track (see racetrack_tracks.py for examples)
- **x_vel_limits** (*tuple*) – (OPTIONAL) Tuple of (min, max) valid acceleration in x. Default is (-2, 2).
- **y_vel_limits** (*tuple*) – (OPTIONAL) Tuple of (min, max) valid acceleration in y. Default is (-2, 2).
- **x_accel_limits** (*tuple*) – (OPTIONAL) Tuple of (min, max) valid acceleration in x. Default is (-2, 2).
- **y_accel_limits** (*tuple*) – (OPTIONAL) Tuple of (min, max) valid acceleration in y. Default is (-2, 2).
- **max_total_accel** (*int*) – (OPTIONAL) Integer maximum total acceleration in one action. Total acceleration is computed by abs(x_a)+abs(y_a), representing the sum of change in acceleration in both directions Default is infinite (eg: any accel described by x and y limits)

Notes

See also discrete.DiscreteEnv for additional attributes, members, and arguments (missing due here to Sphinx bug with inheritance in docs)

DOCTODO: Add examples

track = None
List of strings describing track or the string name of a default track

Type list, str

desc = None
Numpy character array of the track (better for printing on screen/accessing track at xy locations)

Type np.array

color_map = None
Map from grid tile type to display color

Type dict

index_to_state = None
Attribute to map from state index to full tuple describing state

Ex: index_to_state[state_index] -> state_tuple

Type list

state_to_index = None
Attribute to map from state tuple to state index

Ex: state_to_index[state_tuple] -> state_index

Type dict

is_location_terminal = None
no rewards/transitions leading out of state).

Keyed by state tuple

Type dict

Type Attribute to map whether a state is terminal (eg

s = None
Current state (inherited from parent)

Type int, tuple

reset()
Reset the environment to a random starting location

Returns None

render(*mode='human'*, *current_location='*'*)
Render the environment.

> **Warning:** This method does not follow the prototype of it's parent. It is presently a very simple version for printing the environment's current state to the screen

Parameters

- **mode** – (NOT USED)
- **current_location** – Character to denote the current location

Returns None

step(*a*)
Take a step in the environment.

This wraps the parent object's step(), interpreting integer actions as mapped to human-readable actions

Parameters **a** (*tuple, int*) – Action to take, either as an integer (0..nA-1) or true action (tuple of (x_accel,y_accel))

Returns Next state, either as a tuple or int depending on type of state used

close()
Override _close in your subclass to perform any necessary cleanup.

Environments will automatically close() themselves when garbage collected or when the program exits.

seed(*seed=None*)
Sets the seed for this env's random number generator(s).

> **Note:** Some environments use multiple pseudorandom number generators. We want to capture all such seeds used in order to ensure that there aren't accidental correlations between multiple generators.

Returns

Returns the list of seeds used in this env's random number generators. The first value in the list should be the "main" seed, or the value which a reproducer should pass to 'seed'. Often, the main seed equals the provided 'seed', but this won't be true if seed=None, for example.

Return type list<bigint>

property unwrapped
Completely unwrap this env.

Returns The base non-wrapped gym.Env instance

Return type gym.Env

3.3 Experiment Runners

lrl.utils.experiment_runners.**run_experiment**(*env*, *params*, *output_path*)
Run a single experiment (env/solver combination), outputing results to a given location

FUTURE: Improve easy reproducibility by outputting a settings file or similar? Could use gin-config or just output params. Outputting params doesn't cover env though...

Parameters

- **env** – An instanced environment object (eg: Racetrack(or RewardingFrozenLake())
- **params** – A dictionary of solver parameters for this run
- **output_path** (*str*) – Path to output data (plots and csvs)

Output to output_path:

- **iteration_data.csv**: Data about each solver iteration (shows how long each iteration took, how quickly the solver converged, etc.)
- **solver_results*.png**: Images of policy (and value for planners). If environment state is defined by xy alone, a single image is returned. Else, an image for each additional state is returned (eg: for state = (x, y, vx, vy), plots of solver_results_vx_vy.png are returned for each (vx, vy))
- **scored_episodes.csv** and **scored_episodes.png**: Detailed data for each episode taken during the final scoring, and a composite image of those episodes in the environment
- **intermediate_scoring_results.csv**: Summary data from each evaluation during training (shows history of how the solver improved over time)
- **intermediate_scoring_results_*.png**: Composite images of the intermediate scoring results taken during training, indexed by the iteration at which they were produced
- **training_episodes.csv** and **training_episodes.png**: Detailed data for each episode taken during training, and an composite image of those episodes exploring the environment (only available for an explorational learner like Q-Learning)

Returns

dict containing:

- **solver** (*BaseSolver*, *ValueIteration*, *PolicyIteration*, *QLearner*): Fully populated solver object (after solving env)

- **scored_results** (*EpisodeStatistics*): EpisodeStatistics object of results from scoring the final policy
- **solve_time** (*float*): Time in seconds used to solve the env (eg: run solver.iterate_to_convergence())

Return type (dict)

lrl.utils.experiment_runners.**run_experiments**(*environments*, *solver_param_grid*, *output_path='./output/'*)

Runs a set of experiments defined by param_grid, writing results to output_path

Parameters

- **environments** (*list*) – List of instanced environments
- **solver_param_grid** (*dict*) – Solver parameters in suitable form for sklearn.model_selection.ParameterGrid
- **output_path** (*str*) – Relative path to which results will be output

Output to output_path:

- For each environment:
  - **env_name/grid_search_summary.csv**: high-level summary of results for this env
  - **env_name/case_name**: Directory with detailed results for each env/case combination See run_experiment for details on casewise output)

Returns None

3.4 Plotting

lrl.utils.plotting.**plot_solver_convergence**(*solver*, ***kwargs*)

Convenience binding to plot convergence statistics for a solver object.

Also useful as a recipe for custom plotting.

Parameters

- **solver** (*BaseSolver (or child)*) – Solver object to be plotted
- **args** (*Other*) – See plot_solver_convergence_from_df()

Returns Matplotlib axes object

Return type Axes

lrl.utils.plotting.**plot_solver_convergence_from_df**(*df*, *y='delta_max'*, *y_label=None*, *x='iteration'*, *x_label='Iteration'*, *label=None*, *ax=None*, *savefig=None*, ***kwargs*)

Convenience binding to plot convergence statistics for a set of solver objects.

Also useful as a recipe for custom plotting.

Parameters

- **df** (*pandas.DataFrame*) – DataFrame with solver convergence data
- **y** (*str*) – Convergence statistic to be plotted (eg: delta_max, delta_mean, time, or policy_changes)

- **y_label** (*str*) – Optional label for y_axis (if omitted, will use y as default name unless axis is already labeled)
- **x** (*str*) – X axis data (typically 'iteration', but could be any convergence data)
- **x_label** (*str*) – Optional label for x_axis (if omitted, will use 'Iteration')
- **label** (*str*) – Optional label for the data set (shows up in axes legend)
- **ax** (*Axes*) – Optional Matplotlib Axes object to add this line to
- **savefig** (*str*) – Optional filename to save the figure to
- **kwargs** – Additional args passed to matplotlib's plot

Returns Matplotlib axes object

Return type Axes

lrl.utils.plotting.**plot_env**(*env*, *ax=None*, *edgecolor='k'*, *resize_figure=True*, *savefig=None*)

Plot the map of an environment

Parameters

- **env** – Environment to plot
- **ax** (*axes*) – (Optional) Axes object to plot on
- **edgecolor** (*str*) – Color of the edge of each grid square (matplotlib format)
- **resize_figure** (*bool*) – If true, resize the figure to:
  - width = 0.5 * n_cols inches
  - height = 0.5 * n_rows inches
- **savefig** (*str*) – If not None, save the figure to this filename

Returns Matplotlib axes object

Return type Axes

lrl.utils.plotting.**plot_solver_results**(*env*, *solver=None*, *policy=None*, *value=None*, *savefig=None*, ***kwargs*)

Convenience function to plot results from a solver over the environment map

Input can be using a BaseSolver or child object, or by specifying policy and/or value directly via dict or DictWithHistory.

See plot_solver_result() for more info on generation of individual plots and additional arguments for color/precision.

Parameters

- **env** – Augmented OpenAI Gym-like environment object
- **solver** (BaseSolver) – Solver object used to solve the environment
- **policy** (*dict,* DictWithHistory) – Policy for the environment, keyed by integer state-index or tuples of state
- **value** (*dict,* DictWithHistory) – Value function for the environment, keyed by integer state-index or tuples of state
- **savefig** (*str*) – If not None, save figures to this name. For cases with multiple policies per grid square, this will be the suffix on the name (eg: for policy at Vx=1, Vy=2, we get name of savefig_1_2.png)
- ****kwargs** (*dict*) – Other arguments passed to plot_solver_result

Returns list of Matplotlib Axes for the plots

Return type list

lrl.utils.plotting.**plot_policy**(*env*, *policy*, ***kwargs*)
Convenience binding for plot_policy_or_value(). See plot_policy_or_value for more detail

lrl.utils.plotting.**plot_value**(*env*, *value*, ***kwargs*)
Convenience binding for plot_policy_or_value(). See plot_policy_or_value for more detail

lrl.utils.plotting.**plot_solver_result**(*env*, *policy=None*, *value=None*, *ax=None*, *add_env_to_plot=True*, *hide_terminal_locations=True*, *color='k'*, *title=None*, *savefig=None*, *size_policy='auto'*, *size_value='auto'*, *value_precision=2*)
Plot result for a single xy map using a numpy array of shaped policy and/or value

Parameters

- **env** (*Racetrack, FrozenLake, other environment*) – Instantiated environment object
- **policy** (*np.array*) – Policy for each grid square in the environment, in the same shape as env.desc For plotting environments where we have multiple states for a given grid square (eg for Racetrack), will call plotting for each given additional state (eg: for v=(0, 0), v=(1, 0), ..)
- **value** – (np.array): Value for each grid square in the environment, in the same shape as env.desc For plotting environments where we have multiple states for a given grid square (eg for Racetrack), will call plotting for each given additional state (eg: for v=(0, 0), v=(1, 0), ..)
- **ax** (*Axes*) – (OPTIONAL) Matplotlib axes object to plot to
- **add_env_to_plot** (*bool*) – If True, add the environment map to the axes before plotting policy using plot_env()
- **hide_terminal_locations** (*bool*) – If True, all known terminal locations will have no text printed (as policy here doesn't matter)
- **color** (*str*) – Matplotlib color string denoting color of the text for policy/value
- **title** (*str*) – (Optional) title added to the axes object
- **savefig** (*str*) – (Optional) string filename to output the figure to
- **size_policy** (*str, numeric*) – (Optional) Specification of text font size for policy printing. One of:
  - 'auto': Will automatically choose a font size based on the number of characters to be printed
  - *str* or *numeric*: Interpreted as a Matplotlib style font size designation
- **size_value** (*str, numeric*) – (Optional) Specification of text font size for value printing. Same interface as size_policy
- **value_precision** (*int*) – Precision of value function to be included on figures

Returns Matplotlib Axes object

lrl.utils.plotting.**plot_episodes**(*episodes*, *env=None*, *add_env_to_plot=True*, *max_episodes=100*, *alpha=None*, *color='k'*, *title=None*, *ax=None*, *savefig=None*)
Plot a list of episodes through an environment over a drawing of the environment

Parameters

- **episodes** (*list, EpisodeStatistics*) – Series of episodes to be plotted. If EpisodeStatistics instance, .episodes will be extracted
- **env** – Environment traversed
- **add_env_to_plot** (*bool*) – If True, use plot_env to plot the environment to the image
- **alpha** (*float*) – (Optional) alpha (transparency) used for plotting the episode. If left as None, a value will be chosen based on the number of episodes to be plotted
- **color** (*str*) – Matplotlib-style color designation
- **title** (*str*) – (Optional) Title to be added to the axes
- **ax** (*axes*) – (Optional) Matplotlib axes object to write the plot to
- **savefig** (*str*) – (Optional) string filename to output the figure to
- **max_episodes** (*int*) – Maximum number of episodes to add to the plot. If len(episodes) exceeds this value, randomly chosen episodes will be used

Returns Matplotlib Axes object with episodes plotted to it

lrl.utils.plotting.**plot_episode**(*episode*, *env=None*, *add_env_to_plot=True*, *alpha=None*, *color='k'*, *title=None*, *ax=None*, *savefig=None*)

Plot a single episode (walk) through the environment

Parameters

- **episode** (*list*) – List of states encountered in the episode
- **env** – Environment traversed
- **add_env_to_plot** (*bool*) – If True, use plot_env to plot the environment to the image
- **alpha** (*float*) – (Optional) alpha (transparency) used for plotting the episode.
- **color** (*str*) – Matplotlib-style color designation
- **title** (*str*) – (Optional) Title to be added to the axes
- **ax** (*axes*) – (Optional) Matplotlib axes object to write the plot to
- **savefig** (*str*) – (Optional) string filename to output the figure to

Returns Matplotlib Axes object with a single episode plotted to it

lrl.utils.plotting.**choose_text_size**(*n_chars*, *boxsize=1.0*)

Helper to choose an appropriate text size when plotting policies. Size is chosen based on length of text

Return is calibrated to something that typically looked nice in testing

Parameters

- **n_chars** – Text caption to be added to plot
- **boxsize** (*float*) – Size of box inside which text should print nicely. Used as a scaling factor. Default is 1 inch

Returns Matplotlib-style text size argument

lrl.utils.plotting.**policy_dict_to_array**(*env*, *policy_dict*)

Convert a policy stored as a dictionary into a dictionary of one or more policy numpy arrays shaped like env.desc

Can also be used for a value_dict.

policy_dict is a dictionary relating state to policy at that state in one of several forms. The dictionary can be keyed by state-index or a tuple of state (eg: (x, y, [other_state]), with x=0 in left column, y=0 in bottom row). If using tuples of state, state may be more than just x,y location as shown above, eg: (x, y, v_x, v_y). If len(state_tuple) > 2, we must plot each additional state separately.

Translate policy_dict into a policy_list_of_tuples of:

```
[(other_state_0, array_of_policy_at_other_state_0),
 (other_state_1, array_of_policy_at_other_state_1),
  ... ]
```

where the array_of_policy_at_other_state_* is in the same shape as env.desc (eg: cell [3, 2] of the array is the policy for the env.desc[3, 2] location in the env).

Examples

If state is described by tuples of (x, y) (where there is a single unique state for each grid location), eg:

```
policy_dict = {
    (0, 0): policy_0_0,
    (0, 1): policy_0_1,
    (0, 2): policy_0_2,
    ...
    (1, 0): policy_2_1,
    (1, 1): policy_2_1,
    ...
    (xmax, ymax): policy_xmax_ymax,
    }
```

then a single-element list is returned of the form:

```
returned = [
  (None, np_array_of_policy),
]
```

where np_array_of_policy is of the same shape as env.desc (eg: the map), with each element corresponding to the policy at that grid location (for example, cell [3, 2] of the array is the policy for the env.desc[3, 2] location in the env).

If state is described by tuples of (x, y, something_else, [more_something_else...]), for example if state = (x, y, Vx, Vy) like below:

```
policy_dict = {
    (0, 0, 0, 0): policy_0_0_0_0,
    (0, 0, 1, 0): policy_0_0_1_0,
    (0, 0, 0, 1): policy_0_0_0_1,
    ...
    (1, 0, 0, 0): policy_1_0_0_0,
    (1, 0, 0, 1): policy_1_0_0_1,
    ...
    (xmax, ymax, Vxmax, Vymax): policy_xmax_ymax_Vxmax_Vymax,
    }
```

then a list is returned of the form:

```
returned = [
#   (other_state, np_array_of_policies_for_this_other_state)
```

(continues on next page)

(continued from previous page)

```
    ((0, 0), np_array_of_policies_with_Vx-0_Vy-0),
    ((1, 0), np_array_of_policies_with_Vx-0_Vy-0),
    ((0, 1), np_array_of_policies_with_Vx-0_Vy-0),
    ...
    ((Vxmax, Vymax), np_array_of_policies_with_Vxmax_Vymax),
]
```

where each element corresponds to a different combination of all the non-location state. This means that each element of the list is:

```
(Identification_of_this_case, shaped_xy-grid_of_policies_for_this_case)
```

and can be easily plotted over the environment's map.

If policy_dict is keyed by state-index rather than state directly, the same logic as above still applies.

Notes

If using an environment (with policy keyed by either index or state) that has more than one unique state per grid location (eg: state has more than (x, y)), then environment must also have an index_to_state attribute to identify overlapping states. This constraint exists both for policies keyed by index or state, but the code could be refactored to avoid this limitation for state-keyed policies if required.

Parameters

- **env** – Augmented OpenAI Gym-like environment object
- **policy_dict** (*dict*) – Dictionary of policy for the environment, keyed by integer state-index or tuples of state

Returns list of (description, shaped_policy) elements as described above

lrl.utils.plotting.**get_ax**(*ax*)
Returns figure and axes objects associated with an axes, instantiating if input is None

3.5 Data Stores

class lrl.data_stores.**GeneralIterationData**(*columns=None*)
Bases: object

Class to store data about solver iterations

Data is stored as a list of dictionaries. This is a placeholder for more advanced storage. Class gives a minimal set of extra bindings for convenience.

The present object has no checks to ensure consistency between added records (all have same fields, etc.). If any columns are missing from an added record, outputting to a dataframe will result in Pandas treating these as missing columns from a record.

Parameters **columns** (*list*) – An optional list of column names for the data (if specified, this sets the order of the columns in any output Pandas DataFrame or csv)

DOCTODO: Add example of usage

columns = None
Column names used for data output.

If specified, this sets the order of any columns being output to Pandas DataFrame or csv

Type list

data = None
List of dictionaries representing records.

Intended to be internal in future, but public at present to give easy access to records for slicing

Type list

add(*d*)
Add a dictionary record to the data structure.

Parameters **d** (*dict*) – Dictionary of data to be stored

Returns None

get(*i=-1*)
Return the ith entry in the data store (index of storage is in order in which data is committed to this object)

Parameters **i** (*int*) – Index of data to return (can be any valid list index, including -1 and slices)

Returns ith entry in the data store

Return type dict

to_dataframe()
Returns the data structure as a Pandas DataFrame

Returns Pandas DataFrame of the data

Return type dataframe

to_csv(*filename*, ***kwargs*)
Write data structure to a csv via the Pandas DataFrame

Parameters

- **filename** (*str*) – Filename or full path to output data to
- **kwargs** (*dict*) – Optional arguments to be passed to DataFrame.to_csv()

Returns None

class lrl.data_stores.**DictWithHistory**(*timepoint_mode='explicit'*, *tolerance=1e-07*)
Bases: collections.abc.MutableMapping

Dictionary-like object that maintains a history of all changes, either incrementally or at set timepoints

This object has access like a dictionary, but stores data internally such that the user can later recreate the state of the data from a past timepoint.

The intended use of this object is to store large objects which are iterated on (such as value or policy functions) in a way that a history of changes can be reproduced without having to store a new copy of the object every time. For example, when doing 10000 episodes of Q-Learning in a grid world with 2500 states, we can retain the full policy history during convergence (eg: answer "what was my policy after episode 527") without keeping 10000 copies of a nearly-identical 2500 element numpy array or dict. The cost for this is some computation, although this generally has not been seen to be too significant (~10's of seconds for a large Q-Learning problem in testing)

Parameters

- **timepoint_mode** (*str*) – One of:
- **explicit** (***) – Timepoint incrementing is handled explicitly by the user (the timepoint only changes if the user invokes .update_timepoint()

- **implicit** (*) – Timepoint incrementing is automatic and occurs on every setting action, including redundant sets (setting a key to a value it already holds). This is useful for a timehistory of all sets to the object
- **tolerance** (*float*) – Absolute tolerance to test for when replacing values. If a value to be set is less than tolerance different from the current value, the current value is not changed.

Warning:

- Deletion of keys is not specifically supported. Deletion likely works for the most recent timepoint, but the history does not handle deleted keys properly
- Numeric data may work best due to how new values are compared to existing data, although tuples have also been tested. See __setitem__ for more detail

DOCTODO: Add example

timepoint_mode = None
See Parameters for definition

Type str

current_timepoint = None
Timepoint that will be written to next

Type int

__getitem__(*key*)
Return the most recent value for key

Returns Whatever is contained in ._data[key][-1][-1] (return only the value from the most recent timepoint, not the timepoint associated with it)

__setitem__(*key*, *value*)
Set the value at a key if it is different from the current data stored at key

Data stored here is stored under the self.current_timepoint.

Difference between new and current values is assessed by testing:

- new_value == old_value
- np.isclose(new_value, old_value)

where if neither returns True, the new value is taken to be different from the current value

Side Effects: If timepoint_mode == 'implicit', self.current_timepoint will be incremented after setting data

Parameters

- **key** (*immutable*) – Key under which data is stored
- **value** – Value to store at key

Returns None

update(*d*)
Update this instance with a dictionary of data, d (similar to dict.update())

Keys in d that are present in this object overwrite the previous value. Keys in d that are missing in this object are added.

All data written from d is given the same timepoint (even if timepoint_mode=implicit) - the addition is treated as a single update to the object rather than a series of updates.

Parameters **d** (*dict*) – Dictionary of data to be added here

Returns None

get_value_history(*key*)
Returns a list of tuples of the value at a given key over the entire history of that key

Parameters **key** (*immutable*) – Any valid dictionary key

Returns

list containing tuples of:

- **timepoint** (*int*): Integer timepoint for this value
- **value** (*float*): The value of key at the corresponding timepoint

Return type (list)

get_value_at_timepoint(*key*, *timepoint=-1*)
Returns the value corresponding to a key at the timepoint that is closest to but not greater than timepoint

Raises a KeyError if key did not exist at timepoint. Raises an IndexError if no timepoint exists that applies

Parameters

- **key** (*immutable*) – Any valid dictionary key
- **timepoint** (*int*) – Integer timepoint to return value for. If negative, it is interpreted like typical python indexing (-1 means most recent, -2 means second most recent, ...)

Returns Value corresponding to key at the timepoint closest to but not over timepoint

Return type numeric

to_dict(*timepoint=-1*)
Return the state of the data at a given timepoint as a dict

Parameters **timepoint** (*int*) – Integer timepoint to return data as of. If negative, it is interpreted like typical python indexing (-1 means most recent, -2 means second most recent, ...)

Returns Data at timepoint

Return type dict

clear() → None. Remove all items from D.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

increment_timepoint()
Increments the timepoint at which the object is currently writing

Returns None

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

values() → an object providing a view on D's values

class lrl.data_stores.**EpisodeStatistics**
Bases: object

Container for statistics about a set of independent episodes through an environment, typically following one policy

Statistics are lazily computed and memorized

DOCTODO: Add example usage. show plot_episodes

rewards = None
List of the total reward for each episode (raw data)

Type list

episodes = None
List of all episodes passed to the data object (raw data)

Type list

steps = None
List of the total steps taken for each episode (raw data)

Type list

terminals = None
List of whether each input episode was terminal (raw data)

Type list

add(*reward*, *episode*, *terminal*)
Add an episode to the data store

Parameters

- **reward** (*float*) – Total reward from the episode
- **episode** (*list*) – List of states encoutered in the episode, including the starting and final state
- **terminal** (*bool*) – Boolean indicating if episode was terminal (did environment say episode has ended)

Returns None

get_statistic(*statistic='reward_mean'*, *index=-1*)
Return a lazily computed and memorized statistic about the rewards from episodes 0 to index

If the statistic has not been previous computed, it will be computed and returned. See .get_statistics() for list of statistics available

Side Effects: self.statistics[index] will be computed using self.compute() if it has not been already

Parameters

- **statistic** (*str*) – See .compute() for available statistics
- **index** (*int*) – Episode index for requested statistic

Notes

Statistics are computed for all episodes up to and including the requested statistic. For example if episodes have rewards of [1, 3, 5, 10], get_statistic('reward_mean', index=2) returns 3 (mean of [1, 3, 5]).

DOCTODO: Example usage (show getting some statistics)

Returns Value of the statistic requested

Return type int, float

get_statistics(*index=-1*)

Return a lazily computed and memorized dictionary of all statistics about episodes 0 to index

If the statistic has not been previous computed, it will be computed here.

Side Effects: self.statistics[index] will be computed using self.compute() if it has not been already

Parameters **index** (*int*) – Episode index for requested statistic

Returns

Details and statistics about this iteration, with keys:

Details about this iteration:

- **episode_index** (*int*): Index of episode
- **terminal** (*bool*): Boolean of whether this episode was terminal
- **reward** (*float*): This episode's reward (included to give easy access to per-iteration data)
- **steps** (*int*): This episode's steps (included to give easy access to per-iteration data)

Statistics computed for all episodes up to and including this episode:

- **reward_mean** (*float*):
- **reward_median** (*float*):
- **reward_std** (*float*):
- **reward_max** (*float*):
- **reward_min** (*float*):
- **steps_mean** (*float*):
- **steps_median** (*float*):
- **steps_std** (*float*):
- **steps_max** (*float*):
- **steps_min** (*float*):
- **terminal_fraction** (*float*):

Return type dict

compute(*index=-1*, *force=False*)

Compute and store statistics about rewards and steps for episodes up to and including the indexth episode

Side Effects: self.statistics[index] will be updated

Parameters

- **index** (*int or 'all'*) – If integer, the index of the episode for which statistics are computed. Eg: If index==3, compute the statistics (see get_statistics() for list) for the series of episodes from 0 up to and not including 3 (typical python indexing rules) If 'all', compute statistics for all indices, skipping any that have been previously computed unless force == True
- **force** (*bool*) – If True, always recompute statistics even if they already exist.

  If False, only compute if no previous statistics exist.

Returns None

to_dataframe(*include_episodes=False*)

Return a Pandas DataFrame of the episode statistics

See .get_statistics() for a definition of each column. Order of columns is set through self.statistics_columns

Parameters **include_episodes** (*bool*) – If True, add column including the entire episode for each iteration

Returns Pandas DataFrame

to_csv(*filename*, ***kwargs*)

Write statistics to csv via the Pandas DataFrame

See .get_statistics() for a definition of each column. Order of columns is set through self.statistics_columns

Parameters

- **filename** (*str*) – Filename or full path to output data to
- **kwargs** (*dict*) – Optional arguments to be passed to DataFrame.to_csv()

Returns None

3.6 Miscellaneous Utilities

class lrl.utils.misc.**Timer**

Bases: object

A Simple Timer class for timing code

start = None

timeit.default_timer object initialized at instantiation

elapsed()

Return the time elapsed since this object was instantiated, in seconds

Returns Time elapsed in seconds

Return type float

lrl.utils.misc.**print_dict_by_row**(*d*, *fmt='{key:20s}: {val:d}'*)

Print a dictionary with a little extra structure, printing a different key/value to each line.

Parameters

- **d** (*dict*) – Dictionary to be printed
- **fmt** (*str*) – Format string to be used for printing. Must contain key and val formatting references

Returns None

lrl.utils.misc.**count_dict_differences**(*d1*, *d2*, *keys=None*, *raise_on_missing_key=True*, *print_differences=False*)

Return the number of differences between two dictionaries. Useful to compare two policies stored as dictionaries.

Does not properly handle floats that are approximately equal. Mainly use for int and objects with __eq__

Optionally raise an error on missing keys (otherwise missing keys are counted as differences)

Parameters

- **d1** (*dict*) – Dictionary to compare
- **d2** (*dict*) – Dictionary to compare
- **keys** (*list*) – Optional list of keys to consider for differences. If None, all keys will be considered
- **raise_on_missing_key** (*bool*) – If true, raise KeyError on any keys not shared by both dictionaries
- **print_differences** (*bool*) – If true, print all differences to screen

Returns Number of differences between the two dictionaries

Return type int

lrl.utils.misc.**dict_differences**(*d1*, *d2*)

Return the maximum and mean of the absolute difference between all elements of two dictionaries of numbers

Parameters

- **d1** (*dict*) – Dictionary to compare
- **d2** (*dict*) – Dictionary to compare

Returns

tuple containing:

- *float*: Maximum elementwise difference
- *float*: Sum of elementwise differences

Return type tuple

lrl.utils.misc.**rc_to_xy**(*row*, *col*, *rows*)

Convert from (row, col) coordinates (eg: numpy array) to (x, y) coordinates (bottom left = 0,0)

(x, y) convention

- (0,0) in bottom left
- x +ve to the right
- y +ve up

(row,col) convention:

- (0,0) in top left
- row +ve down
- col +ve to the right

Parameters

- **row** (*int*) – row coordinate to be converted

- **col** (*int*) – col coordinate to be converted
- **rows** (*int*) – Total number of rows

Returns (x, y)

Return type tuple

lrl.utils.misc.**params_to_name**(*params*, *n_chars=4*, *sep='_'*, *first_fields=None*, *key_remap=None*)

Convert a mappable of parameters into a string for easy test naming

Warning: Currently includes hard-coded formatting that interprets keys named 'alpha' or 'epsilon'

Parameters

- **params** (*dict*) – Dictionary to convert to a string
- **n_chars** (*int*) – Number of characters per key to add to string. Eg: if key='abcdefg' and n_chars=4, output will be 'abcd'
- **sep** (*str*) – Separator character between fields (uses one of these between key and value, and two between different key-value pairs
- **first_fields** (*list*) – Optional list of keys to write ahead of other keys (otherwise, output order it sorted)
- **key_remap** (*list*) – List of dictionaries of {key_name: new_key_name} for rewriting keys into more readable strings

Returns

Return type str

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

l

INDEX

Symbols

A

B

C

D

E

G

I

K

L

N

P

Q

R

S

T

U

V