AA228 - Final Project Report

Spencer Diehl - bigdiehl@stanford.edu
Tyler Pharris - tpharris@stanford.edu
Alp Ozturk - azoturk@stanford.edu

December 7, 2018

Abstract

A Roomba has been powered on and found itself in a familiar room. It now needs to make its way to the next room in order to begin its task of cleaning the floors. This would normally be a simple task, but the Roomba's location and direction in the room are unknown. The Roomba knows the dimensions of the room it is in and what sensor it has. The sensor on the Roomba is either a bump sensor or a lidar. Using the sensor, an unknown initial position and direction, and the room dimensions, the goal is to reach the doorway to the next room while avoiding the flight of stairs that are also in the room. This Roomba problem can be modeled as a POMDP [1] with the location and direction of the Roomba as the state uncertainty. Two different sensors were tested on 3 different types of methods. The bump sensor was tested on a random policy, a predefined policy, an offline QMDP solver, and an online solver that used QMDP for the offline portion. The bump sensor's scores ranged from -9.596 to -5.600 points. The lidar sensor was tested on a predefined policy and an offline QMDP solver. The lidar sensor scores ranged from -3.900 to -2.188 points. Each method had a couple of failure modes with the most common being due to a lack of lookahead depth. In conclusion, when comparing results using the same sensor, online methods outperformed offline methods,which outperformed predefined policy methods, which outperformed random policy methods. Each successive method type was more effective, but required more computational power. The inclusion of the more accurate lidar sensor (which incurred no penalties for making observations, unlike the bumper sensor) increased the mean total reward for the different methods using that sensor. These results suggest that POMDP solution methods are effective methods for solving problems with partially observable state spaces. It further suggests that it may be more efficient to use POMDP solution techniques to solve this and similar problems rather than attempting to derive an explicit policy.

Problem Description

Our team was challenged with solving a problem where a Roomba was in an unknown location in a room and needed to leave the room through a doorway. The room's dimensions, including the location of the doorway and a flight of stairs, are known to the Roomba. However, its location in (x,y) coordinates and its heading θ, where the heading θ = 0° is aligned with the positive x-direction, are unknown.

Observations

The only sensor available to the Roomba is either a bump sensor, which returns a signal to the Roomba when it contacts a wall, or a lidar which return a noisy measurement of the closest wall in front of the Roomba. Using the sensor, an unknown initial position and direction, and the room dimensions the Roomba must reach the doorway to the next room.

Rewards

To quantify positive and negative behaviour, the Roomba problem is defined to have the following rewards/penalties:

- Each time that the Roomba hits a wall it receives -1 point.
  The bump sensor provides crucial informations when it hits the wall, so this incentivized our Roomba to balance impacting walls in order to acquire more information with the penalty for doing so.
- For falling down the stairs our Roomba receives a penalty of -10 points. This provides a large incentive for the Roomba to avoid the stairs.
- Reaching the goal state of the doorway to the next room gives a reward of +10 points. This provides a large incentive for our Roomba to attempt to reach the goal.
- The last scoring factor is that for every time step our Roomba receives a

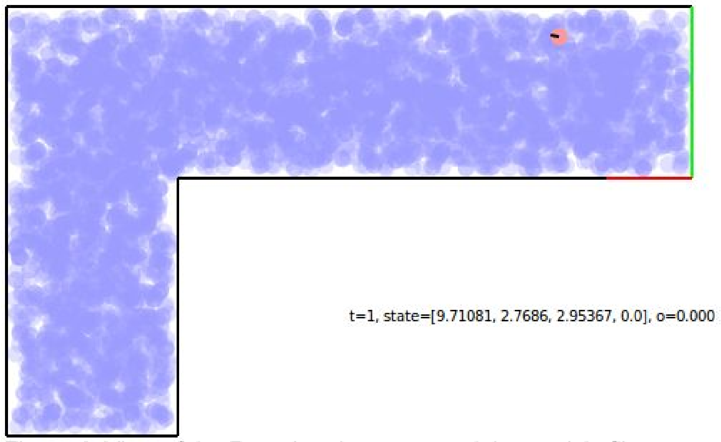

Figure 1: View of the Roomba, the room, and the particle filter state at t=0. It can be seen that the particle filter has a fairly uniform distribution across the room. The goal is indicated by the green line and the stairs by the red line. The blue dots are the particles.

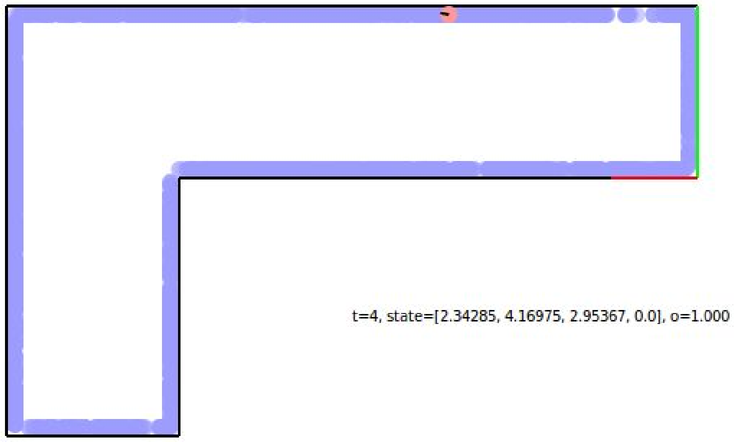

Figure 2: When the bumper sensor is used, the particle filter localizes to points along the wall when contact is made.

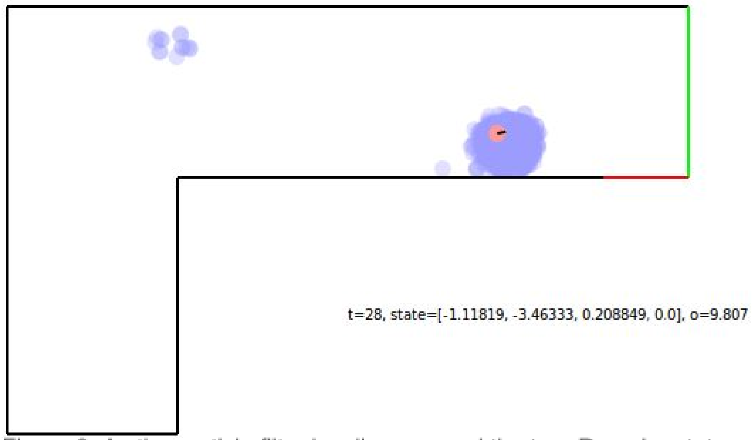

Figure 3: As the particle filter localizes around the true Roomba state (through either lidar or bumper measurements) the Roomba is better able to form an optimal policy and head towards the goal while avoiding the stairs.

-0.1 point penalty. This incentivizes our Roomba to minimize the time taken to reach the goal

Particle Filter

Our team used a particle filter in order to update our belief state from an initial uniformly sampled distribution. This particle filter was able to filter out "bad" particles using a couple of different methods.

1. If the Roomba's bump sensor is activated then all the particles must be on one of the walls. A corollary to this is that if the bump sensor is activated but the goal or stairs are not reached then all of the particles on those locations are thrown out.
2. If a particle is outside of the room's dimensions then it is thrown out of our sample.

This results in a couple of different methods for updating our belief state to a more accurate belief state using the bump sensor.

1. We can attempt to run particles out of the room's dimensions by moving farther than the state that the particle represents would allow.
2. We can eliminate particles that were not along the walls by running into a wall.

The combination of these two methods allow us to localize our belief state to where our Roomba actually is and then move to the goal state. Doing these actions has costs however.

1. The longer that it takes us to localize, the longer it takes before the goal is reached, and the lower our score will be.
2. The more walls we bounce into, the better our localization, but our score is reduced in large increments.

This means that our Roomba has to balance the decision of collecting information versus attempting to reach the goal state. Each solution method discussed in the following sections has its own calculation for balancing these tradeoffs.

Baseline Methods

Two baseline methods were tested and scored to provide a comparison for a series of four different POMDP methods.

1. The first baseline method was a random policy where the Roomba would begin by driving straight until it impacted a wall. After hitting the wall it would turn to a random direction and then drive straight until it hit another wall or an end state (goal or stairs). This would repeat until an end state was reached or 100 time steps had elapsed.
2. The second baseline method used a Lidar sensor which gave the Roomba a noisy measurement of the distance to the wall directly in front of it. This method started with the Roomba rotating for 25 time steps so that it could localize and then it pointed itself directly at the goal state and drove to it.

POMDP Solution Methods

After testing and scoring the baseline methods we used 3 different POMDP methods to attempt to beat the baseline scores.

1. The first was a simple policy that attempted to use the bump sensor to approximate the lidar sensor before pointing and driving towards the goal state.
2. The second was a QMDP method where the Roomba initially drove into a wall to localize and then used the alpha vectors calculated in QMDP and it’s current belief state to create a policy to take at each time step.
3. The third was an online variation of one-step lookahead where the utility used to calculate the policy was the sum of the alpha vectors calculated using QMDP multiplied by the current belief state and the alpha vectors multiplied by the next belief state assuming that same action. This was not the optimal one-step lookahead method but was used to see if it would improve on the QMDP method alone.

Solution Methods

This section describes in more detail each of the solution methods implemented. Each solution method uses a particle filter to update the belief state when given the belief state at the past time step, the action, and the observation made given that action occurred.

Lidar Baseline

The policy for this method was to initially have the Roomba spin in place for 25 time steps. This allowed the Roomba to localize itself using lidar measurements. After the first 25 times steps of localization the Roomba would travel towards the goal state using a simple proportional controller.

The main failure mode that this method experienced was that if the Roomba was initialized anywhere with a wall between its starting point and the goal state then it would never reach the goal state. To overcome this failure mode, a policy could be devised where if the Roomba’s bump sensor stays active for 10 time steps then it will point in a random direction and travel forward for 5 time steps before once again travelling to the goal state. Another solution was implemented in the *Lidar QMDP Method* discussed in more detail below.

Bumper Baseline

The policy for this method was to initially have the Roomba travel straight forward until it hit a wall or an end state. After impacting the wall the Roomba would then turn to a random direction and then travel straight forward until it again either hit a wall or an end state. This policy would continue until either an end state was reached or all 100 time steps ran. This method was purely random.

A policy could be devised where the Roomba will generally attempt to travel towards where it thinks the goal state is. As it impacts walls its localization will improve and its ability to travel to the goal state will get better. We implemented something similar to this below in the Localization Bumper Method. This general policy improvement won't solve every failure instance, but it would take the random chance out of the system. Many other solutions could be implemented which would improve the results of this method. Those include the *QMDP Bumper Method,* and the *Updated Belief State Lookahead Bumper Method*.

Localization Bumper Method

For this policy, the Roomba travelled straight forward until it hit a wall or an end state. After impacting the wall the Roomba would then turn to 180 degrees and then travel straight forward until it reached the initial location that it started at. It would then turn 90 degrees and travel forward until it hit a wall. It would repeat those steps 4 times which is when it arrived back at its initial state.

The intent of this procedure was to duplicate the initial localization that would occur with the Lidar, but with more uncertainty because of the noise in the velocity and angular velocity inputs to the actions.

Once the localization step was finished the Roomba would point itself in the direction of the goal state and travel forward. This policy would continue until either an end state was reached or all 100 time steps ran. This method was better than the Bumper Baseline Method but still could be improved upon. The main failure mode was that this method did not localize well, and it was guaranteed to hit at least 4 walls before it could use its localization knowledge to direct it.

Another failure mode occurred if there was a wall in between the initial location of the Roomba and the goal state, as the roomba would get stuck on said wall.. Many better solutions could be implemented which would improve the results over this method. Those include the *QMDP Bumper Method*, and the *Updated Belief State Lookahead Bumper Method*.

QMDP Bumper Method

To compute the alpha vectors necessary to create a policy from the belief state, the predefined QMDP solver included in POMDP.jl was used. Once the alpha vectors were computed offline, the simulation was started. To kickstart the simulation, the Roomba was made to travel straight forward until it hit a wall or an end state. This allowed the belief state to localize along the walls rather than being completely uniform. After impacting the wall the Roomba would then dot product the alpha vector for a given action, $\vec{\alpha}_a$,with the current belief state, $\vec{b}$,to calculate the utility of that respective action, $U_a(\vec{b})$. This would be done for all of the action states in our discretized action space and the action that had the greatest utility for that belief state would be chosen for our policy. This method is described by the equation below:

$$\pi(\vec{b}) = argmax_a(\vec{\alpha}_a^T \cdot \vec{b})$$

Equation 1: Equation for finding the policy at each belief state in the *QMDP Bumper Method* [2]

This method was proved effective but still failed on occasion. The main failure mode occurred when two clusters of particles overlapped with each other, but had opposite beliefs of what theta was. When this occurred the action with the highest utility was to just to drive in a circle with a small radius. This is because the QMDP solution was based solely on a single step horizon. It is supposed that a policy where the Roomba could do a lookahead to a certain depth to determine its highest utility would eliminate that failure mode. We implemented something similar to this below in the *Updated Belief State Lookahead Bumper Method.*

QMDP Lidar Method

This method was very similar to the *QMDP Bumper Method.* The main difference is that the lidar measurements had to be discretized in order to use the QMDP solver. Another difference is that at the beginning of the simulation the Roomba spun in place for 15 time steps while the lidar measured the distance to the nearest wall in front of the Roomba. Once the 15 time steps passed the Roomba began to choose its actions based on the method laid out in *QMDP Bumper Method* where the Roomba chose the action that would result in the greatest utility given its belief state and the alpha vectors computed offline. This method was highly effective and no major failure method occured.

Updated Belief State Lookahead Bumper Method

We first calculate the alpha vectors for our system using QMDPsolver in the POMDP.jl file . The simulation then begins with the Roomba travelling straight until it hit a wall or an end state. This localizes the particles to a more concentrated location along the walls. The Roomba then takes the dot product of the alpha vector for a given action, $\vec{\alpha}_a$, and the current belief state, $\vec{b}$,to calculate the utility of that respective action, $U_a(\vec{b})$. This would be done for all 50 of the action states in our discretized state space and then we would calculate the utility of the next step, $U_a'(\vec{b}\,|a)$,by taking the dot product of the alpha vector for a given action, $\vec{\alpha}_a$, with the next belief state given that the first action occurred, $\vec{b}\,|$ a. Our total utility was calculated by taking the sum of the dot product of the first step and the dot product of the second step and then the action that resulted in the maximum utility was chosen for our policy. [3] This method is described by the equation below:

$$\pi(\vec{b}) = argmax_a\left(\vec{\alpha}_a^T \cdot \vec{b} + max_{a'}\left[\vec{\alpha}_a^T \cdot \vec{b}|a\right]\right)$$

Equation 2: Equation for finding the policy at each belief state in the *Updated Belief State Bumper Method*

This policy would continue until either an end state was reached or all 100 time steps ran. This method was effective, but could still be improved upon by a better online method. The main failure mode again occurred when two clusters of particles overlapped with each other, but had opposite beliefs of what theta was. A policy with a greater lookahead depth can reduce the chances of the circling from happening. In this policy we only looked ahead two steps, but you could theoretically do this all the way to the 100 time steps available and find the exact actions that would maximize our utility.

Summary of Results

Below we have summarized the results obtained from testing our different methods. In each case, the simulator was run for a statistically significant number of times to approximate the mean total reward obtainable from using a particular method. As can be seen in the tables, the POMDP solution methods were able to improve on the predefined baseline methods.

Table 1: Test results for the Roomba simulation with the bumper sensor. The simulation was done for a statistically significant number of testing cycles. The results were then averaged to give the mean total reward and the standard deviation of the total reward.

Solution Method	**Mean Total Reward**	**StdErr Total Reward**	**Testing Cycles**
Bumper Baseline	-9.596	3.737	50
Localization Bumper Method	-9.452	3.196	50
QMDP Bumper Method	-6.444	3.418	50
Updated Belief State Lookahead Bumper Method	-5.600	2.950	50

Table 2: Test results for the Roomba simulation with the lidar sensor. The simulation was done for a statistically significant number of testing cycles. The results were then averaged to give the mean total reward and the standard deviation of the total reward.

Solution Method	**Mean Total Reward**	**StdErr Total Reward**	**Testing Cycles**
Lidar Baseline	-3.900	4.130	50
QMDP Lidar Method	-2.188	3.656	50

Conclusions

In conclusion, when comparing results using the same sensor, online methods such as the *Updated Belief State Lookahead Bumper Method* outperformed offline methods, such as the *QMDP Bumper Method*, which outperformed predefined policy methods, such as the *Localization Bumper Metho*d, which outperformed random policy methods, such as the *Bumper Baseline Method*. Each successive method type was more effective, but required more computational power.

The inclusion of the more accurate lidar sensor (which incurred no penalties for making observations, unlike the bumper sensor) increased the mean total reward for the different methods using that sensor.

These results suggest that POMDP solution methods are effective methods for solving problems with partially observable state spaces. It further suggests that it may be more efficient to use POMDP solution techniques to solve this and similar problems rather than attempting to derive an explicit policy.

References

[1] Kochenderfer, M. J., 2015, "Decision Making Under Uncertainty: Theory and Application," State Uncertainty. M. J. Kochenderfer, The MIT Press, Cambridge, Massachusetts, pp. 133.

[2] Kochenderfer, M. J., 2015, "Decision Making Under Uncertainty: Theory and Application," State Uncertainty. M. J. Kochenderfer, The MIT Press, Cambridge, Massachusetts, pp. 144.

[3] Kochenderfer, M. J., 2015, "Decision Making Under Uncertainty: Theory and Application," State Uncertainty. M. J. Kochenderfer, The MIT Press, Cambridge, Massachusetts, pp. 149.

Appendix

Here we list the code files used in the development of this paper. The files are as follows:

Solution methods file listing: The first listing contains the full file code. Subsequent listings exclude the simulation and testing routines and the import statements:

Support file listing: All the solution methods above used the same support files, as listed below:

1. roomba_env.jl - Contains definitions for the Roomba environment and for the Roombda POMDP
2. filtering.jl - Contains definitions for the particle filter used during simulations
3. env_room.jl - Contains definitions and functions for creating the room environment
4. line_segment_utils.jl - Contains functions for determining whether the Roomba's path intersects with a line segment.

Lidar_Baseline.jl

```
# activate project environment
# include these lines of code in any future scripts/notebooks
import Pkg

if !haskey(Pkg.installed(), "AA228FinalProject")
      jenv = joinpath(dirname(@__FILE__()), ".") # this assumes the
notebook is in the same dir
      # as the Project.toml file, which should be in top level dir of the
project.
      # Change accordingly if this is not the case.
      Pkg.activate(jenv)
end

# import necessary packages
using AA228FinalProject
using POMDPs
using POMDPPolicies
using BeliefUpdaters
using ParticleFilters
using POMDPSimulators
using Cairo
using Gtk
using Random
using Printf

# %%
-----------------------------------------------------------------------
sensor = Lidar() # or Bumper() for the bumper version of the environment
config = 3 # 1,2, or 3
m = RoombaPOMDP(sensor=sensor, mdp=RoombaMDP(config=config));

# %%
-----------------------------------------------------------------------
num_particles = 2000
resampler = LidarResampler(num_particles,
LowVarianceResampler(num_particles))
# for the bumper environment
# resampler = BumperResampler(num_particles)

spf = SimpleParticleFilter(m, resampler)
```

```
v_noise_coefficient = 2.0
om_noise_coefficient = 0.5

belief_updater = RoombaParticleFilter(spf, v_noise_coefficient,
om_noise_coefficient);

# %%
----------------------------------------------------------------------
# Define the policy to test
mutable struct ToEnd <: Policy
      ts::Int64 # to track the current time-step.
end

# extract goal for heuristic controller
goal_xy = get_goal_xy(m)

# define a new function that takes in the policy struct and current belief,
# and returns the desired action
function POMDPs.action(p::ToEnd, b::ParticleCollection{RoombaState})

      # spin around to localize for the first 25 time-steps
      if p.ts < 25
      p.ts += 1
      return RoombaAct(0.,1.0) # all actions are of type RoombaAct(v,om)
      end
      p.ts += 1

      # after 25 time-steps, we follow a proportional controller to
navigate
      # directly to the goal, using the mean belief state

      # compute mean belief of a subset of particles
      s = mean(b)

      # compute the difference between our current heading and one that
would
      # point to the goal
      goal_x, goal_y = goal_xy
      x,y,th = s[1:3]
      ang_to_goal = atan(goal_y - y, goal_x - x)
      del_angle = wrap_to_pi(ang_to_goal - th)
      # apply proportional control to compute the turn-rate
      Kprop = 1.0
      om = Kprop * del_angle

      # always travel at some fixed velocity
      v = 5.0

      return RoombaAct(v, om)
end

# %%
-----------------------------------------------------------------------
# first seed the environment
Random.seed!(2)

# reset the policy
p = ToEnd(0) # here, the argument sets the time-steps elapsed to 0

#RUN THE SIMULATION
c = @GtkCanvas()
win = GtkWindow(c, "Roomba Environment", 600, 600)
for (t, step) in enumerate(stepthrough(m, p, belief_updater,
max_steps=100))
      @guarded draw(c) do widget

      # the following lines render the room, the particles, and the roomba
      ctx = getgc(c)
      set_source_rgb(ctx,1,1,1)
      paint(ctx)
      render(ctx, m, step)

      # render some information that can help with debugging
      # here, we render the time-step, the state, and the observation
      move_to(ctx,300,400)
      show_text(ctx, @sprintf("t=%d, state=%s,
o=%.3f",t,string(step.s),step.o))
      end
      show(c)
      sleep(0.1) # to slow down the simulation
end

# %%
```

```
-----------------------------------------------------------------------
using Statistics

total_rewards = []

for exp = 1:50
      println(string(exp))

      Random.seed!(exp)

      p = ToEnd(0)
      traj_rewards = sum([step.r for step in
stepthrough(m,p,belief_updater, max_steps=100)])

      push!(total_rewards, traj_rewards)
end

@printf("Mean Total Reward: %.3f, StdErr Total Reward: %.3f",
mean(total_rewards), std(total_rewards)/sqrt(5))
```

Bumper_Baseline.jl

```
# %%
----------------------------------------------------------------------
sensor = Bumper() # or Bumper() for the bumper version of the environment
config = 3 # 1,2, or 3

m = RoombaPOMDP(sensor=sensor, mdp=RoombaMDP(config=config));
# %%
----------------------------------------------------------------------
num_particles = 5000
resampler = BumperResampler(num_particles)
# for the bumper environments
# resampler = BumperResampler(num_particles)

spf = SimpleParticleFilter(m, resampler)

v_noise_coefficient = 2.0
om_noise_coefficient = 0.5

belief_updater = RoombaParticleFilter(spf, v_noise_coefficient,
om_noise_coefficient);

# %%
----------------------------------------------------------------------
# Define the policy to test
mutable struct ToEnd <: Policy
      ts::Int64 # to track the current time-step.
end

# extract goal for heuristic controller
goal_xy = get_goal_xy(m)

#flag for our wall hit policy
previousBumpState = false
spinStep = 1

# define a new function that takes in the policy struct and current belief,
# and returns the desired action
function POMDPs.action(p::ToEnd, b::ParticleCollection{RoombaState})

      #Seed the environment
      Random.seed!()
      #Naive approach: Bump the wall, spin in a random direction, and then
drive
      #again.
      global previousBumpState
      global spinSteps

      #Fixed Velocity and spin rate maximums
      velMax = m.mdp.v_max
      omegaMax = m.mdp.om_max

      #Normal driving speed
      vel = 5.0

      #Set max and min number of time-steps to spin
      maxSpinCount = 8
      minSpinCount = 2

      #Increase time step
      p.ts += 1

      #If the wall has been bumped, then all particles are on the wall. If
so,
      #then any particle will do for determining wall contact
      s = particle(b,1)

      #Call the wall_contact function to determine if we are in wall
contact
      #(returns true or false)
      currentBumpState = AA228FinalProject.wall_contact(m,s)

      #The bump sensor tells us we are in contact with the wall
      if (currentBumpState == true)
      #Our memory variable tells us we weren't in contact the previous
timestep
      if (previousBumpState == false)
            #Set a random number of time steps to spin
            spinSteps =
floor(minSpinCount+rand()*(maxSpinCount-minSpinCount))
            #Update the previous state
            previousBumpState = currentBumpState
            #Return our trajectory
            return RoombaAct(0.0, omegaMax)
      #Our memory variable tells us we were in contact with the wall in the
      #last time-step as well
      elseif (previousBumpState == true)
            #If we are still spinning
            if (spinSteps != 0)
                  #decrement spinSteps
                  spinSteps -= 1

                  #Return our trajectory
                  return RoombaAct(0.0, omegaMax)
            #Otherwise we are ready to test to see if we can drive forward
            else
                  #Assume that we won't aren't pointed into a wall
                  previousBumpState = false
                  #Attempt to drive forward. If we can't them the logic
flow will
                  #reset the spinSteps variable and the Roomba will spin a
random
                  #number of time-steps again
                  return RoombaAct(vel, 0.0)
            end
      end

      #If we aren't in contact with the wall
      elseif (currentBumpState == false)
      #Update previous state
      previousBumpState = currentBumpState
      #Drive forward at a fixed velocity
      return RoombaAct(vel, 0.0)
      end
end
```

Lidar_QMDP.jl

```julia
# %%
----------------------------------------------------------------------
#Discretize the observation space
cut_points = collect(0:1:30)
sensor_discrete = DiscreteLidar(cut_points)

#Discretize state space
num_x_pts = 20
num_y_pts = 20
num_th_pts = 10
sspace = DiscreteRoombaStateSpace(num_x_pts,num_y_pts,num_th_pts)

#Discretize action space
vlist = collect(0:2.5:10.0)
omlist = collect(0:0.25:1.0)
aspace = vec(collect(RoombaAct(v, om) for v in vlist, om in omlist))

#Define our sensor and room layout
sensor = Lidar()
config = 1 # 1,2, or 3

#Define POMDP for the simulation
m = RoombaPOMDP(sensor=sensor, mdp=RoombaMDP(config=config));

#Define the POMDP with discretized state/action/observations for the solver
m_discrete = RoombaPOMDP(sensor=sensor_discrete,
mdp=RoombaMDP(config=config, sspace = sspace,
                        aspace = aspace));

# %%
----------------------------------------------------------------------
#Create our particle filter
num_particles = 5000
resampler = LidarResampler(num_particles,
LowVarianceResampler(num_particles))

spf = SimpleParticleFilter(m, resampler)

v_noise_coefficient = 2.0
om_noise_coefficient = 0.5
```

```
belief_updater = RoombaParticleFilter(spf, v_noise_coefficient,
om_noise_coefficient);

# %%
----------------------------------------------------------------------
#Define our solver
solver = QMDPSolver(max_iterations=20,
                  tolerance=1e-3,
                  verbose=true)

#If we need to compute our policy for the first time
if (1 == 1)
      #Use our solver and our POMDP model to find a policy
      policy = solve(solver,m_discrete)
      #Save our policy so we don't have to recompute
      using JLD2, FileIO
      @JLD2.save "my_policy_lidar.jld" policy

#Otherwise use the saved policy we computed previously
else
      @JLD2.load "my_policy_lidar.jld" policy
end

# %%
----------------------------------------------------------------------
# Define the policy to test
mutable struct ToEnd <: Policy
      ts::Int64 # to track the current time-step.
end

# %%
----------------------------------------------------------------------

states = POMDPs.states(m_discrete)

# define a new function that takes in the policy struct and current belief,
# and returns the desired action
function POMDPs.action(p::ToEnd, b::ParticleCollection{RoombaState})

      # spin around to localize for the first 25 time-steps
      if p.ts < 15
      p.ts += 1
      return RoombaAct(0.,1.0) # all actions are of type RoombaAct(v,om)
      end
      p.ts += 1

      #Get our policy from our solver
      global policy

      #Extract our set of alpha vectors from our policy (one for each
action)
      alphas = policy.alphas

      greatestUtilityIndex = 0
      greatestUtility = -Inf

      numStates = POMDPs.n_states(m_discrete)
      numActions = length(alphas)

      #Create our belief vector from our particle filter
      belief = zeros(numStates)
      for i = 1:num_particles
      s = particle(b,i)
      index = POMDPs.stateindex(m_discrete,s)
      belief[index] += 1
      end
      belief = belief/num_particles

      # see which action gives the highest util value
      for i = 1:numActions
      utility = dot(alphas[i], belief)
      if utility > greatestUtility
            greatestUtility = utility
            greatestUtilityIndex = i
      end
      end

      # map the index to action
      a = policy.action_map[greatestUtilityIndex]

      if greatestUtilityIndex == 1
      a = policy.action_map[2]
      end
```

```julia
      return RoombaAct(a[1], a[2])
end
```

Bumper_QMDP.jl

```
# %%
----------------------------------------------------------------------
#Define our sensor
sensor = Bumper()

#Room configuration. Choose from 1,2, or 3
config = 1

#Discretize state space
num_x_pts = 50
num_y_pts = 50
num_th_pts = 20
sspace = DiscreteRoombaStateSpace(num_x_pts,num_y_pts,num_th_pts)

#Discretize action space
vlist = collect(0:1.0:10.0)
omlist = collect(0:0.2:1.0)
aspace = vec(collect(RoombaAct(v, om) for v in vlist, om in omlist))

#Get rid of first action which is (0,0). Doesn't do us any good
#aspace = aspace[2:length(aspace)]

#Construct the POMPDP for the QMDP solver (Discrete state/action space)
m_discrete = RoombaPOMDP(sensor=sensor, mdp=RoombaMDP(config=config, sspace
= sspace,
                        aspace = aspace));

#Try increasing the rewards
m_discrete.mdp.goal_reward = 100
m_discrete.mdp.stairs_penalty = -100


#Construct the POMDP for the simulator (Continuous state/action space)
m = RoombaPOMDP(sensor=sensor, mdp=RoombaMDP(config=config));

# %%
----------------------------------------------------------------------
#Create the particle filter
num_particles = 10000 #2000
```

```
resampler = BumperResampler(num_particles)
spf = SimpleParticleFilter(m, resampler)

#Create our belief updater using our simple particle filter
v_noise_coefficient = 2.0
om_noise_coefficient = 0.5
belief_updater = RoombaParticleFilter(spf, v_noise_coefficient,
om_noise_coefficient);

# %%
----------------------------------------------------------------------
#Define our solver
#solver = FIBSolver()
solver = QMDPSolver(max_iterations=20,
                  tolerance=1e-3,
                  verbose=true)


#If we need to compute our policy for the first time
if (1 == 0)
      #Use our solver and our POMDP model to find a policy
      policy = solve(solver,m_discrete)
      #Save our policy so we don't have to recompute
      using JLD2, FileIO
      @JLD2.save "my_policy3.jld" policy

#Otherwise use the saved policy we computed previously
else
      @JLD2.load "my_policy2.jld" policy
end

# %%
----------------------------------------------------------------------

# Define the policy to test
mutable struct ToEnd <: Policy
      ts::Int64 # to track the current time-step.
      policy::AlphaVectorPolicy
end

# %%
----------------------------------------------------------------------
```

```
#flag for our wall hit policy
previousBumpState = false

states = POMDPs.states(m_discrete)

# define a new function that takes in the policy struct and current belief,
# and returns the desired action
function POMDPs.action(p::ToEnd, b::ParticleCollection{RoombaState})

      global previousBumpState

      #Drive straight into wall to localize belief state
      if previousBumpState == false
      #If the wall has been bumped, then all particles are on the wall. If
so,
      #then any particle will do for determining wall contact
      s = particle(b,1)

      #Call the wall_contact function to determine if we are in wall
contact
      #(returns true or false)
      currentBumpState = AA228FinalProject.wall_contact(m,s)

      if currentBumpState == false
            return RoombaAct(5.0, 0.0)
      else
            previousBumpState = true
      end
      end

      #Use alpha vectors once first wall contact is made
      if previousBumpState == true
      #Extract our policy from struct p
      policy = p.policy
      #Extract our set of alpha vectors from our policy (one for each
action)
      alphas = policy.alphas

      greatestUtilityIndex = 6
      greatestUtility = -Inf

      numStates = POMDPs.n_states(m_discrete)
      numActions = length(alphas)


      #Create our belief vector from our particle filter
      belief = zeros(numStates)
      for i = 1:num_particles
            s = particle(b,i)
            index = POMDPs.stateindex(m_discrete,s)
            belief[index] += 1
      end
      belief = belief/num_particles

      #print(belief)
      #print("\n")

      # see which action gives the highest util value
      for i = 1:numActions
            utility = dot(alphas[i], belief)
            if utility > greatestUtility
                  greatestUtility = utility
                  greatestUtilityIndex = i

            end
      end

      # map the index to action
      a = policy.action_map[greatestUtilityIndex]

      if greatestUtilityIndex == 1
            a = policy.action_map[6]
      end

      return RoombaAct(a[1], a[2])
      end

end
```

Localization_Bumper.jl

```julia
# %%
----------------------------------------------------------------------
sensor = Bumper() # or Bumper() for the bumper version of the environment
config = 1 # 1,2, or 3
m = RoombaPOMDP(sensor=sensor, mdp=RoombaMDP(config=config));

# %%
----------------------------------------------------------------------
num_particles = 10000
resampler = BumperResampler(num_particles)
# for the bumper environments
# resampler = BumperResampler(num_particles)

spf = SimpleParticleFilter(m, resampler)

v_noise_coefficient = 2.0
om_noise_coefficient = 0.5

belief_updater = RoombaParticleFilter(spf, v_noise_coefficient,
om_noise_coefficient);

# %%
----------------------------------------------------------------------
# Define the policy to test
mutable struct ToEnd <: Policy
    ts::Int64 # to track the current time-step.
end

# extract goal for heuristic controller
goal_xy = get_goal_xy(m)

#flag for our wall hit policy
previousBumpState = false
spinStep =

# define a new function that takes in the policy struct and current belief,
# and returns the desired action
function POMDPs.action(p::ToEnd, b::ParticleCollection{RoombaState})

    #Seed the environment
    Random.seed!()

    #Naive approach: Bump the wall, spin in a random direction, and then
drive
    #again.
    global velSteps
    global previousBumpState
    global spinSteps
    global L

    #Fixed Velocity and spin rate maximums
    velMax = m.mdp.v_max
    omegaMax = m.mdp.om_max

    #Normal driving speed
    vel = 2

    #Set max and min number of time-steps to spin
    maxSpinCount = 7
    minSpinCount = 3

    #Increase time step
    p.ts += 1

    #If the wall has been bumped, then all particles are on the wall. If
so,
    #then any particle will do for determining wall contact
    s = particle(b,1)

    #Call the wall_contact function to determine if we are in wall contact
    #(returns true or false)
    currentBumpState = AA228FinalProject.wall_contact(m,s)
    initialBumpState = AA228FinalProject.wall_contact(m,s)
    OmegaNinetyDegrees=0.7853981634
    if (p.ts<2 && currentBumpState==true)
        L=1
        velSteps=0
    end
    if (p.ts<2 && currentBumpState==false)
        L=2
        velSteps=0
    end
    if (p.ts>400)
        L=3
    end
    if (p.ts>420)
        L=5
    end
    if (L==1)
        if (currentBumpState==true)
            if (previousBumpState==false)
                spinSteps=4
                previousBumpState=currentBumpState
            elseif (previousBumpState=true)
                if (spinSteps!=0)
                    spinSteps-=1
                    p.ts+=1
                    return RoombaAct(0.0,OmegaNinetyDegrees)
                else
                    previousBumpState=false
                    p.ts+=1
                    L=2
                    return RoombaAct(vel,0.0)
                end
            end
        end
    elseif (L==2)
        while (currentBumpState==false)
            velSteps+=1
            previousBumpState=currentBumpState
            p.ts+=1
            return RoombaAct(vel,0.0)
        end
        while (currentBumpState==true)
            while (previousBumpState==false)
                spinSteps=7
                previousBumpState=currentBumpState
                p.ts+=1
                return RoombaAct(0.0,OmegaNinetyDegrees)
            end
            while (previousBumpState==true)
                if (spinSteps!=0)
                    spinSteps-=1
                    p.ts+=1
                    return RoombaAct(0.0,OmegaNinetyDegrees)
                else
                    velSteps-=1
                    p.ts+=1
                    L=4
                    return RoombaAct(vel,0.0)
                end
            end
        end
    elseif (L==4)
        if (velSteps>0)
            velSteps-=1
            p.ts+=1
            spinSteps=4
            return RoombaAct(vel,0.0)
        else
            if (spinSteps>0)
                spinSteps-=1
                p.ts+=1
                return RoombaAct(0.0,OmegaNinetyDegrees)
            else
                L=2
                velSteps+=1
                p.ts+=1
                return RoombaAct(vel,0.0)
            end
        end
    elseif (L==3)
        s = mean(b)
        goal_x, goal_y = goal_xy
        x,y,th = s[1:3]
        ang_to_goal = atan(goal_y - y, goal_x - x)
        del_angle = wrap_to_pi(ang_to_goal - th)
        Kprop = 1.0
        om = Kprop * del_angle
        v = 5.0
        p.ts+=1
        return RoombaAct(0.0, om)
    elseif (L==5)
        s = mean(b)
        goal_x, goal_y = goal_xy
        x,y,th = s[1:3]
```

```
        ang_to_goal = atan(goal_y - y, goal_x - x)
        del_angle = wrap_to_pi(ang_to_goal - th)
        Kprop = 1.0
        om = Kprop * del_angle
        v = 5.0
        p.ts+=1
        return RoombaAct(v,om)
    end
End
```

Updated_Belief_State_Bumper.jl

```
# %%
----------------------------------------------------------------------
#Define our sensor
sensor = Bumper()

#Room configuration. Choose from 1,2, or 3
config = 1

#Discretize state space

num_x_pts = 50
num_y_pts = 50
num_th_pts = 20
sspace = DiscreteRoombaStateSpace(num_x_pts,num_y_pts,num_th_pts)

#Discretize action space
vlist = collect(0:1.0:10.0)
omlist = collect(0:0.2:1.0)
aspace = vec(collect(RoombaAct(v, om) for v in vlist, om in omlist))

#Get rid of first action which is (0,0). Doesn't do us any good
#aspace = aspace(2:length(aspace))

#Construct the POMPDP for the QMDP solver (Discrete state/action space)
m_discrete = RoombaPOMDP(sensor=sensor, mdp=RoombaMDP(config=config, sspace
= sspace,
                      aspace = aspace));

#Try increasing the rewards
m_discrete.mdp.goal_reward = 10
m_discrete.mdp.stairs_penalty = -10


#Construct the POMDP for the simulator (Continuous state/action space)
m = RoombaPOMDP(sensor=sensor, mdp=RoombaMDP(config=config));

# %%
----------------------------------------------------------------------
#Create the particle filter
```

```julia
num_particles = 5000 #2000
resampler = BumperResampler(num_particles)
spf = SimpleParticleFilter(m, resampler)

#Create our belief updater using our simple particle filter
v_noise_coefficient = 2.0
om_noise_coefficient = 0.5
belief_updater = RoombaParticleFilter(spf, v_noise_coefficient,
om_noise_coefficient);

# %%
-----------------------------------------------------------------------
#Define our solver
#solver = FIBSolver()
solver = QMDPSolver(max_iterations=20,
                    tolerance=1e-3,
                    verbose=true)


#If we need to compute our policy for the first time
if (1 == 0)
    #Use our solver and our POMDP model to find a policy
    policy = solve(solver,m_discrete)
    #Save our policy so we don't have to recompute
    using JLD2, FileIO
    @JLD2.save "my_policy.jld" policy


#Otherwise use the saved policy we computed previously
else
    @JLD2.load "my_policy2.jld" policy
end

# %%
-----------------------------------------------------------------------

# Define the policy to test
mutable struct ToEnd <: Policy
    ts::Int64 # to track the current time-step.
    policy::AlphaVectorPolicy
end
```

```
# %%
----------------------------------------------------------------------
#flag for our wall hit policy
previousBumpState = false

states = POMDPs.states(m_discrete)

# define a new function that takes in the policy struct and current belief,
# and returns the desired action
function POMDPs.action(p::ToEnd, b::ParticleCollection{RoombaState})

    global previousBumpState

    #Drive straight into wall to localize belief state
    if previousBumpState == false
        #If the wall has been bumped, then all particles are on the wall.
If so,
        #then any particle will do for determining wall contact
        s = particle(b,1)

        #Call the wall_contact function to determine if we are in wall
contact
        #(returns true or false)
        currentBumpState = AA228FinalProject.wall_contact(m,s)

        if currentBumpState == false
            return RoombaAct(5.0, 0.0)
        else
            previousBumpState = true
        end
    end

    #Use alpha vectors once first wall contact is made
    if previousBumpState == true
        #Extract our policy from struct p
        policy = p.policy
        #Extract our set of alpha vectors from our policy (one for each
action)
        alphas = policy.alphas

        greatestUtilityIndex = 6
        greatestUtility = -Inf
        numStates = POMDPs.n_states(m_discrete)
        numActions = length(alphas)


        #Create our belief vector from our particle filter
        belief = zeros(numStates)
        for i = 1:num_particles
            s = particle(b,i)
            index = POMDPs.stateindex(m_discrete,s)
            belief[index] += 1
        end
        belief = belief/num_particles

        #print(belief)
        #print("\n")
        global s1
        # see which action gives the highest util value
        for i = 1:numActions
            bp1=POMDPs.update(belief_updater,
b,RoombaAct(policy.action_map[i]),AA228FinalProject.wall_contact(m_discrete
,s))
            for j = 1:num_particles
                s1 = particle(bp1,j)
            end
            bp2=POMDPs.update(belief_updater,
b,RoombaAct(policy.action_map[i]),AA228FinalProject.wall_contact(m_discrete
,s1))
            beliefnew2 = zeros(numStates)
            for j = 1:num_particles
                s2 = particle(bp2,j)
                index2 = POMDPs.stateindex(m_discrete,s2)
                beliefnew2[index2] += 1
            end
            beliefnew2 = beliefnew2/num_particles
            utility = dot(alphas[i], belief)+dot(alphas[i], beliefnew2)
            if utility > greatestUtility
                greatestUtility = utility
                greatestUtilityIndex = i
            end
        end
        # map the index to action
        a = policy.action_map[greatestUtilityIndex]

        if greatestUtilityIndex == 1
            a = policy.action_map[6]
        end
        return RoombaAct(a[1], a[2])
    end

end
```

Two_Step_Lookahead.jl

```
# %%
----------------------------------------------------------------------
#Define our sensor
sensor = Bumper()

#Room configuration. Choose from 1,2, or 3
config = 1

#Discretize state space

num_x_pts = 50
num_y_pts = 50
num_th_pts = 20
sspace = DiscreteRoombaStateSpace(num_x_pts,num_y_pts,num_th_pts)

#Discretize action space
vlist = collect(0:1.0:10.0)
omlist = collect(0:0.2:1.0)
aspace = vec(collect(RoombaAct(v, om) for v in vlist, om in omlist))

#Get rid of first action which is (0,0). Doesn't do us any good
#aspace = aspace(2:length(aspace))

#Construct the POMPDP for the QMDP solver (Discrete state/action space)
m_discrete = RoombaPOMDP(sensor=sensor, mdp=RoombaMDP(config=config, sspace
= sspace,
                      aspace = aspace));

#Try increasing the rewards
m_discrete.mdp.goal_reward = 10
m_discrete.mdp.stairs_penalty = -10


#Construct the POMDP for the simulator (Continuous state/action space)
m = RoombaPOMDP(sensor=sensor, mdp=RoombaMDP(config=config));

# %%
----------------------------------------------------------------------
#Create the particle filter
```

```
num_particles = 40000 #2000
resampler = BumperResampler(num_particles)
spf = SimpleParticleFilter(m, resampler)

#Create our belief updater using our simple particle filter
v_noise_coefficient = 2.0
om_noise_coefficient = 0.5
belief_updater = RoombaParticleFilter(spf, v_noise_coefficient,
om_noise_coefficient);

# %%
----------------------------------------------------------------------
#Define our solver
#solver = FIBSolver()
solver = QMDPSolver(max_iterations=20,
                    tolerance=1e-3,
                    verbose=true)


#If we need to compute our policy for the first time
if (1 == 0)
    #Use our solver and our POMDP model to find a policy
    policy = solve(solver,m_discrete)
    #Save our policy so we don't have to recompute
    using JLD2, FileIO
    @JLD2.save "my_policy.jld" policy


#Otherwise use the saved policy we computed previously
else
    @JLD2.load "my_policy2.jld" policy
end

# %%
----------------------------------------------------------------------

# Define the policy to test
mutable struct ToEnd <: Policy
    ts::Int64 # to track the current time-step.
    policy::AlphaVectorPolicy
end
```

```
# %%
----------------------------------------------------------------------
#flag for our wall hit policy
previousBumpState = false

states = POMDPs.states(m_discrete)

# define a new function that takes in the policy struct and current belief,
# and returns the desired action
function POMDPs.action(p::ToEnd, b::ParticleCollection{RoombaState})

    global previousBumpState

    #Drive straight into wall to localize belief state
    if previousBumpState == false
        #If the wall has been bumped, then all particles are on the wall.
If so,
        #then any particle will do for determining wall contact
        s = particle(b,1)

        #Call the wall_contact function to determine if we are in wall
contact
        #(returns true or false)
        currentBumpState = AA228FinalProject.wall_contact(m,s)

        if currentBumpState == false
            return RoombaAct(5.0, 0.0)
        else
            previousBumpState = true
        end
    end

    #Use alpha vectors once first wall contact is made
    if previousBumpState == true
        #Extract our policy from struct p
        policy = p.policy
        #Extract our set of alpha vectors from our policy (one for each
action)
        alphas = policy.alphas

        greatestUtilityIndex = 6
        greatestUtility = -Inf
        greatestUtilityIndex2 = 6
        greatestUtility2 = -Inf

        numStates = POMDPs.n_states(m_discrete)
        numActions = length(alphas)


        #Create our belief vector from our particle filter
        belief = zeros(numStates)
        for i = 1:num_particles
            s = particle(b,i)
            index = POMDPs.stateindex(m_discrete,s)
            belief[index] += 1
        end
        belief = belief/num_particles

        #print(belief)
        #print("\n")
        global s1
        # see which action gives the highest util value
        for i = 1:numActions
            bp1=POMDPs.update(belief_updater,
b,RoombaAct(policy.action_map[i]),AA228FinalProject.wall_contact(m_discrete
,s))
            for j = 1:num_particles
                s1 = particle(bp1,j)
            end
            bp2=POMDPs.update(belief_updater,
b,RoombaAct(policy.action_map[i]),AA228FinalProject.wall_contact(m_discrete
,s1))
            beliefnew2 = zeros(numStates)
            greatestUtility2=zeros(numActions)
            for j = 1:num_particles
                s2 = particle(bp2,j)
                index2 = POMDPs.stateindex(m_discrete,s2)
                beliefnew2[index2] += 1
            end
            beliefnew2 = beliefnew2/num_particles
            utility2 =dot(alphas[i], beliefnew2)
            greatestUtility2[i] = utility2
            greatestUtilityIndex2 = i
        end
        for i = 1:numActions
            utility = dot(alphas[i], belief)+greatestUtility2[i]
            if utility > greatestUtility
                greatestUtility = utility
                greatestUtilityIndex = i
            end
        end

        # map the index to action
        a = policy.action_map[greatestUtilityIndex]

        if greatestUtilityIndex == 1
            a = policy.action_map[6]
        end
        return RoombaAct(a[1], a[2])
    end

end
```

roomba_env.jl

```julia
# Defines the environment as a POMDPs.jl MDP and POMDP
# maintained by {jmorton2,kmenda}@stanford.edu

# Wraps ang to be in (-pi, pi]
function wrap_to_pi(ang::Float64)
      if ang > pi
       ang -= 2*pi
    elseif ang <= -pi
       ang += 2*pi
      end
    ang
end

"""
State of a Roomba.

# Fields
- `x::Float64` x location in meters
- `y::Float64` y location in meters
- `theta::Float64` orientation in radians
- `status::Bool` indicator whether robot has reached goal state or stairs
"""
struct RoombaState <: FieldVector{4, Float64}
      x::Float64
      y::Float64
      theta::Float64
      status::Float64
end

# Struct for a Roomba action
struct RoombaAct <: FieldVector{2, Float64}
      v::Float64  # meters per second
      omega::Float64 # theta dot (rad/s)
end

# action spaces
struct RoombaActions end

function gen_amap(aspace::RoombaActions)
```

```
    return nothing
end

function gen_amap(aspace::AbstractVector{RoombaAct})
    return Dict(aspace[i]=>i for i in 1:length(aspace))
end

"""
Define the Roomba MDP.

# Fields
- `v_max::Float64` maximum velocity of Roomba [m/s]
- `om_max::Float64` maximum turn-rate of Roombda [rad/s]
- `dt::Float64` simulation time-step [s]
- `contact_pen::Float64` penalty for wall-contact
- `time_pen::Float64` penalty per time-step
- `goal_reward::Float64` reward for reaching goal
- `stairs_penalty::Float64` penalty for reaching stairs
- `config::Int` specifies room configuration (location of stairs/goal)
{1,2,3}
- `room::Room` environment room struct
- `sspace::SS` environment state-space (ContinuousRoombaStateSpace or
DiscreteRoombaStateSpace)
- `aspace::AS` environment action-space struct
"""
@with_kw mutable struct RoombaMDP{SS,AS} <: MDP{RoombaState, RoombaAct}
    v_max::Float64  = 10.0  # m/s
    om_max::Float64 = 1.0   # rad/s
    dt::Float64       = 0.5   # s
    contact_pen::Float64 = -1.0
    time_pen::Float64 = -0.1
    goal_reward::Float64 = 10
    stairs_penalty::Float64 = -10
    config::Int = 1
    room::Room  = Room(configuration=config)
    sspace::SS = ContinuousRoombaStateSpace()
    aspace::AS = RoombaActions()
    _amap::Union{Nothing, Dict{RoombaAct, Int}} = gen_amap(aspace)
end

# state-space definitions
struct ContinuousRoombaStateSpace end
```

```julia
"""
Specify a DiscreteRoombaStateSpace
- `x_step::Float64` distance between discretized points in x
- `y_step::Float64` distance between discretized points in y
- `th_step::Float64` distance between discretized points in theta
- `XLIMS::Vector` boundaries of room (x-dimension)
- `YLIMS::Vector` boundaries of room (y-dimension)

"""
struct DiscreteRoombaStateSpace
      x_step::Float64
      y_step::Float64
      th_step::Float64
      XLIMS::Vector
      YLIMS::Vector
end

# function to construct DiscreteRoombaStateSpace:
# `num_x_pts::Int` number of points to discretize x range to
# `num_y_pts::Int` number of points to discretize yrange to
# `num_th_pts::Int` number of points to discretize th range to
function DiscreteRoombaStateSpace(num_x_pts::Int, num_y_pts::Int,
num_theta_pts::Int)

      # hardcoded room-limits
      # watch for consistency with env_room
      XLIMS = [-30.0, 20.0]
      YLIMS = [-30.0, 10.0]

      return DiscreteRoombaStateSpace((XLIMS[2]-XLIMS[1])/(num_x_pts-1),
                                    (YLIMS[2]-YLIMS[1])/(num_y_pts-1),
                                    2*pi/(num_theta_pts-1),
                                    XLIMS,YLIMS)
end




"""
Define the Roomba POMDP
```

```julia
Fields:
- `sensor::T` struct specifying the sensor used (Lidar or Bump)
- `mdp::T` underlying RoombaMDP
"""
struct RoombaPOMDP{T, O} <: POMDP{RoombaState, RoombaAct, O}
      sensor::T
      mdp::RoombaMDP
end

# observation models
struct Bumper

end
POMDPs.obstype(::Type{Bumper}) = Bool
POMDPs.obstype(::Bumper) = Bool

struct Lidar
      ray_stdev::Float64 # measurement noise: see POMDPs.observation
definition
                        # below for usage
end
Lidar() = Lidar(0.1)

POMDPs.obstype(::Type{Lidar}) = Float64
POMDPs.obstype(::Lidar) = Float64 #float64(x)

struct DiscreteLidar
      ray_stdev::Float64
      disc_points::Vector{Float64} # cutpoints: endpoints of (0, Inf)
assumed
end

POMDPs.obstype(::Type{DiscreteLidar}) = Int
POMDPs.obstype(::DiscreteLidar) = Int
DiscreteLidar(disc_points) = DiscreteLidar(Lidar().ray_stdev, disc_points)



# Shorthands
const RoombaModel = Union{RoombaMDP, RoombaPOMDP}
const BumperPOMDP = RoombaPOMDP{Bumper, Bool}
const LidarPOMDP = RoombaPOMDP{Lidar, Float64}
const DiscreteLidarPOMDP = RoombaPOMDP{DiscreteLidar, Int}

# access the mdp of a RoombaModel
mdp(e::RoombaMDP) = e
mdp(e::RoombaPOMDP) = e.mdp


# RoombaPOMDP Constructor
function RoombaPOMDP(sensor, mdp)
      RoombaPOMDP{typeof(sensor), obstype(sensor)}(sensor, mdp)
end

RoombaPOMDP(;sensor=Bumper(), mdp=RoombaMDP()) = RoombaPOMDP(sensor,mdp)

# function to determine if there is contact with a wall
wall_contact(e::RoombaModel, state) = wall_contact(mdp(e).room, state[1:2])

POMDPs.actions(m::RoombaModel) = mdp(m).aspace
POMDPs.n_actions(m::RoombaModel) = length(mdp(m).aspace)

# maps a RoombaAct to an index in a RoombaModel with discrete actions
function POMDPs.actionindex(m::RoombaModel, a::RoombaAct)
      if mdp(m)._amap != nothing
      return mdp(m)._amap[a]
      else
      error("Action index not defined for continuous actions.")
      end
end

# function to get goal xy location for heuristic controllers
function get_goal_xy(m::RoombaModel)
      grn = mdp(m).room.goal_rect
      gwn = mdp(m).room.goal_wall
      gr = mdp(m).room.rectangles[grn]
      corners = gr.corners
      if gwn == 4
      return (corners[1,:] + corners[4,:]) / 2.
      else
      return (corners[gwn,:] + corners[gwn+1,:]) / 2.
      end
end
```

```julia
# initializes x,y,th of Roomba in the room
function POMDPs.initialstate(m::RoombaModel, rng::AbstractRNG)
      e = mdp(m)
      x, y = init_pos(e.room, rng)
      th = rand() * 2*pi - pi

      is = RoombaState(x, y, th, 0.0)

      if mdp(m).sspace isa DiscreteRoombaStateSpace
      isi = stateindex(m, is)
      is = index_to_state(m, isi)
      end

      return is
end

# transition Roomba state given curent state and action
function POMDPs.transition(m::RoombaModel,
                        s::AbstractVector{Float64},
                        a::AbstractVector{Float64})

      e = mdp(m)
      v, om = a
      v = clamp(v, 0.0, e.v_max)
      om = clamp(om, -e.om_max, e.om_max)

      # propagate dynamics without wall considerations
      x, y, th, _ = s
      dt = e.dt

      # dynamics assume robot rotates and then translates
      next_th = wrap_to_pi(th + om*dt)

      # make sure we arent going through a wall
      p0 = SVector(x, y)
      heading = SVector(cos(next_th), sin(next_th))
      des_step = v*dt
      next_x, next_y = legal_translate(e.room, p0, heading, des_step)

      # Determine whether goal state or stairs have been reached
      grn = mdp(m).room.goal_rect
      gwn = mdp(m).room.goal_wall
      srn = mdp(m).room.stair_rect
      swn = mdp(m).room.stair_wall
      gr = mdp(m).room.rectangles[grn]
      sr = mdp(m).room.rectangles[srn]
      next_status = 1.0*contact_wall(gr, gwn, [next_x, next_y]) -
1.0*contact_wall(sr, swn, [next_x, next_y])

      # define next state
      sp = RoombaState(next_x, next_y, next_th, next_status)

      if mdp(m).sspace isa DiscreteRoombaStateSpace
      # round the states to nearest grid point
      si = stateindex(m, sp)
      sp = index_to_state(m, si)
      end

      return Deterministic(sp)
end

# enumerate all possible states in a DiscreteRoombaStateSpace
function POMDPs.states(m::RoombaModel)
      if mdp(m).sspace isa DiscreteRoombaStateSpace
      ss = mdp(m).sspace
      x_states = range(ss.XLIMS[1], stop=ss.XLIMS[2], step=ss.x_step)
      y_states = range(ss.YLIMS[1], stop=ss.YLIMS[2], step=ss.y_step)
      th_states = range(-pi, stop=pi, step=ss.th_step)
      statuses = [-1.,0.,1.]
      return vec(collect(RoombaState(x,y,th,st) for x in x_states, y in
y_states, th in th_states, st in statuses))
      else
      return mdp(m).sspace
      end
end

# return the number of states in a DiscreteRoombaStateSpace
function POMDPs.n_states(m::RoombaModel)
      if mdp(m).sspace isa DiscreteRoombaStateSpace
      ss = mdp(m).sspace
      nstates = prod((convert(Int, diff(ss.XLIMS)[1]/ss.x_step)+1,
                              convert(Int, diff(ss.YLIMS)[1]/ss.y_step)+1,
                              round(Int, 2*pi/ss.th_step)+1,
                              3))
      return nstates
      else
      error("State-space must be DiscreteRoombaStateSpace.")
      end
end

# map a RoombaState to an index in a DiscreteRoombaStateSpace
function POMDPs.stateindex(m::RoombaModel, s::RoombaState)
      if mdp(m).sspace isa DiscreteRoombaStateSpace
      ss = mdp(m).sspace
      xind = floor(Int, (s[1] - ss.XLIMS[1]) / ss.x_step + 0.5) + 1
      yind = floor(Int, (s[2] - ss.YLIMS[1]) / ss.y_step + 0.5) + 1
      thind = floor(Int, (s[3] - (-pi)) / ss.th_step + 0.5) + 1
      stind = convert(Int, s[4] + 2)

      lin = LinearIndices((convert(Int, diff(ss.XLIMS)[1]/ss.x_step)+1,
                          convert(Int, diff(ss.YLIMS)[1]/ss.y_step)+1,
                          round(Int, 2*pi/ss.th_step)+1,
                          3))
      return lin[xind,yind,thind,stind]
      else
      error("State-space must be DiscreteRoombaStateSpace.")
      end
end

# map an index in a DiscreteRoombaStateSpace to the corresponding
RoombaState
function index_to_state(m::RoombaModel, si::Int)
      if mdp(m).sspace isa DiscreteRoombaStateSpace
      ss = mdp(m).sspace
      lin = CartesianIndices((convert(Int, diff(ss.XLIMS)[1]/ss.x_step)+1,
                          convert(Int, diff(ss.YLIMS)[1]/ss.y_step)+1,
                          round(Int, 2*pi/ss.th_step)+1,
                          3))

      xi,yi,thi,sti = Tuple(lin[si])

      x = ss.XLIMS[1] + (xi-1) * ss.x_step
      y = ss.YLIMS[1] + (yi-1) * ss.y_step
      th = -pi + (thi-1) * ss.th_step
      st = sti - 2
      return RoombaState(x,y,th,st)

      else
      error("State-space must be DiscreteRoombaStateSpace.")
      end
end


# defines reward function R(s,a,s')
function POMDPs.reward(m::RoombaModel,
                  s::AbstractVector{Float64},
                  a::AbstractVector{Float64},
                  sp::AbstractVector{Float64})

#function POMDPs.reward(m::RoombaPOMDP{Bumper,Bool},
#                        s::RoombaState,
#                        a::RoombaAct,
#                        sp::RoombaState)

      # penalty for each timestep elapsed
      cum_reward = mdp(m).time_pen

      # penalty for bumping into wall (not incurred for consecutive
contacts)
      previous_wall_contact = wall_contact(m,s)
      current_wall_contact = wall_contact(m,sp)
      if(!previous_wall_contact && current_wall_contact)
      cum_reward += mdp(m).contact_pen
      end

      # terminal rewards
      cum_reward += mdp(m).goal_reward*(sp.status == 1.0)
      cum_reward += mdp(m).stairs_penalty*(sp.status == -1.0)

      return cum_reward
end

# determine if a terminal state has been reached
POMDPs.isterminal(m::RoombaModel, s::AbstractVector{Float64}) =
abs(s.status) > 0.0

# Bumper POMDP observation
```

```julia
function POMDPs.observation(m::BumperPOMDP,
                              a::AbstractVector{Float64},
                              sp::AbstractVector{Float64})
      return Deterministic(wall_contact(m, sp)) # in {0.0,1.0}
end

POMDPs.n_observations(m::BumperPOMDP) = 2
POMDPs.observations(m::BumperPOMDP) = [false, true]

# Lidar POMDP observation
function POMDPs.observation(m::LidarPOMDP,
                              a::AbstractVector{Float64},
                              sp::AbstractVector{Float64})
      x, y, th = sp

      # determine uncorrupted observation
      rl = ray_length(mdp(m).room, [x, y], [cos(th), sin(th)])

      # compute observation noise
      sigma = m.sensor.ray_stdev * max(rl, 0.01)

      # disallow negative measurements
      return Truncated(Normal(rl, sigma), 0.0, Inf)
end

function POMDPs.n_observations(m::LidarPOMDP)
      error("n_observations not defined for continuous observations.")
end

function POMDPs.observations(m::LidarPOMDP)
      error("LidarPOMDP has continuous observations. Use DiscreteLidarPOMDP
for discrete observation spaces.")
end

# DiscreteLidar POMDP observation
function POMDPs.observation(m::DiscreteLidarPOMDP,
                              a::AbstractVector{Float64},
                              sp::AbstractVector{Float64})

      m_lidar = LidarPOMDP(Lidar(m.sensor.ray_stdev), mdp(m))

      d = observation(m_lidar, a, sp)
      disc_points = [-Inf, m.sensor.disc_points..., Inf]

      d_disc = diff(cdf.(d, disc_points))

      return SparseCat(1:length(d_disc), d_disc)
end

POMDPs.n_observations(m::DiscreteLidarPOMDP) = length(m.sensor.disc_points)
+ 1
POMDPs.observations(m::DiscreteLidarPOMDP) = vec(1:n_observations(m))

# define discount factor
POMDPs.discount(m::RoombaModel) = 0.95

# struct to define an initial distribution over Roomba states
struct RoombaInitialDistribution{M<:RoombaModel}
      m::M
end

# definition of initialstate and initialstate_distribution for Roomba
environment
POMDPs.rand(rng::AbstractRNG, d::RoombaInitialDistribution) =
initialstate(d.m, rng)
POMDPs.initialstate_distribution(m::RoombaModel) =
RoombaInitialDistribution(m)

# Render a room and show robot
function render(ctx::CairoContext, m::RoombaModel, step)
      env = mdp(m)
      state = step[:sp]

      radius = ROBOT_W*6

      # render particle filter belief
      if haskey(step, :bp)
      bp = step[:bp]
      if bp isa AbstractParticleBelief
            for p in particles(bp)
                  x, y = transform_coords(p[1:2])
                  arc(ctx, x, y, radius, 0, 2*pi)
                  set_source_rgba(ctx, 0.6, 0.6, 1, 0.3)
                  fill(ctx)
            end
      end
      end

      # Render room
      render(env.room, ctx)

      # Find center of robot in frame and draw circle
      x, y = transform_coords(state[1:2])
      arc(ctx, x, y, radius, 0, 2*pi)
      set_source_rgb(ctx, 1, 0.6, 0.6)
      fill(ctx)

      # Draw line indicating orientation
      move_to(ctx, x, y)
      end_point = [state[1] + ROBOT_W*cos(state[3])/2, state[2] +
ROBOT_W*sin(state[3])/2]
      end_x, end_y = transform_coords(end_point)
      line_to(ctx, end_x, end_y)
      set_source_rgb(ctx, 0, 0, 0)
      stroke(ctx)
      return ctx
end

function render(m::RoombaModel, step)
      io = IOBuffer()
      c = CairoSVGSurface(io, 800, 600)
      ctx = CairoContext(c)
      render(ctx, m, step)
      finish(c)
      return HTML(String(take!(io)))
end
```

filtering.jl

```julia
# specification of particle filters for the bumper and lidar Roomba
environments
# maintained by {jmorton2,kmenda}@stanford.edu

import POMDPs

# structs specifying resamplers for bumper and lidar sensors
struct BumperResampler
      n::Int # number of particles
end

struct LidarResampler
      n::Int # number of particles
      lvr::LowVarianceResampler
end

"""
Definition of the particle filter for the Roomba environment
Fields:
- `spf::SimpleParticleFilter` standard particle filter struct defined in
ParticleFilters.jl
- `v_noise_coeff::Float64` coefficient to scale particle-propagation noise
in velocity
- `om_noise_coeff::Float64`coefficient to scale particle-propagation noise
in turn-rate
"""
mutable struct RoombaParticleFilter <: POMDPs.Updater
      spf::SimpleParticleFilter
      v_noise_coeff::Float64
      om_noise_coeff::Float64
end

# Resample function for weights in {0,1} necessary for bumper sensor
function ParticleFilters.resample(br::BumperResampler,
b::WeightedParticleBelief{RoombaState}, rng::AbstractRNG)
      new = RoombaState[]
      for (p, w) in weighted_particles(b)
      if w == 1.0
            push!(new, p)
      else
            @assert w == 0
      end
      end
      extras = rand(rng, new, br.n-length(new))
      for p in extras
      push!(new, p)
      end
      return ParticleCollection(new)
end

# resample function for unweighted particles
function
ParticleFilters.resample(br::Union{BumperResampler,LidarResampler}, b,
rng::AbstractRNG)
      ps = Array{RoombaState}(undef, br.n)
      for i in 1:br.n
      ps[i] = rand(rng, b)
      end
      return ParticleCollection(ps)
end

# Resample function for continuous weights necessary for lidar sensor
function ParticleFilters.resample(lr::LidarResampler,
b::WeightedParticleBelief{RoombaState}, rng::AbstractRNG)

      ps = resample(lr.lvr, b, rng)
      return ps
end

# Modified Update function adds noise to the actions that propagate
particles
function POMDPs.update(up::RoombaParticleFilter,
b::ParticleCollection{RoombaState}, a, o)
      ps = particles(b)
      pm = up.spf._particle_memory
      wm = up.spf._weight_memory
      resize!(pm, 0)
      resize!(wm, 0)
      sizehint!(pm, n_particles(b))
      sizehint!(wm, n_particles(b))
      all_terminal = true
      for i in 1:n_particles(b)
      s = ps[i]
      if !isterminal(up.spf.model, s)
            all_terminal = false
            # noise added here:
            a_pert = a + SVector(up.v_noise_coeff*(rand(up.spf.rng)-0.5), 
up.om_noise_coeff*(rand(up.spf.rng)-0.5))
            sp = generate_s(up.spf.model, s, a_pert, up.spf.rng)
            push!(pm, sp)
            push!(wm, obs_weight(up.spf.model, s, a_pert, sp, o))
      end
      end
      # if all particles are terminal, return previous belief state
      if all_terminal
      return b
      end

      return resample(up.spf.resample, 
WeightedParticleBelief{RoombaState}(pm, wm, sum(wm), nothing), up.spf.rng)
end

# initialize belief state
function ParticleFilters.initialize_belief(up::RoombaParticleFilter, 
d::Any)
      resample(up.spf.resample, d, up.spf.rng)
end
```

env_room.jl

```julia
# Code to define the environment room and rectangles used to define it
# maintained by {jmorton2,kmenda}@stanford.edu

# Define constants  -- all units in m
RW = 5. # room width
ROBOT_W = 1. # robot width
MARGIN = 1e-12

# Define rectangle type for constructing hallway
# corners: 4x2 np array specifying
#           bottom-left, top-left,
#           top-right, bottom-right corner
# walls: length 4 list of bools specifying
#       if left, top, right, bottom sides are
#       open (False) or walls (True)
mutable struct Rectangle
      corners::Array{Float64, 2}
      walls::Array{Bool, 1}
      segments::Array{LineSegment, 1}
      width::Float64
      height::Float64
      midpoint::Array{Float64, 1}
      area::Float64
      xl::Float64
      xu::Float64
      yl::Float64
      yu::Float64

      function Rectangle(
      corners::Array{Float64, 2},
      walls::Array{Bool, 1};
      goal_idx::Int=0,
      stair_idx::Int=0
      )

      retval = new()

      retval.corners = corners
      retval.walls = walls
      retval.width = corners[3, 1] - corners[2, 1]
      retval.height = corners[2, 2] - corners[1, 2]
      mean_vals = mean(corners, dims=1)
      retval.midpoint = SVector(mean_vals[1, 1], mean_vals[1, 2])

      # compute area in which robot could be initialized
      retval.xl = corners[2, 1]
      retval.xu = corners[3, 1]
      retval.yl = corners[1, 2]
      retval.yu = corners[2, 2]
      if walls[1]
            retval.width -= ROBOT_W/2
            retval.xl += ROBOT_W/2
      end
      if walls[2]
            retval.height -= ROBOT_W/2
            retval.yu -= ROBOT_W/2
      end
      if walls[3]
            retval.width -= ROBOT_W/2
            retval.xu -= ROBOT_W/2
      end
      if walls[4]
            retval.height -= ROBOT_W/2
            retval.yl += ROBOT_W/2
      end
      @assert retval.width > 0.0 && retval.height > 0.0 "Negative width or
height"
      retval.area = retval.width * retval.height

      retval.segments = [LineSegment(corners[i, :], corners[i+1, :],
(goal_idx == i), (stair_idx == i)) for i =1:3 if walls[i]]
      if walls[4]
            push!(retval.segments, LineSegment(corners[1, :], corners[4,
:], (goal_idx == 4), (stair_idx == 4)))
      end

      retval
      end
end
```

```julia
# Randomly initializes the robot in this rectangle
function init_pos(rect::Rectangle, rng)
      w = rect.xu - rect.xl
      h = rect.yu - rect.yl
      init_pos = SVector(rand(rng)*w + rect.xl, rand(rng)*h + rect.yl)

      init_pos
end

# Determines if pos (center of robot) is within the rectangle
function in_rectangle(rect::Rectangle, pos::AbstractVector{Float64})
      corners = rect.corners
      xlims = SVector(rect.xl - MARGIN, rect.xu + MARGIN)
      ylims = SVector(rect.yl - MARGIN, rect.yu + MARGIN)
      if xlims[1] < pos[1] < xlims[2]
      if ylims[1] < pos[2] < ylims[2]
            return true
      end
      end
      return false
end

# determines if pos (center of robot) is intersecting with a wall
# returns: -2, -Inf if center of robot not in room
#           -1, -Inf if not in wall contact
#           0~3, violation mag, indicating which wall has contact
#           if multiple, returns largest violation
function wall_contact(rect::Rectangle, pos::AbstractVector{Float64})
      if !(in_rectangle(rect, pos))
      return -2, -Inf
      end
      corners = rect.corners
      xlims = SVector(corners[2, 1], corners[3, 1])
      ylims = SVector(corners[1, 2], corners[2, 2])

      contacts = []
      contact_mags = []
      if pos[1] - ROBOT_W/2 <= xlims[1] + MARGIN && rect.walls[1]
      # in contact with left wall
      push!(contacts, 1)
      push!(contact_mags, abs(pos[1] - ROBOT_W/2 - xlims[1]))
      end
      if pos[2] + ROBOT_W/2 + MARGIN >= ylims[2] && rect.walls[2]
      # in contact with top wall
      push!(contacts, 2)
      push!(contact_mags, abs(pos[2] + ROBOT_W/2 - ylims[2]))
      end
      if pos[1] + ROBOT_W/2 + MARGIN >= xlims[2] && rect.walls[3]
      # in contact with right wall
      push!(contacts, 3)
      push!(contact_mags, abs(pos[1] + ROBOT_W/2 - xlims[2]))
      end
      if pos[2] - ROBOT_W/2 <= ylims[1] + MARGIN && rect.walls[4]
      # in contact with bottom wall
      push!(contacts, 4)
      push!(contact_mags, abs(pos[2] - ROBOT_W/2 - ylims[1]))
      end

      if length(contacts) == 0
      return -1, -Inf
      else
      return contacts[argmax(contact_mags)], maximum(contact_mags)
      end
end

# Find closest distance to any wall
function furthest_step(rect::Rectangle, pos::AbstractVector{Float64},
heading::AbstractVector{Float64})
      return minimum(furthest_step(seg, pos, heading, ROBOT_W/2) for seg in
rect.segments)
end

# computes the length of a ray from robot center to closest segment
# from p0 pointing in direction heading
function ray_length(rect::Rectangle, pos::AbstractVector{Float64},
heading::AbstractVector{Float64})
      return minimum(ray_length(seg, pos, heading) for seg in
rect.segments)
end

# Render rectangle based on segments
function render(rect::Rectangle, ctx::CairoContext)
      for seg in rect.segments
      render(seg, ctx)
      end
end

# generate consecutive rectangles that make up the room
# all rectangles share a full "wall" with an adjacent rectangle
# shared walls are not solid - just used to specify geometry
mutable struct Room
      rectangles::Array{Rectangle, 1}
      areas::Array{Float64, 1}
      goal_rect::Int  # Index of rectangle with goal state
      goal_wall::Int  # Index of wall that leads to goal
      stair_rect::Int # Index of rectangle with stairs
      stair_wall::Int # Index of wall that leads to stairs

      function Room(; configuration=1)

      retval = new()

      # Define different configurations for stair and goal locations
      goal_idxs = [0, 0, 0, 0]
      stair_idxs = [0, 0, 0, 0]
      if configuration == 2
            retval.goal_rect = 1
            retval.goal_wall = 4
            retval.stair_rect = 2
            retval.stair_wall = 1
      elseif configuration == 3
            retval.goal_rect = 4
            retval.goal_wall = 3
            retval.stair_rect = 2
            retval.stair_wall = 1
      else
            retval.goal_rect = 4
            retval.goal_wall = 3
            retval.stair_rect = 4
            retval.stair_wall = 4
      end
      goal_idxs[retval.goal_rect] = retval.goal_wall
      stair_idxs[retval.stair_rect] = retval.stair_wall

      # Initialize array of rectangles
      rectangles = []
      # Rectangle 1
      corners = [[-20-RW -20]; [-20-RW 0-RW]; [-20+RW 0-RW]; [-20+RW -20]]
      walls = [true, false, true, true] # top wall shared
      push!(rectangles, Rectangle(corners, walls, goal_idx=goal_idxs[1],
stair_idx=stair_idxs[1]))

      # Rectangle 2
      corners = [[-20-RW 0-RW]; [-20-RW 0+RW]; [-20+RW 0+RW]; [-20+RW
0-RW]]
      walls = [true, true, false, false] # bottom, right wall shared
      push!(rectangles, Rectangle(corners, walls, goal_idx=goal_idxs[2],
stair_idx=stair_idxs[2]))

      # Rectangle 3
      corners = [[-20+RW 0-RW]; [-20+RW 0+RW]; [10 0+RW]; [10 0-RW]]
      walls = [false, true, false, true] # left wall shared
      push!(rectangles, Rectangle(corners, walls, goal_idx=goal_idxs[3],
stair_idx=stair_idxs[3]))

      # Rectangle 4
      corners = [[10 0-RW]; [10 0+RW]; [10+RW 0+RW]; [10+RW 0-RW]]
      walls = [false, true, true, true] # left wall shared
      push!(rectangles, Rectangle(corners, walls, goal_idx=goal_idxs[4],
stair_idx=stair_idxs[4]))

      retval.rectangles = rectangles
      retval.areas = [r.area for r in rectangles]

      retval
      end
end

# Sample from multinomial distribution
function multinomial_sample(p::AbstractVector{Float64})
      rand_num = rand()
      for i = 1:length(p)
      if rand_num < sum(p[1:i])
            return i
      end
      end
end
```

```julia
# Initialize the robot randomly in the room
# Randomly select a rectangle weighted by initializable area
function init_pos(r::Room, rng::AbstractRNG)
      norm_areas = r.areas/sum(r.areas)
      rect = multinomial_sample(norm_areas)
      return init_pos(r.rectangles[rect], rng)
end

# Determines if pos is in contact with a wall
# returns bool indicating contact
function wall_contact(r::Room, pos::AbstractVector{Float64})
      for (i, rect) in enumerate(r.rectangles)
      wc, _ = wall_contact(rect, pos)
      if wc >= 0
            return true
      end
      end
      return false
end

# Determines if pos is in contact with a specific wall
# returns true if true
function contact_wall(r::Rectangle, wall::Int, pos::Array{Float64, 1})
      wc,_ = wall_contact(r, pos)
      return wc == wall
end

# Determines if pos (center of robot) is within the room
function in_room(r::Room, pos::AbstractVector{Float64})
      return any([in_rectangle(rect, pos) for rect in r.rectangles])
end

# Attempts to translate from pos0 in direction heading for des_step without
violating boundaries
function legal_translate(r::Room, pos0::AbstractVector{Float64},
heading::AbstractVector{Float64}, des_step::Float64)
      fs = minimum(furthest_step(rect, pos0, heading) for rect in
r.rectangles)
      fs = min(des_step, fs)
      pos1 = pos0 + fs*heading
      if !in_room(r, pos1)
      return pos0
      else
      return pos1
      end
end

# computes the length of a ray from robot center to closest segment
# from p0 pointing in direction heading
# inputs: p0: array specifying initial point
#            heading: array specifying heading unit vector
#            R: robot radius [m]
# outputs: ray_length [m]
function ray_length(r::Room, pos0::AbstractVector{Float64},
heading::AbstractVector{Float64})
      return minimum(ray_length(rect, pos0, heading) for rect in
r.rectangles)
end

# Render room based on individual rectangles
function render(r::Room, ctx::CairoContext)
      for rect in r.rectangles
      render(rect, ctx)
      end
end
```

line_segment_utils.jl

```
# functions for determining whether the Roomba's path interects
# with a line segment and struct defining line segments
# maintained by {jmorton2,kmenda}@stanford.edu


MARGIN = 1e-8
"""
finds the real points of intersection between a line and a circle
inputs:
- `p0::AbstractVector{Float64}` anchor point
- `uvec::AbstractVector{Float64}` unit vector specifying heading
- `p1::AbstractVector{Float64}` centroid (x,y) of a circle
- `R::Float64` radius of a circle
returns:
- `R1,R2::Float64` where R1,R2 are lengths of vec to get from p0 to the
intersecting
      points. If intersecting points are imaginary, returns `nothing` in
their place
"""
function real_intersect_line_circle(p0::AbstractVector{Float64},
                                    uvec::AbstractVector{Float64},
                                    p1::AbstractVector{Float64},
                                    R::Float64)
      # these equations were generated by Mathematica using the following
command:
      # Simplify[Solve[x0 + dx0 * R0 == x &&
      #                 y0 + dy0 *R0 == y &&
      #                 (x - x1)^2 + (y - y1)^2 == R,
      #                 {x, y, R0}]]
      # Where the solutions for R0 are called R1 and R2 here
      x0, y0 = p0
      dx0, dy0 = uvec
      x1, y1 = p1

      radicand = dx0^2 * (dy0^2 * (R - (x0 - x1)^2) + dx0^2 * (R - (y0 -
y1)^2) + 2*dx0*dy0*(x0 - x1)*(y0 - y1))
      if radicand < 0 # intersecting points are imaginary
      return nothing, nothing
      else
      R1 = (1/(dx0*(dx0^2 + dy0^2)))*(dx0^2 * (-x0 + x1) + sqrt(radicand) +
dx0*dy0*(-y0 + y1))
      R2 = (1/(dx0*(dx0^2 + dy0^2)))*(dx0^2 * (-x0 + x1) - sqrt(radicand) +
dx0*dy0*(-y0 + y1))
      return R1, R2
      end
end

"""
finds the intersection between a line and a line segment
inputs:
- `p0::AbstractVector{Float64}` anchor point
- `uvec::AbstractVector{Float64}` unit vector specifying heading
- `p1, p2::AbstractVector{Float64}` x,y of the endpoints of the segment
returns:
- `sol::AbstractVector{Float64}` x,y of intersection or `nothing` if
doesn't intersect
"""
function intersect_line_linesegment(p0::AbstractVector{Float64},
uvec::AbstractVector{Float64}, p1::AbstractVector{Float64},
p2::AbstractVector{Float64})
      dx, dy = uvec
      n = [-dy, dx]
      dprod1 = dot(n, p1-p0)
      dprod2 = dot(n, p2-p0)

      if sign(dprod1) != sign(dprod2)
      # there's an intersection

      # these equations were generated by Mathematica using the following
command:
      # Simplify[Solve[x0 + dx0 * R0 == x1 + dx1 * R1 && y0 + dy0 *R0 == y1
+ dy1 *R1, {R0,R1}]]
      # Where R0 is the length of the segment originating from p0

      x0, y0 = p0
      x1, y1 = p1
      x2, y2 = p2
      dx0, dy0 = uvec
      dx1 = x2 - x1
      dy1 = y2 - y1
      R = (dy1*x0 - dy1*x1 - dx1*y0 + dx1*y1)/(dx1*dy0 - dx0*dy1)
```

```julia
      if R >= 0
            return R
      else
            return nothing
      end
      else
      return nothing
      end
end

# Define LineSegment
mutable struct LineSegment
      p1::Array{Float64, 1} # anchor point of line-segment
      p2::Array{Float64, 1} # anchor point of line-segment
      goal::Bool # used for rendering purposes
      stairs::Bool # used for rendering purposes
end

"""
determines if traveling in heading from p0 intersects the line passing
through this segment
inputs:
- `ls::LineSegment` line segment under test
- `p0::AbstractVector{Float64}` initial point being travelled from
- `heading::AbstractVector{Float64}` heading unit vector
returns:
- `::Bool` that is true if pointing toward segment
"""
function pointing_toward_segment(ls::LineSegment,
p0::AbstractVector{Float64}, heading::AbstractVector{Float64})
      dp12 = ls.p2 - ls.p1
      normalize!(dp12)
      np12 = [-dp12[2], dp12[1]]

      # ensure it points toward p0
      if dot(np12, p0 - ls.p1) < 0
      np12 *= -1.0
      end

      # return true if heading projects in the opposite direction of np12
      dot(np12, heading) < 0.0
end
```

```
"""
computes the length of a ray from robot center to segment from p0 pointing
in direction heading
inputs:
- `ls: functions for determining whether the Roomba's path interects
# with a line segment and struct defining line segments
# maintained by {jmorton2,kmenda}@stanford.edu


MARGIN = 1e-8
"""
finds the real points of intersection between a line and a circle
inputs:
- `p0::AbstractVector{Float64}` anchor point
- `uvec::AbstractVector{Float64}` unit vector specifying heading
- `p1::AbstractVector{Float64}` centroid (x,y) of a circle
- `R::Float64` radius of a circle
returns:
- `R1,R2::Float64` where R1,R2 are lengths of vec to get from p0 to the
intersecting
      points. If intersecting points are imaginary, returns `nothing` in
their place
"""
function real_intersect_line_circle(p0::AbstractVector{Float64},
                                    uvec::AbstractVector{Float64},
                                    p1::AbstractVector{Float64},
                                    R::Float64)
      # these equations were generated by Mathematica using the following
command:
      # Simplify[Solve[x0 + dx0 * R0 == x &&
      #                  y0 + dy0 *R0 == y &&
      #                  (x - x1)^2 + (y - y1)^2 == R,
      #                  {x, y, R0}]]
      # Where the solutions for R0 are called R1 and R2 here
      x0, y0 = p0
      dx0, dy0 = uvec
      x1, y1 = p1

      radicand = dx0^2 * (dy0^2 * (R - (x0 - x1)^2) + dx0^2 * (R - (y0 -
y1)^2) + 2*dx0*dy0*(x0 - x1)*(y0 - y1))
```

```
      if radicand < 0 # intersecting points are imaginary
      return nothing, nothing
      else
      R1 = (1/(dx0*(dx0^2 + dy0^2)))*(dx0^2 * (-x0 + x1) + sqrt(radicand) +
dx0*dy0*(-y0 + y1))
      R2 = (1/(dx0*(dx0^2 + dy0^2)))*(dx0^2 * (-x0 + x1) - sqrt(radicand) +
dx0*dy0*(-y0 + y1))
      return R1, R2
      end
end

"""
finds the intersection between a line and a line segment
inputs:
- `p0::AbstractVector{Float64}` anchor point
- `uvec::AbstractVector{Float64}` unit vector specifying heading
- `p1, p2::AbstractVector{Float64}` x,y of the endpoints of the segment
returns:
- `sol::AbstractVector{Float64}` x,y of intersection or `nothing` if
doesn't intersect
"""
function intersect_line_linesegment(p0::AbstractVector{Float64},
uvec::AbstractVector{Float64}, p1::AbstractVector{Float64},
p2::AbstractVector{Float64})
      dx, dy = uvec
      n = [-dy, dx]
      dprod1 = dot(n, p1-p0)
      dprod2 = dot(n, p2-p0)

      if sign(dprod1) != sign(dprod2)
      # there's an intersection

      # these equations were generated by Mathematica using the following
command:
      # Simplify[Solve[x0 + dx0 * R0 == x1 + dx1 * R1 && y0 + dy0 *R0 == y1
+ dy1 *R1, {R0,R1}]]
      # Where R0 is the length of the segment originating from p0

      x0, y0 = p0
      x1, y1 = p1
      x2, y2 = p2
      dx0, dy0 = uvec
      dx1 = x2 - x1
      dy1 = y2 - y1
      R = (dy1*x0 - dy1*x1 - dx1*y0 + dx1*y1)/(dx1*dy0 - dx0*dy1)
      if R >= 0
            return R
      else
            return nothing
      end
      else
      return nothing
      end
end

# Define LineSegment
mutable struct LineSegment
      p1::Array{Float64, 1} # anchor point of line-segment
      p2::Array{Float64, 1} # anchor point of line-segment
      goal::Bool # used for rendering purposes
      stairs::Bool # used for rendering purposes
end

"""
determines if traveling in heading from p0 intersects the line passing
through this segment
inputs:
- `ls::LineSegment` line segment under test
- `p0::AbstractVector{Float64}` initial point being travelled from
- `heading::AbstractVector{Float64}` heading unit vector
returns:
- `::Bool` that is true if pointing toward segment
"""
function pointing_toward_segment(ls::LineSegment,
p0::AbstractVector{Float64}, heading::AbstractVector{Float64})
      dp12 = ls.p2 - ls.p1
      normalize!(dp12)
      np12 = [-dp12[2], dp12[1]]

      # ensure it points toward p0
      if dot(np12, p0 - ls.p1) < 0
      np12 *= -1.0
      end
```

```julia
      # return true if heading projects in the opposite direction of np12
      dot(np12, heading) < 0.0
end


"""
computes the length of a ray from robot center to segment from p0 pointing
in direction heading
inputs:
- `ls::LineSegment` line segment under test
- `p0::AbstractVector{Float64}` initial point being travelled from
- `heading::AbstractVector{Float64}` heading unit vector
returns:
- `::Float64` that is the length of the ray
"""
function ray_length(ls::LineSegment, p0::AbstractVector{Float64},
heading::AbstractVector{Float64})
      p1 = ls.p1
      p2 = ls.p2

      ray_length = Inf

      if !(pointing_toward_segment(ls, p0, heading))
      return ray_length
      else
      intr = intersect_line_linesegment(p0, heading, p1, p2)
      if intr != nothing
            return intr
      else
            return Inf
      end
      end
end


"""
computes the furthest step a robot of radius R can take
inputs:
- `ls::LineSegment` line segment under test
- `p0::AbstractVector{Float64}` initial point being travelled from
- `heading::AbstractVector{Float64}` heading unit vector
- `R::Float64` radius of robot
```

```
returns:
- `furthest_step::Float64` furthest step the robot can take
--
The way this is computed is by seeing if a ray originating from
p0 in direction heading intersects the following object. Consider
the shape made by moving the robot along the length of the segment.
We can construct this shape by placing a circle with radius of
the robot radius R at each end, and connecting their sides by shifting
segment line out to its left and right by R.
If the line from p0 intersects this object, then choosing the closest
intersection gives the point at which the robot would stop if traveling
along this line.
"""
function furthest_step(ls::LineSegment, p0::AbstractVector{Float64},
heading::AbstractVector{Float64}, R::Float64)
      furthest_step = Inf

      if !(pointing_toward_segment(ls, p0, heading))
      return furthest_step
      end

      # heading along segment
      dp12 = ls.p2 - ls.p1
      normalize!(dp12)
      np12 = [-dp12[2], dp12[1]]

      # project sides out a robot radius
      p1l = ls.p1 - R*np12
      p1r = ls.p1 + R*np12
      p2l = ls.p2 - R*np12
      p2r = ls.p2 + R*np12

      # intesection with p1
      R1,R2 = real_intersect_line_circle(p0, heading, ls.p1, R/2)

      if R1 != nothing
      if R1 > -MARGIN && R1 < furthest_step
            furthest_step = max(R1, 0.0)
      end
      if R2 > -MARGIN && R2 < furthest_step
            furthest_step = max(R2, 0.0)
      end
      end

      # intesection with p2
      R1, R2 = real_intersect_line_circle(p0, heading, ls.p2, R/2)
      if R1 != nothing
      if R1 > -MARGIN && R1 < furthest_step
            furthest_step = max(R1, 0.0)
      end
      if R2 > -MARGIN && R2 < furthest_step
            furthest_step = max(R2, 0.0)
      end
      end

      # intersection with the segment
      Rl = intersect_line_linesegment(p0, heading, p1l, p2l)
      if Rl != nothing
      if Rl > -MARGIN && Rl < furthest_step
            furthest_step = max(Rl, 0.0)
      end
      end

      Rr = intersect_line_linesegment(p0, heading, p1r, p2r)
      if Rr != nothing
      if Rr > -MARGIN && Rr < furthest_step
            furthest_step = max(Rr, 0.0)
      end
      end

      if Rl != nothing && Rr != nothing
      if Rl > 0 && Rr < 0
            # something wrong
            println("Travelling through a wall!!")
      end
      end

      furthest_step
end

# Transform coordinates in world frame to coordinates used for rendering
function transform_coords(pos::AbstractVector{Float64})
      x, y = pos
      # Specify dimensions of window
      h = 600
      w = 600

      # Perform conversion
      x_trans = (x + 30.0)/50.0*h
      y_trans = -(y - 20.0)/50.0*w

      x_trans, y_trans
end

# Draw line in gtk window based on start and end coordinates
function render(ls::LineSegment, ctx::CairoContext)
      start_x, start_y = transform_coords(ls.p1)
      if ls.goal
      set_source_rgb(ctx, 0, 1, 0)
      elseif ls.stairs
      set_source_rgb(ctx, 1, 0, 0)
      else
      set_source_rgb(ctx, 0, 0, 0)
      end
      move_to(ctx, start_x, start_y)
      end_x, end_y = transform_coords(ls.p2)
      line_to(ctx, end_x, end_y)
      stroke(ctx)
end
```