Mi - H13

February 9, 2017

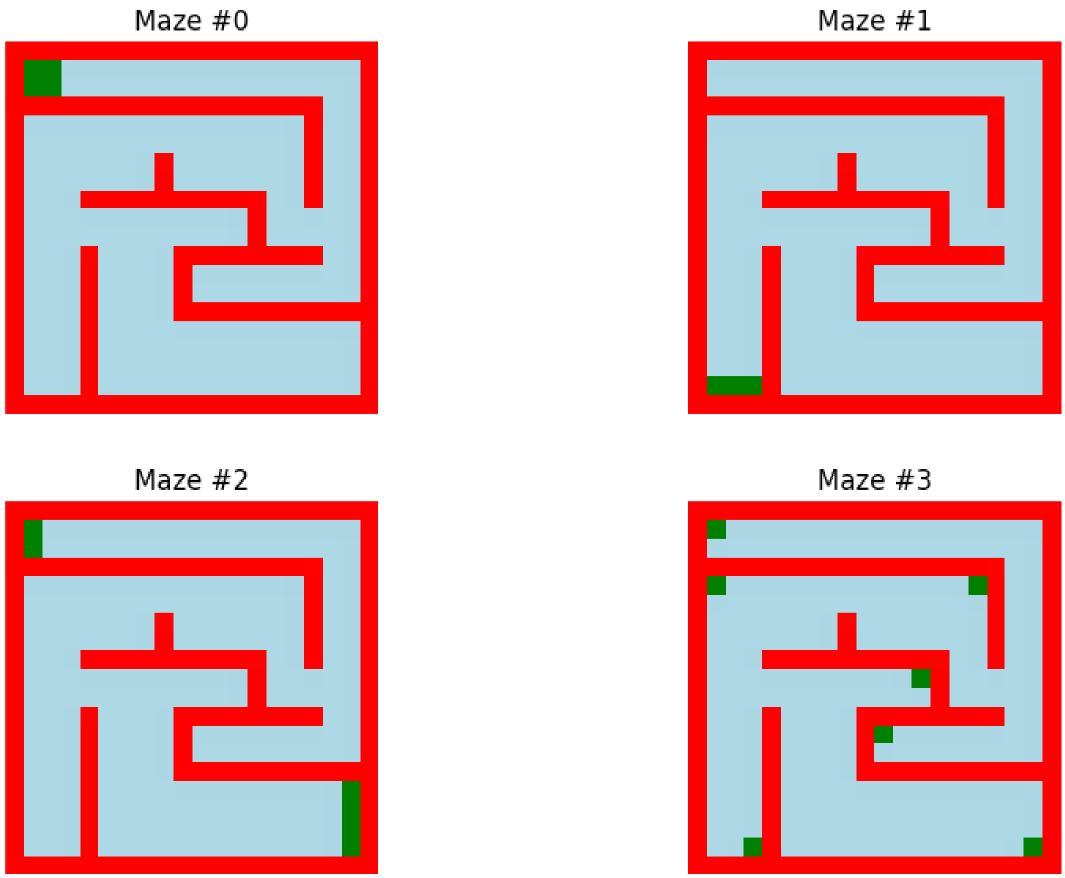
In [133]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors
from functools import reduce
import sklearn.datasets
import os
import re
%matplotlib inline
```

In [552]:
```
def transform(maze):
    # Transform a maze from the text file into a matplotlib friendly form
    return [[int(sign) for sign in line] for line in maze]

def reduce_mazes(list_, line):
    if list_ == '':
        list_ = [[line]]
    elif line == '':
        if list_[-1]:
            list_.append([])
    else:
        list_[-1].append(line)
    return list_

def replace_signs(lines):
    lines = (x.replace('\n', '') for x in lines)
    lines = (x.replace('X', '0') for x in lines)
    lines = (x.replace(' ', '1') for x in lines)
    lines = (x.replace('#', '2') for x in lines)
    lines = (x.replace('>', '3') for x in lines)
    lines = (x.replace('v', '4') for x in lines)
    lines = (x.replace('<', '5') for x in lines)
    lines = (x.replace('^', '6') for x in lines)
    return list(lines)

def read_mazes():
    lines = None
    with open('mazes.txt') as file:
        lines = file.readlines()
```

```
        lines = replace_signs(lines)
        mazes = reduce(reduce_mazes, lines)
        # Remove empty mazes
        mazes = [x for x in mazes if x]
        # Filter out the last maze which consists of invalid signs
        policy_maze = np.array(transform(mazes[-1]))
        mazes = np.array([transform(maze) for maze in mazes[:-1]])
        return mazes, policy_maze

    def plot(data, ax=None, enum=False, title='', labels=None, legend=False,
        axes_defined = ax != None
        if not axes_defined:
            fig, ax = plt.subplots(1, 1, figsize=(13, 4))
        plotted = None
        if enum:
            plotted = ax.plot(data, **kwargs)
        else:
            mapping = np.array(data).T
            plotted = ax.plot(mapping[0], mapping[1], **kwargs)
        if labels:
            ax.set_xlabel(labels[0])
            if (len(labels) > 1):
                ax.set_ylabel(labels[1])
        if legend:
            ax.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0)
        ax.set_title(title)
        ax.grid(True)
        if not axes_defined:
            fig.tight_layout()
        return ax

    def plot_maze(ax, maze, title):
        # red = wall, green = reward, blue = unrewarded
        cmap = colors.ListedColormap(['green', 'lightblue', 'red'])
        bounds=range(4)
        norm = colors.BoundaryNorm(bounds, cmap.N)
        ax.imshow(maze, interpolation='none', cmap=cmap, norm=norm)
        ax.axis('off')
        ax.set_title(title)
```

0.0.1 H13.1 (a) Plot Mazes

```
In [532]: mazes, policy_maze = read_mazes()
          fig, axes = plt.subplots(2, 2, figsize=(10, 6))
          for i, maze in enumerate(mazes):
              plot_maze(axes.flat[i], maze, 'Maze #{}'.format(i))
          fig.tight_layout()
```



0.0.2 H13.1 (b) Transition model

State: * 0 = Reward * 1 = Unrewarded * 2 = Not accessible
Actions: * 1 = Move right * 2 = Move down * 3 = Move left * 4 = Move up

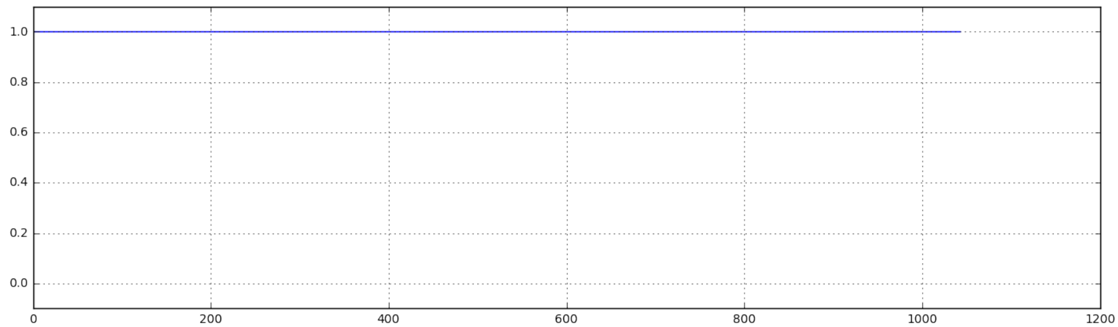
In [514]:
```
STATE_REWARDED = 0
STATE_UNREWARDED = 1
STATE_WALL = 2
MOVE_RIGHT = 0
MOVE_DOWN = 1
MOVE_LEFT = 2
MOVE_UP = 3
actions = (MOVE_RIGHT, MOVE_DOWN, MOVE_LEFT, MOVE_UP)
movements = np.array(((0, 1), (1, 0), (0, -1), (-1, 0)))

def move(maze, state, action):
    width, height = maze.shape
    nx, ny = state + movements[action]
```

```
        if nx < 0 or nx >= width or ny < 0 or ny >= height:
            raise ValueError('Invalid action {}: Destination at ({}, {}) is c
        if maze[nx, ny] == STATE_WALL:
            return state
            # raise ValueError('Invalid action {}: Destination at ({}, {}) is
        return (nx, ny)

    def transition_model(maze, state, next_state, action):
        try:
            return int(next_state == move(maze, state, action))
        except ValueError:
            return 0

    def get_states(maze):
        return list(itertools.product(range(maze.shape[0]), range(maze.shape[
```

```
In [469]: maze = mazes[0]
          states = get_states(maze)
          sums = np.zeros((len(maze.flat), len(actions)))
          values = []
          for i, state in enumerate(states):
              if maze[state[0], state[1]] == STATE_WALL:
                  sums[i] = [-1] * len(actions)
                  continue
              for k, action in enumerate(actions):
                  # Store transition model value in sums
                  for x, y in states:
                      sums[i, k] += transition_model(maze, state, (x, y), action)
                  values.append(sums[i, k])
```

```
In [470]: # plot(enum=True)
          ax = plot(sorted(values), enum=True)
          ax.set_ylim([-0.1, 1.1])

          print(plot)
```

```
<function plot at 0x00000198A52E4AE8>
```



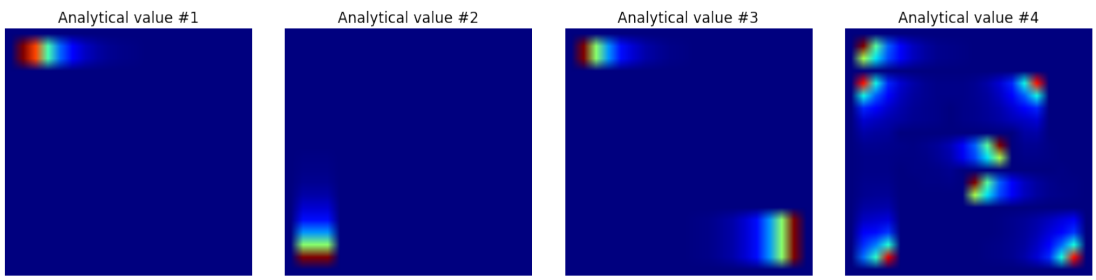
```
In [471]: possible_steps = sums.reshape(*maze.shape, len(actions)).sum(axis=2)
          fig, axes = plt.subplots(1, 2, figsize=(13, 3))
          axes[0].imshow(possible_steps)
          axes[0].axis('off')
          axes[0].set_title('Possible steps for each field')
          plot_maze(axes[1], maze, 'Original maze')
```

0.0.3 H13.1 (c) Policy evaluation

```
In [517]: def simple_policy(maze, state, action):
              return 1 / len(actions)

          def reward(maze, state, action):
              return int(maze[state[0], state[1]] == STATE_REWARDED)

          def controlled_reward(maze, state, policy_func, reward_func):
              # Bellman equation: "controlled" rewarded function
              return np.sum([policy_func(maze, state, action) * reward_func(maze, s
                             for action in actions])

          def controlled_transition(maze, state_i, state_j, policy_func, transition
              #maze, state, next_state, action
              # transition_matrix = np.zeros((len(state), len(state)))
              # for i, j in list(itertools.product(range(maze.shape[0]), range(maze
              return np.sum([policy_func(maze, state_i, action) * transition_func(m
                             for action in actions])

          def analytical_values(maze, policy_func=simple_policy, reward_func=reward
              states = get_states(maze)
              r = np.array([controlled_reward(maze, state, policy_func, reward_func
```

```
                        for state in states])
            P = np.array([[controlled_transition(maze, state_i, state_j, policy_f
                         for state_i in states]
                        for state_j in states])
            I = np.identity(len(states))
            return r, P, np.linalg.inv(I - gamma * P).dot(r)

        vs = [analytical_values(maze) for maze in mazes]
```

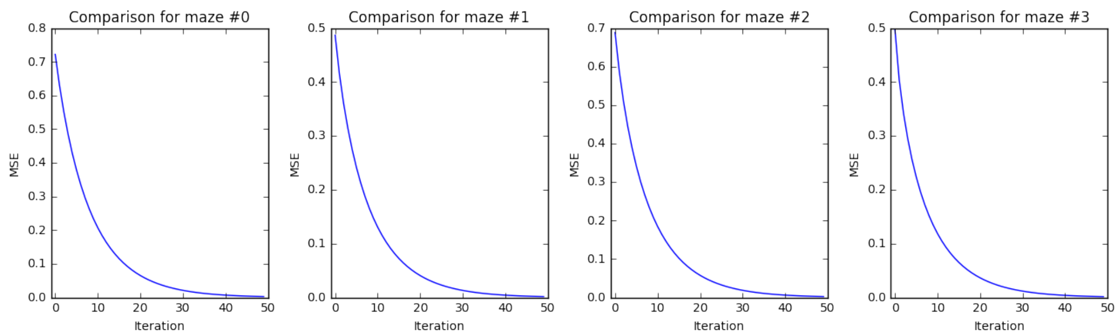
```
In [592]: fig, axes = plt.subplots(1, 4, figsize=(13, 10))
          for i, v in enumerate(vs):
              # Controlled transition model
              axes[i].imshow(v[2].reshape(20, 20))
              axes[i].axis('off')
              axes[i].set_title('Analytical value #{}'.format(i + 1))
          fig.tight_layout()
```



0.0.4 H13.1 (d) Bellman - Value iteration

```
In [515]: def value_iteration(maze, r, P, v, gamma=0.9, steps=50, nr=-1):
              approx = np.zeros((steps, len(r)))
              for i in range(steps):
                  if i == 0:
                      continue
                  approx[i] = r + gamma * P.dot(approx[i - 1])
              return approx

          def mse(v, approx):
              diff = v - approx
              return np.sqrt(np.mean(diff * diff))

          fig, axes = plt.subplots(1, 4, figsize=(13, 4))
          vs_approx = [value_iteration(maze, *vs[i]) for (i, maze) in enumerate(maz
          for i, maze in enumerate(mazes):
              r, P, v = vs[i]
              approximations = value_iteration(maze, r, P, v, nr=i)
```

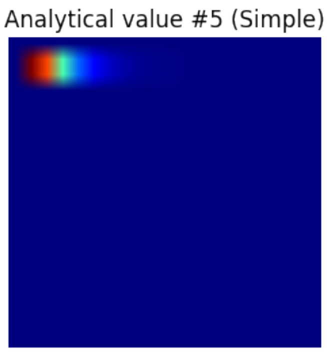
```
        mses = [mse(v, approx) for approx in approximations]
        axes[i].plot(mses)
        axes[i].set_xlabel('Iteration')
        axes[i].set_ylabel('MSE')
        axes[i].set_title('Comparison for maze #{}'.format(i))
        # axes[i].set_ylim([0, 20])
        axes[i].set_xlim([-1, 50])
    fig.tight_layout()
```

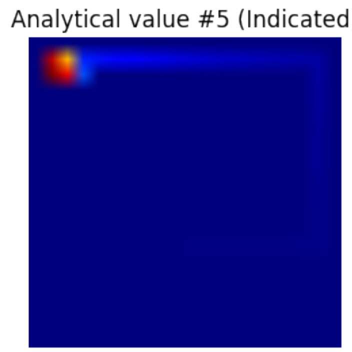


0.0.5 H13.2 (a) Indicated policy

In [541]:
```
# STATE_REWARDED = 0, STATE_UNREWARDED = 1, STATE_WALL = 2
STATE_RIGHT = 3
STATE_DOWN = 4
STATE_LEFT = 5
STATE_UP = 6

def good_policy(maze, state, action):
    state_value = maze[state[0], state[1]]
    if state_value in (STATE_UNREWARDED, STATE_REWARDED):
        return 1 / len(actions)
    if state_value in (STATE_RIGHT, STATE_DOWN, STATE_LEFT, STATE_UP):
        return int( state_value - 3 == action)
    return 0

policy_v_simple = analytical_values(policy_maze)
policy_v = analytical_values(policy_maze, policy_func=good_policy)
```

In [550]:
```
fig, axes = plt.subplots(1, 2, figsize=(13, 3))
axes[0].imshow(policy_v_simple[2].reshape(20, 20))
axes[0].axis('off')
axes[0].set_title('Analytical value #5 (Simple)')
axes[1].imshow(policy_v[2].reshape(20, 20))
axes[1].axis('off')
axes[1].set_title('Analytical value #5 (Indicated)')
fig.tight_layout()
```





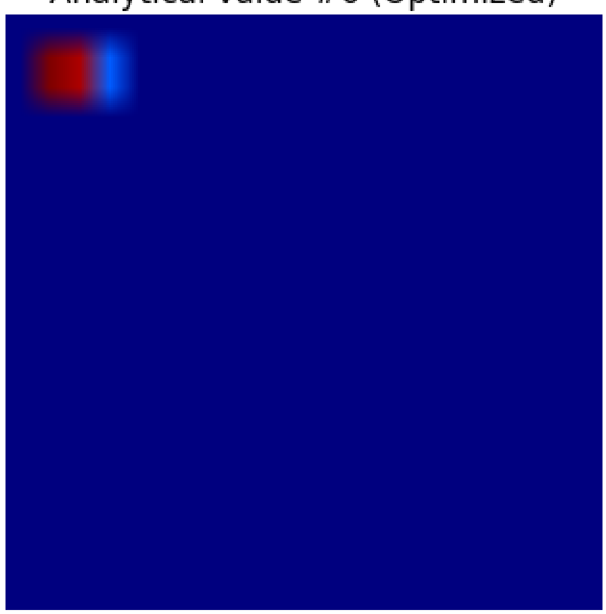
0.0.6 H13.2 (b) Find optimal policy

In [579]:
```
maze_opt_str = """
####################
#XX<<<<<<<<<<<<<<<^#
#XX<<<<<<<<<<<<<<<^#
#################^^#
#>>>>>>>>>>>>>vv#^^#
#>>>>>>>>>>>>>vv#^^#
#^^^^^^^#>>>>>vv#^^#
#^^^^^^^#>>>>>vv#^^#
#^^^##########vv#^^#
#^^^<<<<<<<<<#>>>^^#
#^^^<<<<<<<<<#>>>^^#
#^^^#^<<<########^^#
#^^^#^<<<#>>>>>>>^^#
#^^^#^<<<#>>>>>>>^^#
#^^^#^<<<###########
#^^^#^<<<<<<<<<<<<<#
#^^^#^<<<<<<<<<<<<<#
#^^^#^<<<<<<<<<<<<<#
#^^^#^<<<<<<<<<<<<<#
####################
"""

maze_opt = np.reshape(transform(replace_signs([maze_opt_str])), (20, 20))
policy_v_opt = analytical_values(maze_opt, policy_func=good_policy)
```

In [582]:
```
aximg = plt.imshow(policy_v_opt[2].reshape(20, 20))
aximg.axes.axis('off')
aximg.axes.set_title('Analytical value #6 (Optimized)')
print(maze_opt_str)
```

```
####################
#XX<<<<<<<<<<<<<<<^#
#XX<<<<<<<<<<<<<<<^#
#################^^#
#>>>>>>>>>>>>>vv#^^#
#>>>>>>>>>>>>>vv#^^#
#^^^^^^^#>>>>>vv#^^#
#^^^^^^^#>>>>>vv#^^#
#^^^##########vv#^^#
#^^^<<<<<<<<<#>>>^^#
#^^^<<<<<<<<<#>>>^^#
#^^^#^<<<########^^#
#^^^#^<<<#>>>>>>>^^#
#^^^#^<<<#>>>>>>>^^#
#^^^#^<<<###########
#^^^#^<<<<<<<<<<<<<#
#^^^#^<<<<<<<<<<<<<#
#^^^#^<<<<<<<<<<<<<#
#^^^#^<<<<<<<<<<<<<#
####################
```

Analytical value #6 (Optimized)



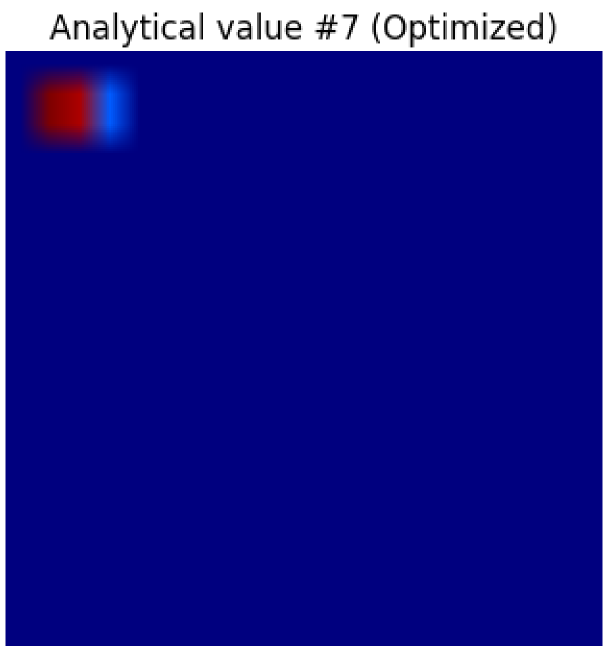
0.0.7 H13.2 (c) Another optimal policy

```
In [590]: maze_opt_str2 = """
          ####################
          #XX<<<<<<<<<<<<<<<<#
          #XX<<<<<<<<<<<<<<<<#
          #################^<#
          #vvvvvvvvvvvvvvv#^<#
          #>>>>>>>>vvvvvvv#^<#
          #>>>>>>^#vvvvvvv#^<#
          #>>>>>>^#>>>>>>v#^<#
          #>>^##########>v#^<#
          #>>^<<<<<<<<<#>>>^<#
          #>>^^^^^^^^^^#^^^^<#
          #>>^#^^^^########^<#
          #>>^#^^^^#>>>>>>>^<#
          #>>^#^^^^#^^^^^^^^<#
          #>>^#^^^^###########
          #>>^#^^^^<<<<<<<<<<#
          #>>^#^^^^^^^^^^^^^^#
          #>>^#^^^^^^^^^^^^^^#
          #>>^#^^^^^^^^^^^^^^#
          ####################
          """

          maze_opt2 = np.reshape(transform(replace_signs([maze_opt_str2])), (20, 2
          policy_v_opt2 = analytical_values(maze_opt2, policy_func=good_policy)
```

```
In [591]: aximg = plt.imshow(policy_v_opt2[2].reshape(20, 20))
          aximg.axes.axis('off')
          aximg.axes.set_title('Analytical value #7 (Optimized)')
          print(maze_opt_str2)
```

```
####################
#XX<<<<<<<<<<<<<<<<#
#XX<<<<<<<<<<<<<<<<#
#################^<#
#vvvvvvvvvvvvvvv#^<#
#>>>>>>>>vvvvvvv#^<#
#>>>>>>^#vvvvvvv#^<#
#>>>>>>^#>>>>>>v#^<#
#>>^##########>v#^<#
#>>^<<<<<<<<<#>>>^<#
#>>^^^^^^^^^^#^^^^<#
#>>^#^^^^########^<#
#>>^#^^^^#>>>>>>>^<#
#>>^#^^^^#^^^^^^^^<#
#>>^#^^^^###########
```

```
#>>^#^^^^<<<<<<<<<<#
#>>^#^^^^^^^^^^^^^^#
#>>^#^^^^^^^^^^^^^^#
#>>^#^^^^^^^^^^^^^^#
####################
```



0.0.8 H13.2 (d) Optimal Policy = Uniform Policy

```
In [ ]: # Missing
```