probcomp /
tasks2D

Code Issues Pull requests Actions Projects Security Insights

tasks2D / notebooks_clean / KidnappedRobot.ipynb

georgematheos clean messy print 4baca6b · last year

6.22 MB

In [1]:

```
import Pkg
Pkg.activate("../Tasks2D")
```

```
Activating project at `~/Developer/tasks2D/Tasks2D`
```

In [2]:

```
using Revise      # For development; makes it so modifications
                  # to imported modules are immediately reflected in this

import LineWorlds # Local module with code for 2D maps where
                  # the primitive objects are line segments.
const L = LineWorlds
const Geo = L.Geometry;
```

In [3]:

```
using Gen
```

In [4]:

```
includet("KidnappedRobot/visualization.jl")
```

Define environment model

In [5]:

```
### Initial state distribution ###

mvuniform = L.ProductDistribution(uniform);
@gen function uniform_agent_pos(params)
    mins, maxs = params.bounding_box
    pos ~ mvuniform(mins, maxs)

    return pos
end
```

```
DynamicDSLFunction{Any}(Dict{Symbol, Any}(), Dict{Symbol, Any}(), Type[An
y], false, Union{Nothing, Some{Any}}[nothing], var"##uniform_agent_pos#31
4", Bool[0], false)
```

In [6]:

```
### Transition model ###

# Load: `det_next_pos`, which computes the determinized effect of actions
# Load: `handle_wall_intersection` to handle wall intersections
includet("KidnappedRobot/motion_model_utils.jl")

# Motion model accepts the previous world state (the agent's previous pos
# and an action in [:up, :down, :left, :right, :stay]
@gen function motion_model(prev_pos, action, params)
    # Move the agent up/down/left/right by params.step.Δ units.
    np = det_next_pos(prev_pos, action, params.step.Δ)

    # Have an affordance in the model for the agent to randomly
    # re-locate to a new position.
    is_kidnapped ~ bernoulli(params.p_kidnapped)

    if !is_kidnapped
        # In normal operation, the agent moves to `np`, plus
```

```
        # a bit of stochastic noise.
        pos ~ broadcasted_normal(np, params.step.σ)

        # If `np` plus the noise
        # would have the agent collide with a wall, the agent
        # halts preemptively.
        next_pos = handle_wall_intersection(prev_pos, pos, params.map)

    else
        # If the robot was kidnapped, it could appear anywhere.
        # {*} syntax inlines the random choices (here, `:pos`) from the
        # `uniform_agent_pos` generative function into this one.
        next_pos = {*} ~ uniform_agent_pos(params)
    end

    return next_pos
end
```

```
DynamicDSLFunction{Any}(Dict{Symbol, Any}(), Dict{Symbol, Any}(), Type[Any, Any, Any], false, Union{Nothing, Some{Any}}[nothing, nothing, nothing], var"##motion_model#315", Bool[0, 0, 0], false)
```

In [7]:

```
### Observation model ###

# Load: `get_sensor_args`; `sensordist_2dp3`.
includet("KidnappedRobot/sensor_model_utils.jl")

# This observation model generates noisy LIDAR measurements
# from the agent to the surrounding walls.
# See the visuals below.
@gen function sensor_model(pos, params)
    sensor_args = get_sensor_args(pos, params)
    obs ~ L.sensordist_2dp3(sensor_args...)
    return obs
end
```

```
DynamicDSLFunction{Any}(Dict{Symbol, Any}(), Dict{Symbol, Any}(), Type[Any, Any], false, Union{Nothing, Some{Any}}[nothing, nothing], var"##sensor_model#316", Bool[0, 0], false)
```

Define POMDP

In [8]:

```
import GenPOMDPs
```

In [9]:

```
# POMDP of this environment
pomdp = GenPOMDPs.GenPOMDP(
    uniform_agent_pos,       # INIT   : params                          ⇝ sta
    motion_model,            # STEP   : prev_state, action, params      ⇝ sta
    sensor_model,            # OBS    : state, params                   ⇝ obs
    (state, action) -> -1.   # UTILITY: state, action, params           → uti
                             # "⇝" denotes a Generative Function; "→" den
)
```

```
GenPOMDPs.GenPOMDP(DynamicDSLFunction{Any}(Dict{Symbol, Any}(), Dict{Symbol, Any}(), Type[Any], false, Union{Nothing, Some{Any}}[nothing], var"##uniform_agent_pos#314", Bool[0], false), DynamicDSLFunction{Any}(Dict{Symbol, A
```

```
ny}(), Dict{Symbol, Any}(), Type[Any, Any, Any], false, Union{Nothing, Some
{Any}}[nothing, nothing, nothing], var"##motion_model#315", Bool[0, 0, 0],
false), DynamicDSLFunction{Any}(Dict{Symbol, Any}(), Dict{Symbol, Any}(), T
ype[Any, Any], false, Union{Nothing, Some{Any}}[nothing, nothing], var"##se
nsor_model#316", Bool[0, 0], false), var"#35#36"())
```

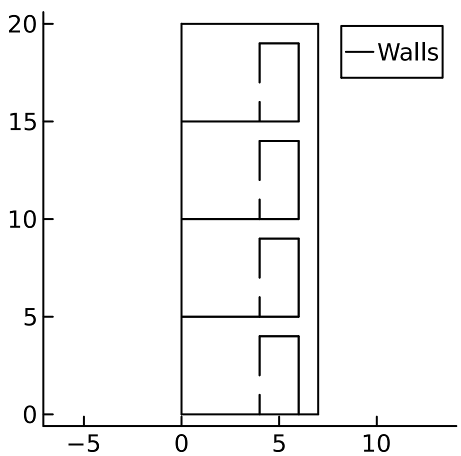
Load an environment

In [10]:

```
# Load function to construct a "hotel" map with a given number
# of identical rooms.
includet("KidnappedRobot/hotel_env.jl")

# Construct a hotel environment with 4 rooms.
(walls, bounding_box) = construct_hotel_env(4);
```

In [11]:

```
plot(size=(250, 250), aspect_ratio=:equal, grid=false)
plot!(walls, c=:black, label="Walls")
```



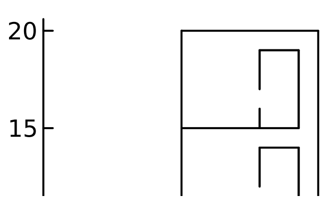
Add goal object to environment

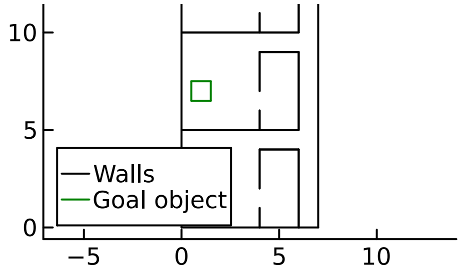
In [12]:

```
includet("KidnappedRobot/box.jl") # get `box_segments`, which draws a box

# Coordinates for where to place goal object in the map we loaded above
GOAL = [1., 7.]

goalobj = box_segments(GOAL);
```

In [13]:

```
plot(size=(250, 250), aspect_ratio=:equal, grid=false, legend=:bottomleft
plot!(walls, c=:black, label="Walls")
plot!(goalobj, c=:green, label="Goal object")
```





Ground truth world parameters

In [14]:

```
### Ground truth world model parameters ###
PARAMS = (;
    map = vcat(walls, goalobj),      # The map consists of the walls, and
    p_kidnapped = 0.,                # Probability the agent is kidnapped
    bounding_box = bounding_box,     # Bounding box for the environment
    step = (; Δ = 0.25, σ = 0.005 ), # step model arguments
    obs = (; fov = 2π, n_rays = 80,  # obs model arguments
        orientation=π/2,
        sensor_args = (;
            w = 5, s_noise = 0.02,
            outlier = 0.0001, outlier_vol = 100.0,
            zmax = 100.0
)));
```

Construct a particle filter

Next, we'll construct a 1-particle particle filter we can use for state estimation in this model. It will be based on a proposal distribution which uses a coarse-to-fine sequence of grid scans over the 2D environment to precisely localize the agent.

In [15]:

```
### Particle Filter args ###

# pf.jl defines `@get_pf`.  This macro simply yields a call `GenPOMDPs.pf
# grid proposal distribution.
#
# `GenPOMDPs.pf` is a function which accepts a POMDP as input, and parame
# behavior of a particle filter, and constructs a particle filter special
# of all POMDPs.
#
# `pf.jl` also defines some particle filtering proposal distributions bas
# coarse-to-fine grid scans.
includet("KidnappedRobot/pf.jl")

# Also load a file where I defined some default arguments for the particl
# proposal distributions.
includet("KidnappedRobot/default_pf_args.jl")

# Construct the POMDP the agent will use as it's mental world model while
# In this case, the mental-model GenPOMDP object will be the same as the g
# However, the exact distributions represented by the mental-model POMDP a
# be different, since we will give the agent a different set of parameter
```

```
# (I could potentially refactor GenPOMDPs so that the GenPOMDP object con
# rather than the user passing this in each time.  That would also more p
# formal definition of a POMDP.  However, I had in mind it may be convenie
# to have an explicit `params` argument which can control details of the 
agent_mental_model = GenPOMDPs.GenPOMDP(uniform_agent_pos, motion_model, 
# [We could just set agent_mental_model = pomdp; I write this out for ill

# We will have the agent's mental model suppose the
# motion noise and observation noise are higher than
# they truly are, and the agent will do particle filtering
# assuming kidnapping is impossible.  (The controller will handle
# kidnapping by resetting the particle filter.)
MENTAL_MODEL_PARAMS = overwrite_params(
    PARAMS;
    p_kidnapped=0.,
    step=(; σ = 0.1),
    sensor_args=(; s_noise=0.1)
)

# Arguments for 1 particle SMC. [The resampling args don't do anything, s
update_grid_args, initialization_grid_args, resampling_args = default_pf_

# Particle filter for inference in the mental world model
pf = @get_pf(agent_mental_model, MENTAL_MODEL_PARAMS, update_grid_args, i

# The particle filter object returned by GenPOMDPs.pf is a pair of a func
# which initializes a particle filter, `initial_pf_state = pf_init(observ
# which updates the filter, `new_pf_state = pf_update(pf_state, action, n
(pf_init, pf_update) = pf;
```

In [16]:

```
# In inference, we'll use this generative function by constructing a Cont
# GenPOMDPs.RolloutModel(pomdp, controller)
ctm = GenPOMDPs.ControlledTrajectoryModel(agent_mental_model)
ctm isa Gen.GenerativeFunction
```

```
true
```

In [68]:

```
trace_of_mental_model = simulate(ctm,
    (
        3,                          # n timesteps to simulate
        [:left, :left, :left],  # action sequence
        MENTAL_MODEL_PARAMS     # parameters
    )
)
# get_choices(trace_of_mental_model) # uncomment this, and delete `nothin
nothing
```

Baseline controller

As a baseline, we'll implement a controller which does state estimation without a particle filter; it will simply do a coarse-to-fine grid scan over the entire map to localize globally at every timestep.

To implement this, we'll simply use the `pf_init` function we obtained above when constructing our particle filter. That is, we will construct a 1-particle filter at every timestep, as though each new observation is the first observation.

For planning, we'll do A* search in a discretized version of the environment. To do this, our first step will be to construct a "GridWorld" environment, by overlaying a cartesian grid onto the continuous environment, and noting which squares in the grid are occupied by walls.

In [18]:

```
includet("KidnappedRobot/astar_planning.jl") # Loads: `find_action_using_g

# Generate a gridworld version of this environment, in which A* planning i
planning_params = get_planning_params(walls, bounding_box);
```

Then, we'll define the controller.

In [19]:

```
includet("KidnappedRobot/handle_sticking.jl") # Loads: `handle_sticking`

@gen function _baseline_controller(controller_state, obs)
    (prev_pf_state, prev_action) = controller_state

    # Do a global localization scan, based on the current timestep's obse
    # Ignore any past inferences.
    pf_state = pf_init(choicemap((:obs, obs)))

    # Plan a trajectory to the goal in that grid contained in `planning_pa
    # Return the first action of that plan.
    action = find_action_using_grid_search(planning_params, currentpos(pf_

    # Sometimes, the details of the motion model and the A* planning
    # can cause the agent to "stick" on the walls.
    # This `handle_sticking` function checks if the agent has been trying
    # to perform the same action for multiple timesteps, but its belief s
    # has not changed; if so, it takes a random action orthogonal to the
    # action that is causing sticking.
    action = handle_sticking(prev_pf_state, prev_action, pf_state, action

    return (action, (pf_state, prev_action)) # (action, next_controller_s
end

baseline_controller = GenPOMDPs.Controller(
    _baseline_controller, # Controller state, observation ↝ action, next c
    (nothing, nothing)    # Initial controller state
)
```

```
GenPOMDPs.Controller(DynamicDSLFunction{Any}(Dict{Symbol, Any}(), Dict{Symb
ol, Any}(), Type[Any, Any], false, Union{Nothing, Some{Any}}[nothing, nothi
ng], var"##_baseline_controller#709", Bool[0, 0], false), (nothing, nothin
g))
```

Now that we have defined the controller, we can get a Generative Function over trajectories from rolling out the true world model, using this controller to choose actions.

The arguments to this generative function are `T`, the number of timesteps to roll out, and the POMDP parameters.

In [20]:

```
baseline_rollout_model = GenPOMDPs.RolloutModel(pomdp, baseline_controlle
```

```
GenPOMDPs.var"##StaticGenFunction__RolloutModel#778"(Dict{Symbol, Any}(), D
ict{Symbol, Any}())
```

Now let's simulate from this model.

First, we'll generate just the initial timestep.

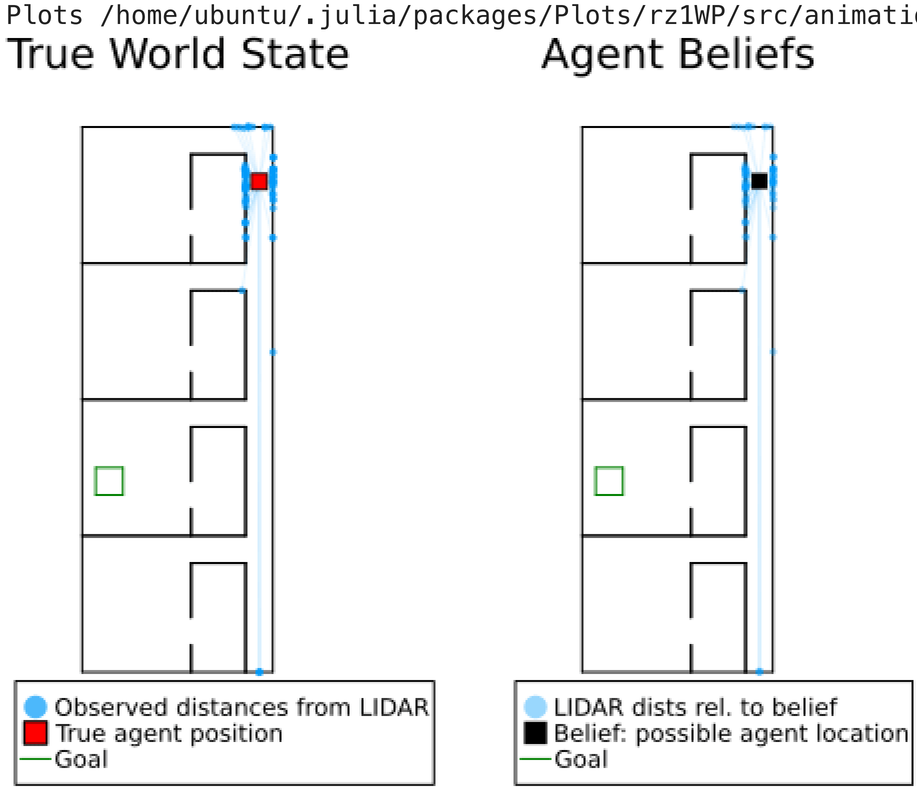
In [21]:

```
# Start the agent off in a hallway.
INITIAL_POS = [6.5, 4*5 - 2];
```

In [22]:

```
baseline_rollout_tr = Gen.generate(baseline_rollout_model,  # Generate a 
    (0, PARAMS),                                            # ...up to ti
    choicemap((GenPOMDPs.state_addr(0, :pos), INITIAL_POS)) # ...and cons
)[1];

trace_to_gif(baseline_rollout_tr; goalobj=goalobj) # Visualize the rollou
```

```
┌ Info: Saved animation to /tmp/jl_kdcr6dutcD.gif
└ @ Plots /home/ubuntu/.julia/packages/Plots/rz1WP/src/animation.jl:156
```

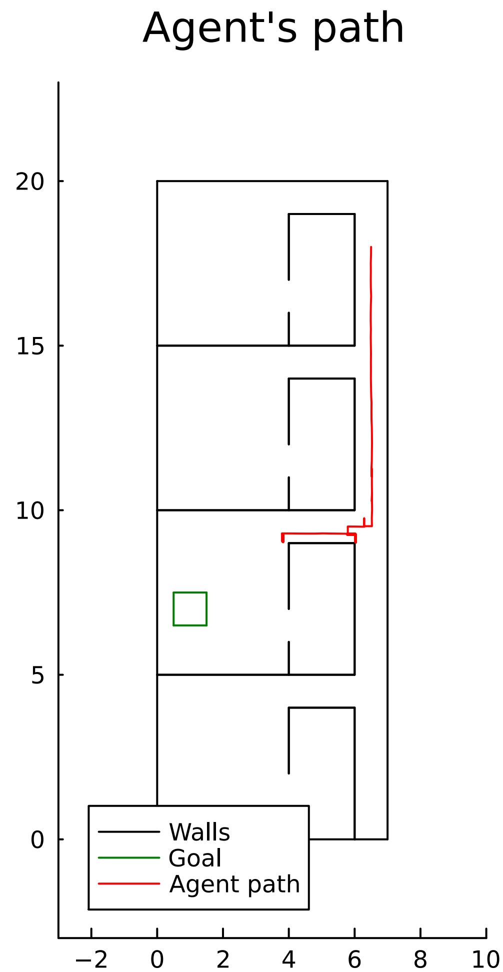


Now, we'll have the model simulate behavior for 100 timesteps.

In [23]:

```
baseline_rollout_tr, _ = Gen.update(baseline_rollout_tr,  # Update the re
    (100, PARAMS),                                        # ...updating i
    (UnknownChange(), NoChange()),                        # ...noting tha
                                                          #    (so Gen kn
```

```
    EmptyChoiceMap()                                              # Don't constra
                                                                  # (In principle
                                                                  # randomness, o
)

# Visualize the path the agent took.
trace_to_path_image(baseline_rollout_tr; goalobj=goalobj)
```

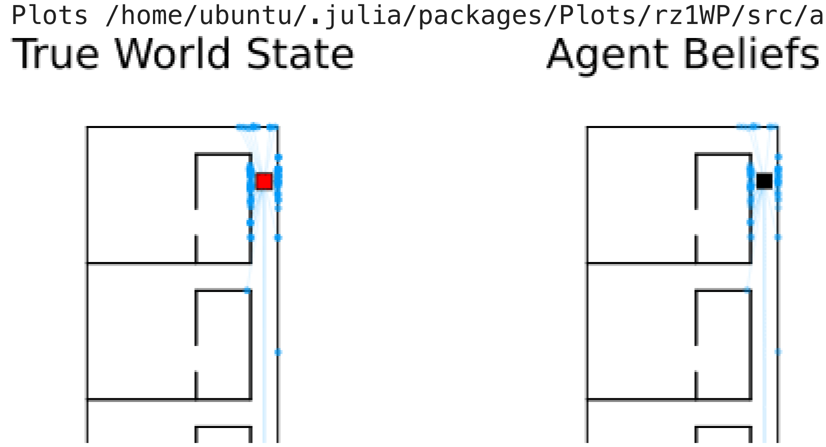


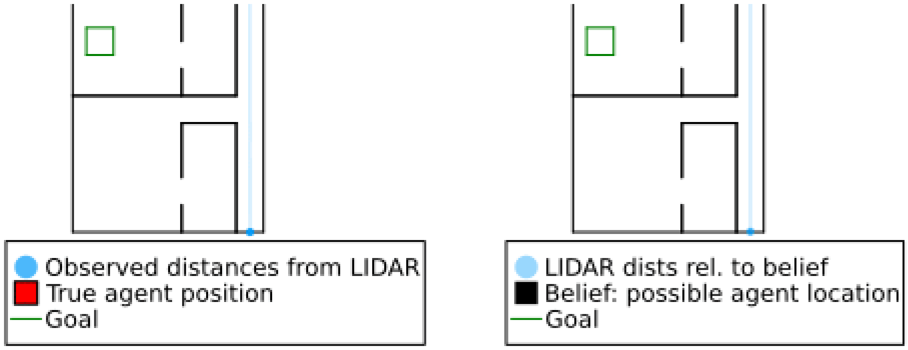
Here's a video of the agent's position and belief, over time.

In [24]:

```
trace_to_gif(baseline_rollout_tr; goalobj=goalobj, fps=10)
```

```
┌ Info: Saved animation to /tmp/jl_qb4lMBsPWp.gif
└ @ Plots /home/ubuntu/.julia/packages/Plots/rz1WP/src/animation.jl:156
```





The issue with this controller is that it does not remember its belief from the last timestep; it tries to fully relocalize at every step just using its current observations.

The result is that when the agent moves into a hallway into one of the rooms (which looks just like the hallways that lead into each other room), the agent gets confused about where it is. The issue is that the observed data from that timestep alone does not dis-ambiguate where the object is. At each timestep it thinks it is in the hallway toward the goal, it takes a step into the room; at each timestep it thinks it is in another hallway, it takes a step out from the room. As a result, it keeps going back and forth, and is stuck!

Baseline particle-filtering controller

To fix this, let's use a controller which uses a 1-particle particle filter, rather than re-localizing at each timestep.

In [25]:

```
@gen function _baseline_pf_controller(controller_state, obs)
    (prev_pf_state, prev_action) = controller_state

    if isnothing(prev_pf_state)
        pf_state = pf_init(choicemap((:obs, obs)))
    else
        pf_state = pf_update(prev_pf_state, prev_action, choicemap((:obs,
    end

    action = find_action_using_grid_search(planning_params, currentpos(pf_
    action = handle_sticking(prev_pf_state, prev_action, pf_state, action

    return (action, (pf_state, action)) # (action, next_controller_state)
end

baseline_pf_controller = GenPOMDPs.Controller(
    _baseline_pf_controller, # Controller state, observation ⇝ action, ne
    (nothing, nothing)       # Initial controller state
)
```

```
GenPOMDPs.Controller(DynamicDSLFunction{Any}(Dict{Symbol, Any}(), Dict{Symb
ol, Any}(), Type[Any, Any], false, Union{Nothing, Some{Any}}[nothing, nothi
ng], var"##_baseline_pf_controller#1208", Bool[0, 0], false), (nothing, not
hing))
```

In [26]:

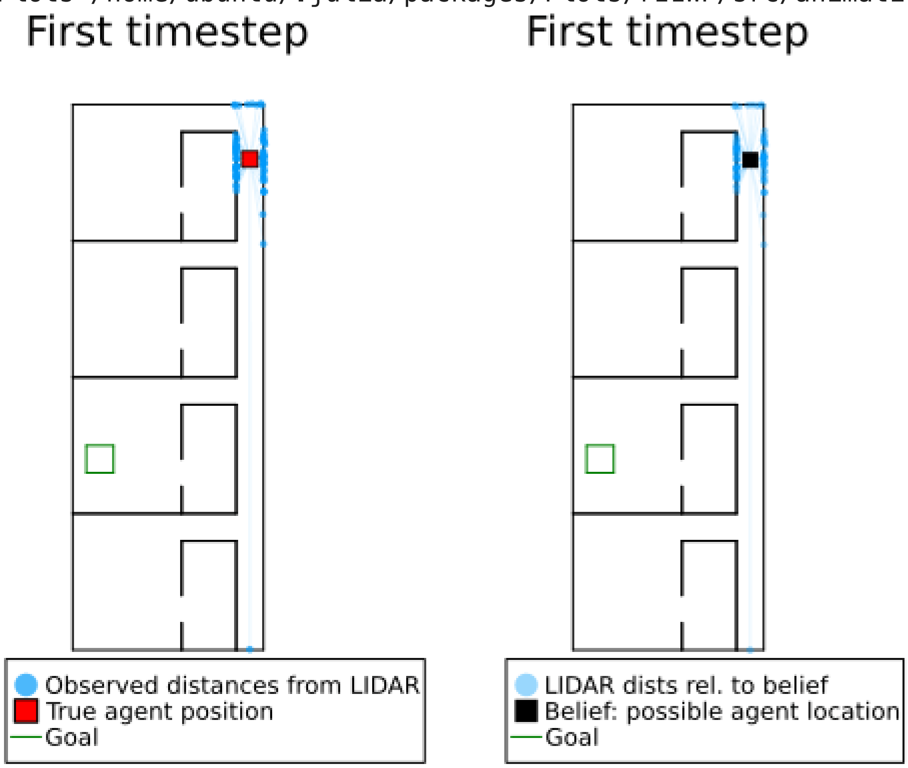
```
baseline_pf_rollout_model = GenPOMDPs.RolloutModel(pomdp, baseline_pf_con
```

```
GenPOMDPs.var"##StaticGenFunction__RolloutModel#1277"(Dict{Symbol, Any}(),
Dict{Symbol, Any}())
```

In [27]:

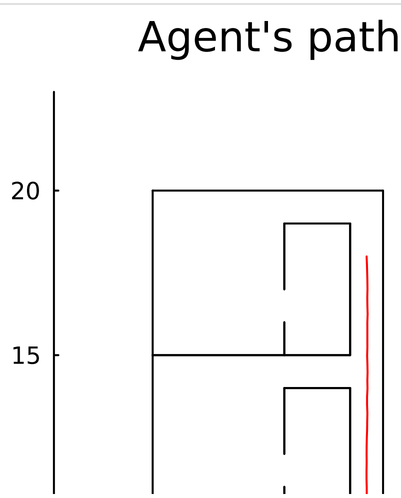
```
baseline_pf_rollout_tr = Gen.generate(baseline_pf_rollout_model, (0, PARAM
    choicemap((GenPOMDPs.state_addr(0, :pos), INITIAL_POS))
)[1]

trace_to_gif(baseline_pf_rollout_tr; goalobj=goalobj, title="First timeste
```

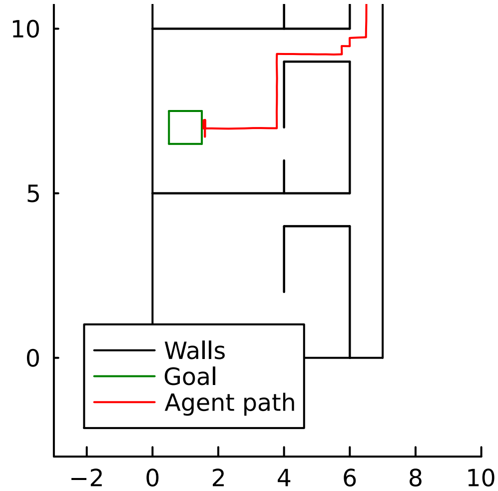
```
┌ Info: Saved animation to /tmp/jl_STW54HgPQ5.gif
└ @ Plots /home/ubuntu/.julia/packages/Plots/rz1WP/src/animation.jl:156
```



In [28]:

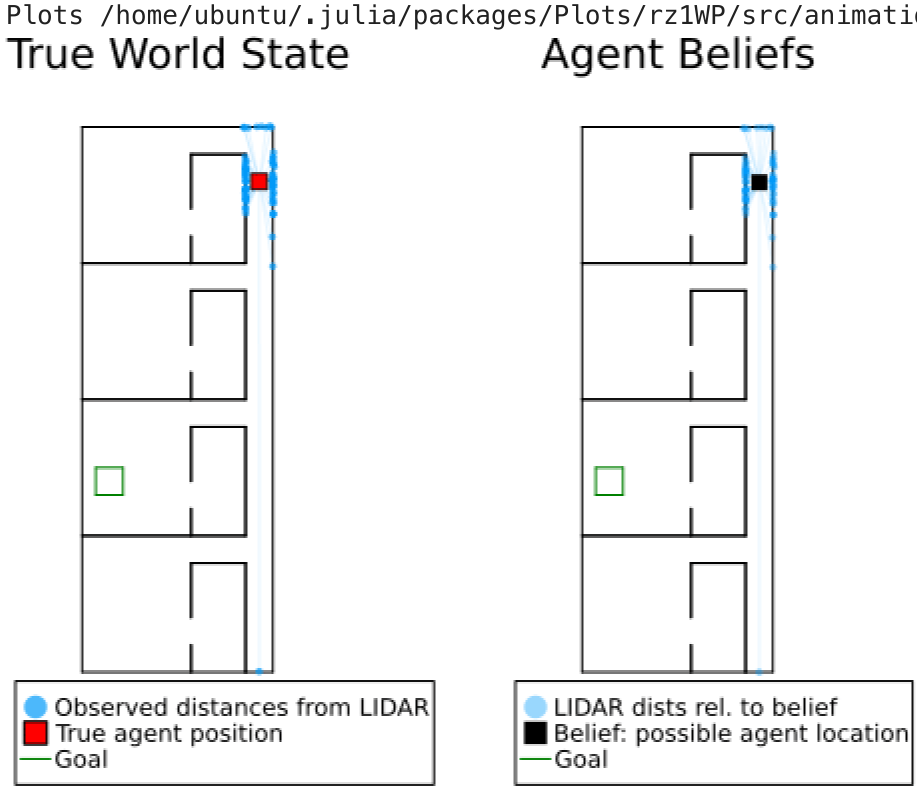
```
baseline_pf_rollout_tr, _ = Gen.update(baseline_pf_rollout_tr, (100, PARAM

trace_to_path_image(baseline_pf_rollout_tr; goalobj=goalobj)
```





In [29]:
```
trace_to_gif(baseline_pf_rollout_tr; goalobj=goalobj, fps=10)
```

```
┌ Info: Saved animation to /tmp/jl_etGps5Klq7.gif
└ @ Plots /home/ubuntu/.julia/packages/Plots/rz1WP/src/animation.jl:156
```



"Kidnap the robot"

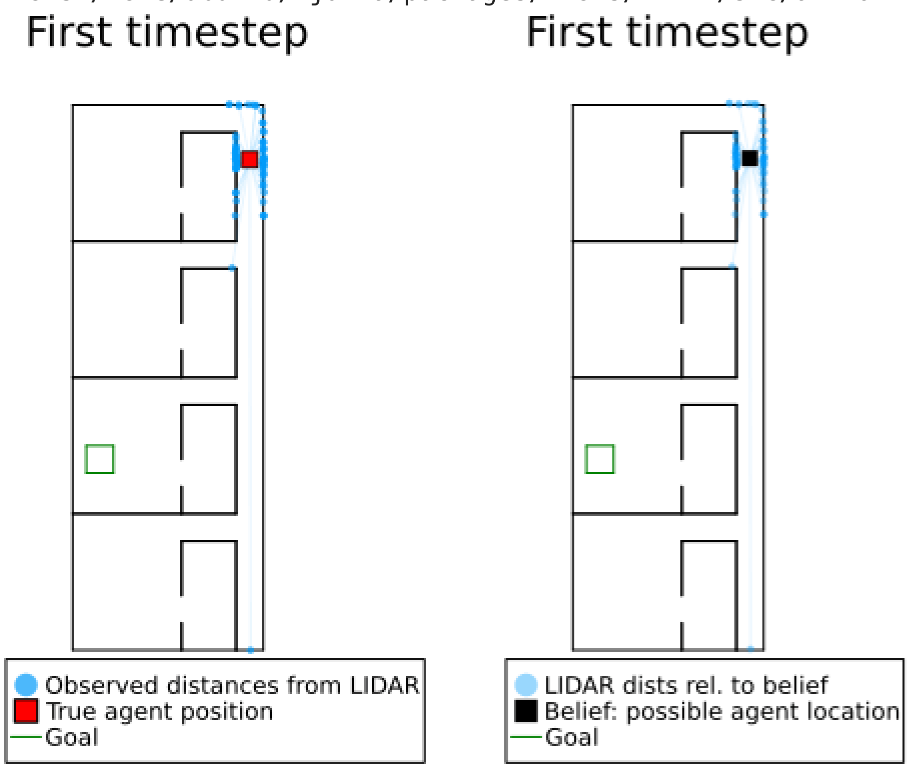
Now, we'll give the robot the same task: navigate to the green square.

But, after 40 timesteps, we'll imagine the robot comes across a well-meaning hotel employee who sees the robot, and doesn't realize we roboticists have it doing an important task for us. The employee turns off the employee and brings it to a storage closet in one of the unoccupied hotel rooms. Eventually, we notice this issue, and we turn the robot back on. The robot controller then tries to pick up where it left off, and find a path to the goal. But with the basic PF controller we defined above, the robot cannot re-localize after it is moved to a new place!

In [30]:

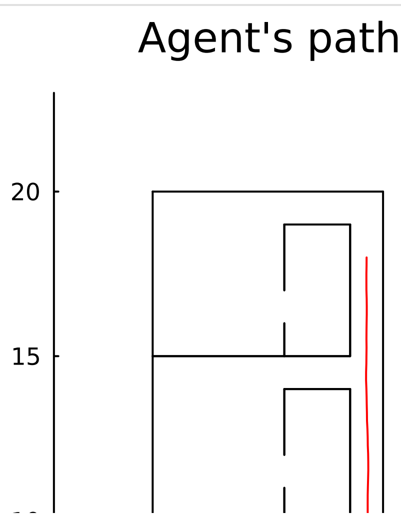
```
baseline_rollout_tr_KR = Gen.generate(baseline_pf_rollout_model, (0, PARAM
    choicemap((GenPOMDPs.state_addr(0, :pos), INITIAL_POS))
)[1]

trace_to_gif(baseline_rollout_tr_KR; goalobj=goalobj, title="First timest
```

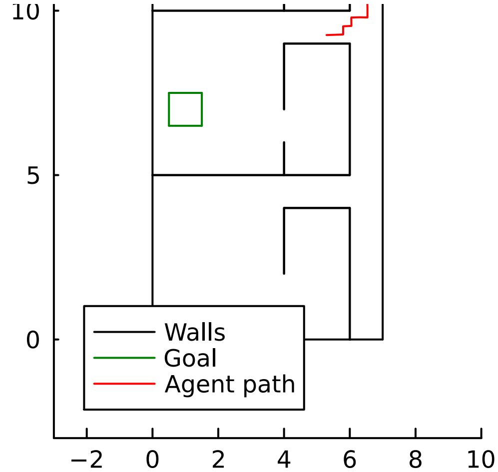
```
┌ Info: Saved animation to /tmp/jl_AVcZWFY9kF.gif
└ @ Plots /home/ubuntu/.julia/packages/Plots/rz1WP/src/animation.jl:156
```



In [31]:

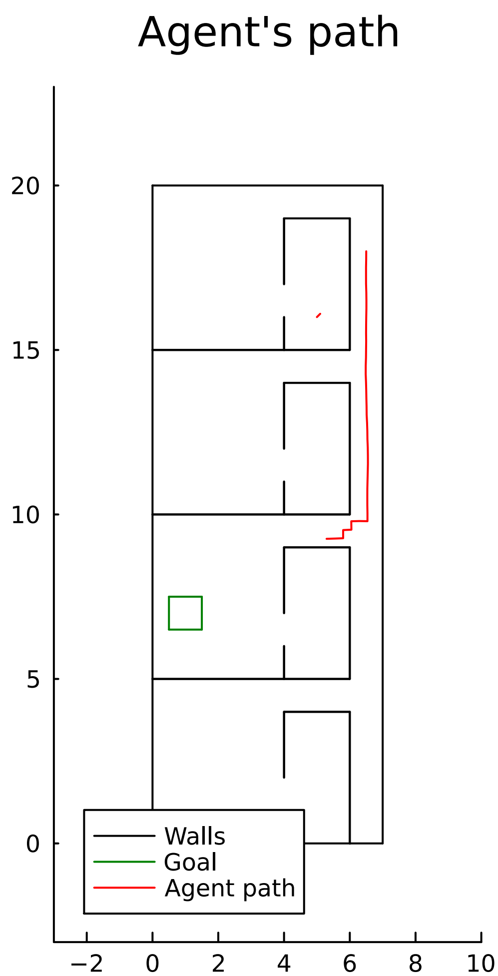
```
# Extend rollout to 40 steps...
baseline_rollout_tr_KR, _ = Gen.update(baseline_rollout_tr_KR, (40, PARAMS

trace_to_path_image(baseline_rollout_tr_KR; goalobj=goalobj)
```





In [32]:

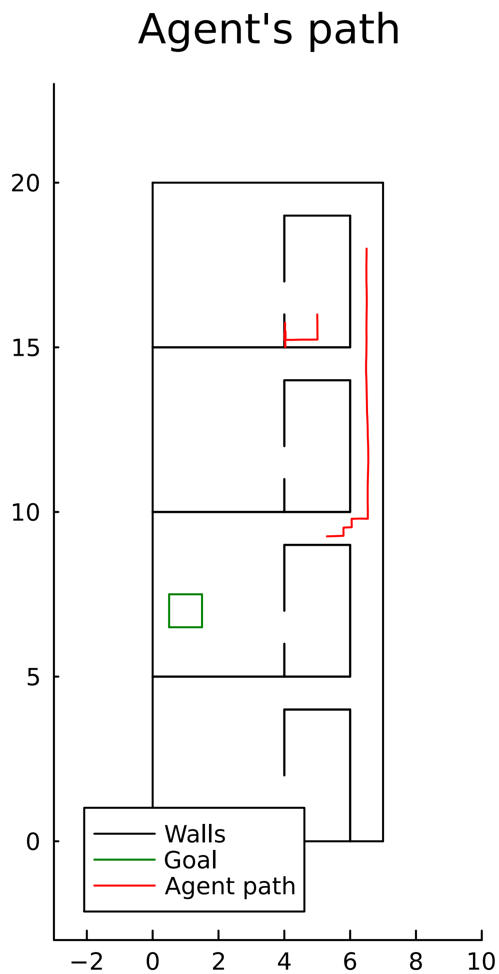
```
## Kidnap the robot!
baseline_rollout_tr_KR, _ = Gen.update(baseline_rollout_tr_KR, (41, PARAMS
    choicemap((GenPOMDPs.state_addr(41, :is_kidnapped), true), (GenPOMDPs
);

trace_to_path_image(baseline_rollout_tr_KR; goalobj=goalobj, kidnapped_at=
```



In [33]:

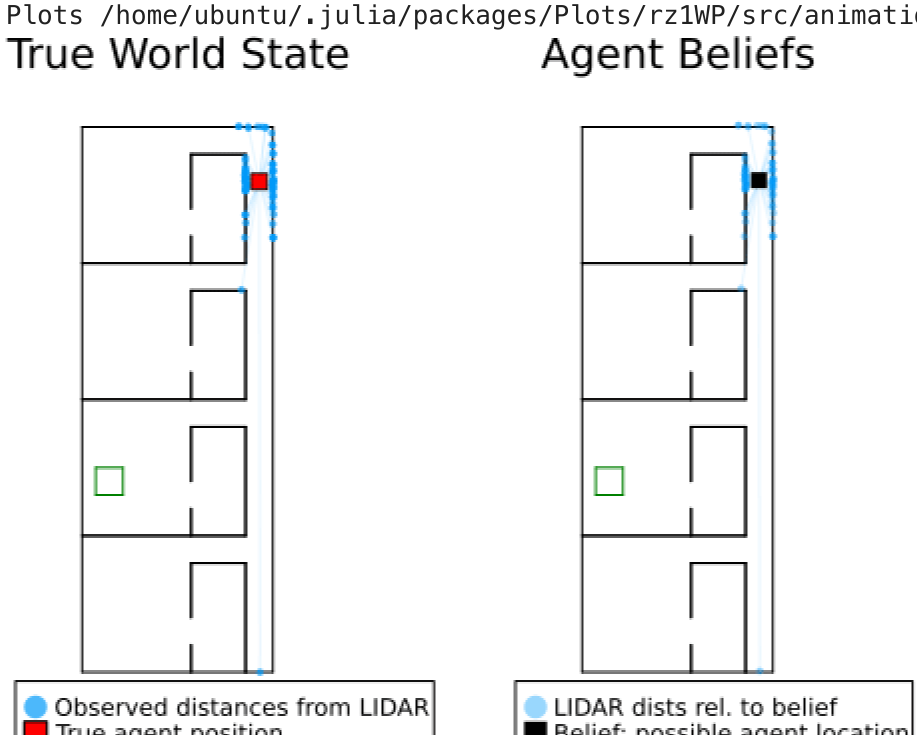
```
# Roll out the trace another 100 steps, after the robot is re-activated.
baseline_rollout_tr_KR, _ = Gen.update(baseline_rollout_tr_KR, (140, PARAM
trace_to_path_image(baseline_rollout_tr_KR; goalobj=goalobj, kidnapped_at=
```

```
trace_to_path_image(baseline_rollout_tr_KR; goalobj=goalobj, kidnapped_at=
```



In [34]:
```
trace_to_gif(baseline_rollout_tr_KR; goalobj=goalobj, fps=10, kidnapped_at
```

```
┌ Info: Saved animation to /tmp/jl_obmH4JotW9.gif
└ @ Plots /home/ubuntu/.julia/packages/Plots/rz1WP/src/animation.jl:156
```



The particle filter can't handle the robot kidnapping.

One solution would be to have the agent do expensive MCMC rejuvenation at every step, to check if it might have been moved elsewhere.

But we don't need to take on this computational cost. Instead, we can have the controller make intelligent decisions about on which steps we should spend more computation to re-localize globally.

Below, we'll implement a simple version of this, which resets the particle filter whenever the marginal likelihood estimate from the particle filter (the average particle weight -- and in this case the only particle weight) falls too low.

Robust controller: particle filtering + reset particle filter when the likelihood falls too low

In [35]:

```
# @dist labeled_categorical(labels, probs) = labels[categorical(probs)]
```

In [36]:

```
@gen function _controller(controller_state, obs)
    prev_pf_state, prev_action = controller_state

    # Create 1-particle belief state
    if isnothing(prev_action) # First timestep
        pf_state = pf_init(choicemap((:obs, obs)))
    else
        # Log marginal likelihood estimate from the particle filter
        prev_lml_est = GenParticleFilters.get_lml_est(prev_pf_state)

        # Try updating the PF belief state
        pf_state = pf_update(prev_pf_state, prev_action, choicemap((:obs,
        new_lml_est = GenParticleFilters.get_lml_est(pf_state)

        # We will define and tune this check below
        if incremental_log_likelihood_est_is_too_low(new_lml_est - prev_l
            # Reset the particle filter!
            # The new pf_state will be over trajectories of length 1.
            pf_state = pf_init(choicemap((:obs, obs)))
        end
    end

    # Choose action
    action = find_action_using_grid_search(planning_params, currentpos(pf_
    action = handle_sticking(prev_pf_state, prev_action, pf_state, action

    # Choose the action to take.
    # is_viable_onehot = [a in viable_actions ? 1. : 0 for a in [:left, :
    # action_probs = is_viable_onehot / sum(is_viable_onehot)
    # action ~ labeled_categorical([:left, :right, :up, :down, :stay], ac
```

```
    return (action, (pf_state, action)) # (action, next_controller_state)
end

controller = GenPOMDPs.Controller(
    _controller,                        # Controller state, observation ⇝ ac
    (nothing, nothing)                  # Initial controller state
)
```

```
GenPOMDPs.Controller(DynamicDSLFunction{Any}(Dict{Symbol, Any}(), Dict{Symb
ol, Any}(), Type[Any, Any], false, Union{Nothing, Some{Any}}[nothing, nothi
ng], var"##_controller#1901", Bool[0, 0], false), (nothing, nothing))
```

Tuning the particle filter log marginal likelihood threshold.

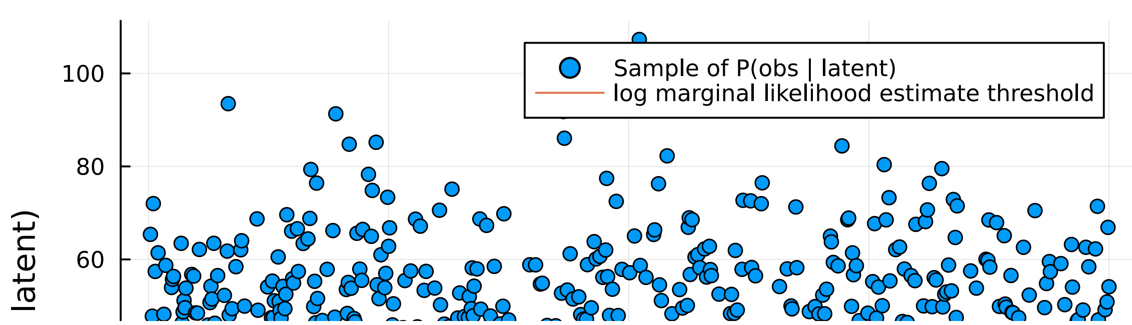
Now, we need to define `incremental_log_likelihood_est_is_too_low` , which we used in the controller above. This will be a simple threshold on the estimated value of `P(obs | latent_{t-1})` from the particle filter.

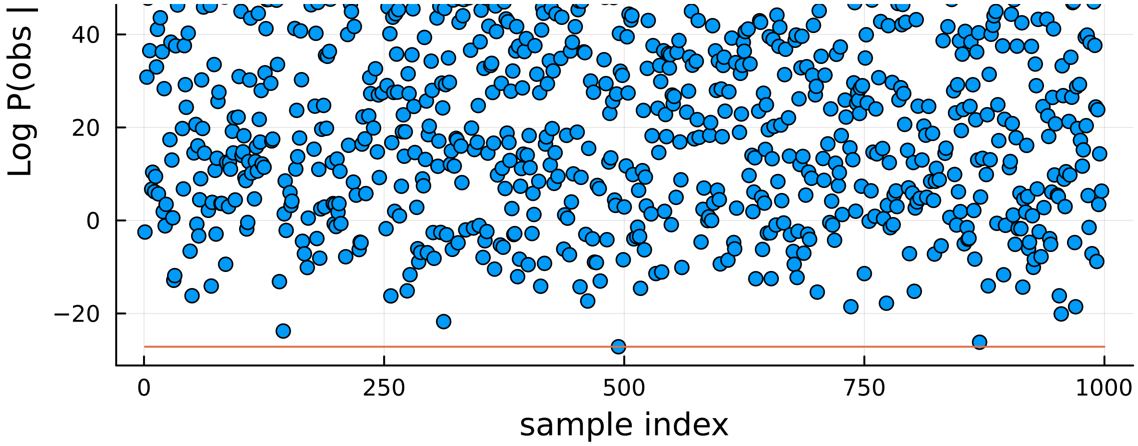
Note that the expected value of `P(obs)` is `P(obs | latent)` . Based on this observation, we will set our threshold by generating 1000 random (latent, obs) pairs from the model, and setting the threshold to be the minimum value of `P(obs | latent)` which arises.

This is currently just a heuristic I quickly thought of to set this threshold, which I have observed works well in this environment. One of my research TODOs is to think more carefully about about whether this method of tuning the threshold can be expected to work well across environments.

In [37]:

```
logpy_values = []
for _=1:1000
    state = uniform_agent_pos(MENTAL_MODEL_PARAMS)
    obs_tr = simulate(sensor_model, (state, MENTAL_MODEL_PARAMS))
    push!(logpy_values, get_score(obs_tr))
end

logpy_threshold = minimum(logpy_values)

function incremental_log_likelihood_est_is_too_low(incremental_logpy_estim
    return incremental_logpy_estimate < logpy_threshold
end

scatter(1:1000, logpy_values, ylabel="Log P(obs | latent)", xlabel="sample
plot!(1:1000, [logpy_threshold for _=1:1000], label="log marginal likelih
```





Simulating the robust controller, in an environment with no robot kidnapping.

In [38]:

```
rollout_model = GenPOMDPs.RolloutModel(pomdp, controller)
```

```
GenPOMDPs.var"##StaticGenFunction__RolloutModel#1971"(Dict{Symbol, Any}(),
Dict{Symbol, Any}())
```

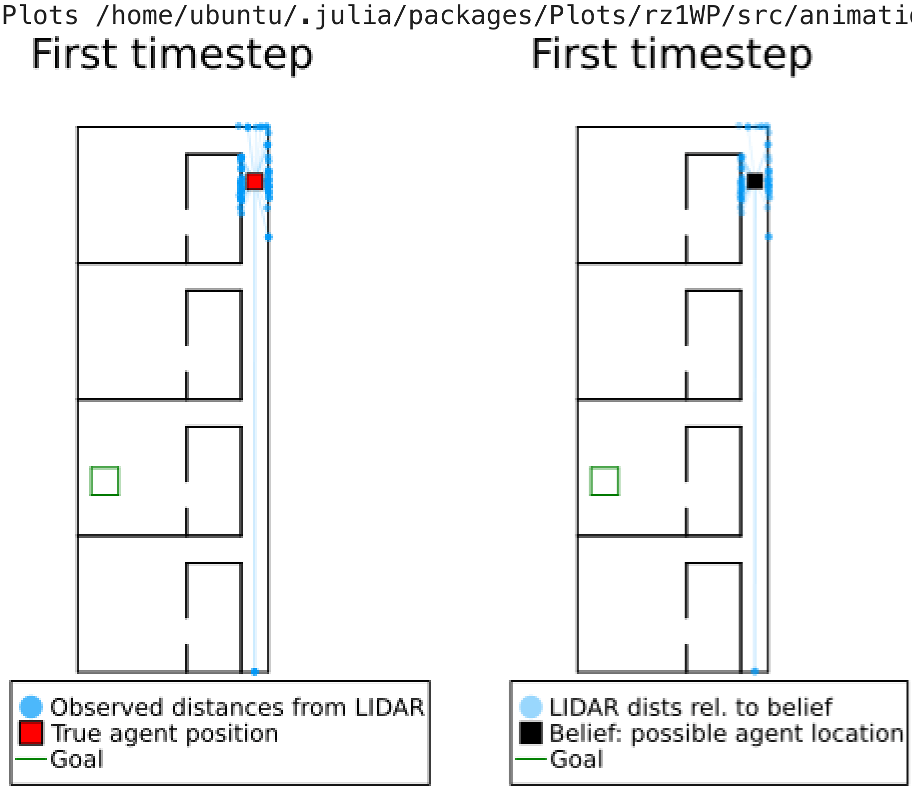
In [40]:

```
```

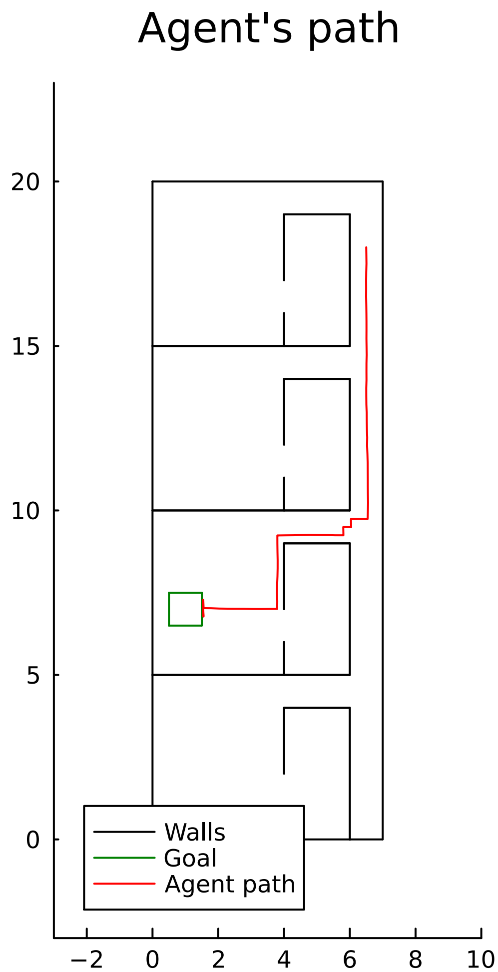
In [43]:

```
rollout_tr = Gen.generate(rollout_model, (0, PARAMS),
    choicemap((GenPOMDPs.state_addr(0, :pos), INITIAL_POS))
)[1]

trace_to_gif(rollout_tr; goalobj=goalobj, title="First timestep")
```

```
┌ Info: Saved animation to /tmp/jl_iNgF4vOX21.gif
└ @ Plots /home/ubuntu/.julia/packages/Plots/rz1WP/src/animation.jl:156
```

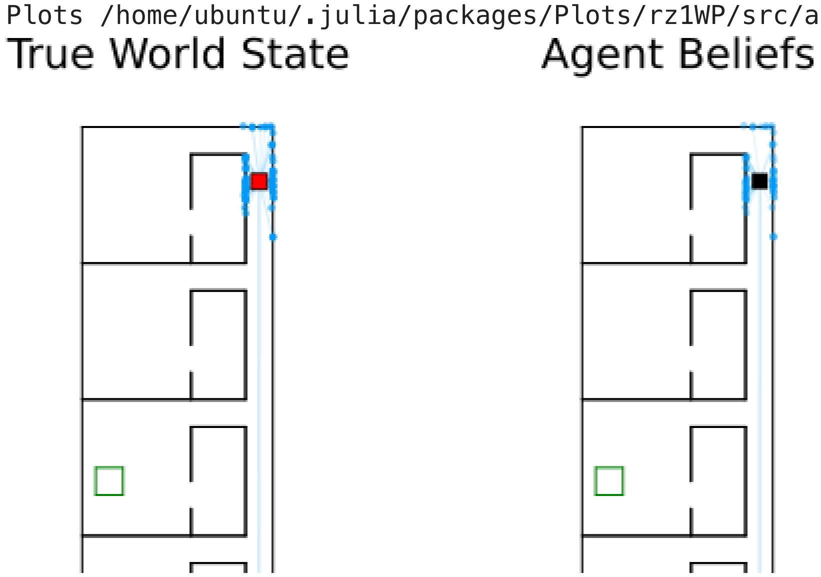


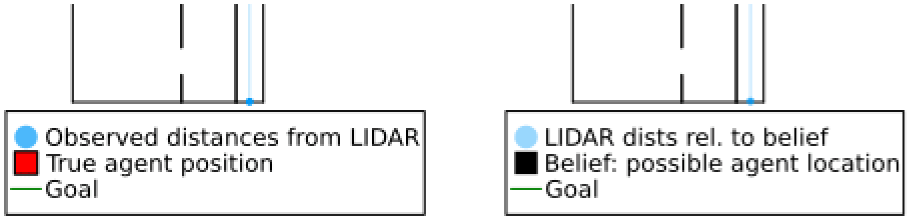
In [44]:
```
# Extend rollout...
rollout_tr, _ = Gen.update(rollout_tr, (80, PARAMS), (UnknownChange(), No(

trace_to_path_image(rollout_tr; goalobj=goalobj)
```



In [45]:
```
trace_to_gif(rollout_tr; goalobj=goalobj, fps=10)
```

```
┌ Info: Saved animation to /tmp/jl_8BIlKn8lcP.gif
└ @ Plots /home/ubuntu/.julia/packages/Plots/rz1WP/src/animation.jl:156
```



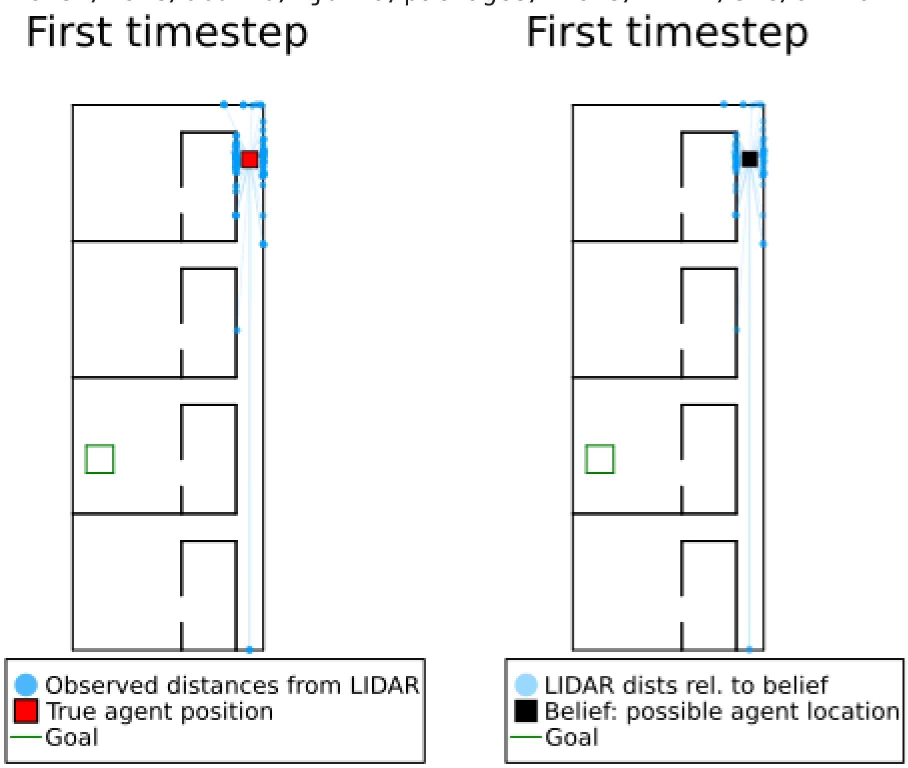


Kidnapped robot with the robust controller

In [46]:

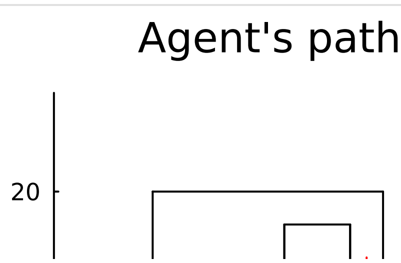
```
robust_rollout_tr_KR = Gen.generate(rollout_model, (0, PARAMS),
    choicemap((GenPOMDPs.state_addr(0, :pos), INITIAL_POS))
)[1]

trace_to_gif(robust_rollout_tr_KR; goalobj=goalobj, title="First timestep'
```

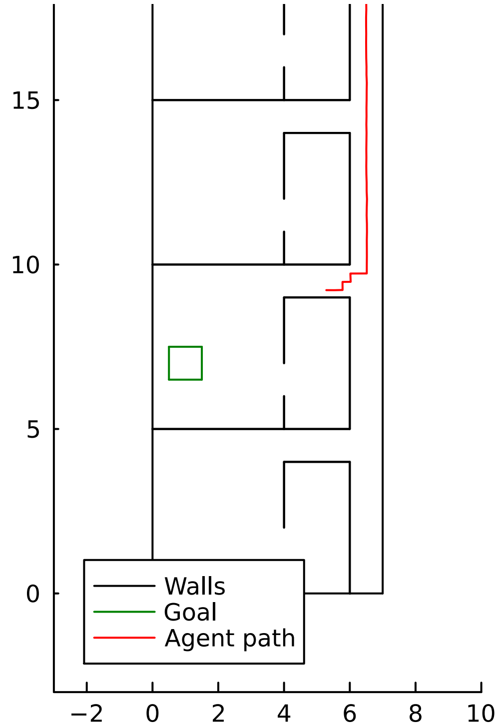
```
┌ Info: Saved animation to /tmp/jl_wfaZwnYxep.gif
└ @ Plots /home/ubuntu/.julia/packages/Plots/rz1WP/src/animation.jl:156
```



In [47]:

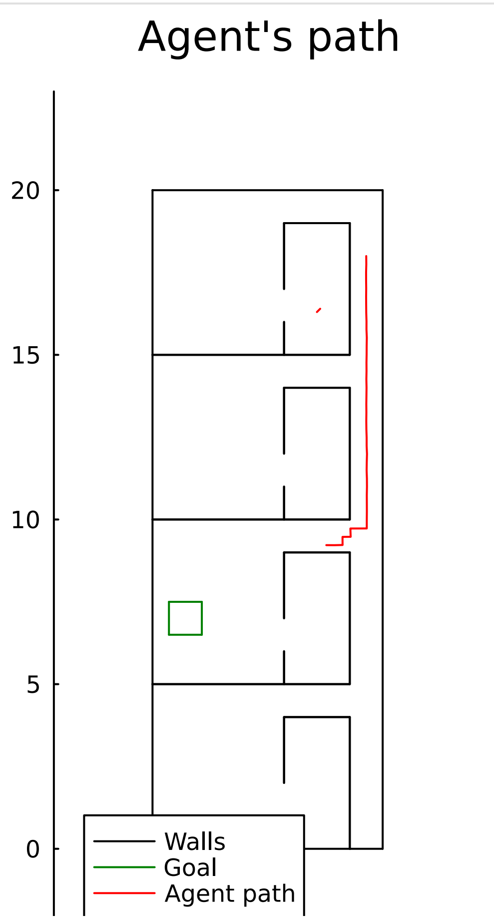
```
# Extend rollout to 40 steps...
robust_rollout_tr_KR, _ = Gen.update(robust_rollout_tr_KR, (40, PARAMS),

trace_to_path_image(robust_rollout_tr_KR; goalobj=goalobj)
```

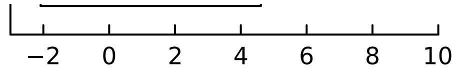




In [48]:

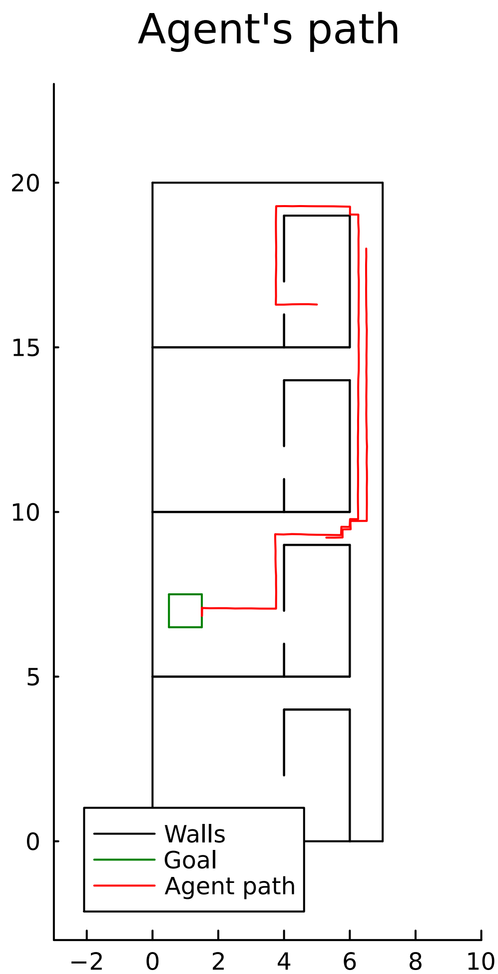
```
## Kidnap the robot!
robust_rollout_tr_KR, _ = Gen.update(robust_rollout_tr_KR, (41, PARAMS),
    choicemap((GenPOMDPs.state_addr(41, :is_kidnapped), true), (GenPOMDPs
);

trace_to_path_image(robust_rollout_tr_KR; goalobj=goalobj, kidnapped_at=[4
```





In [49]:

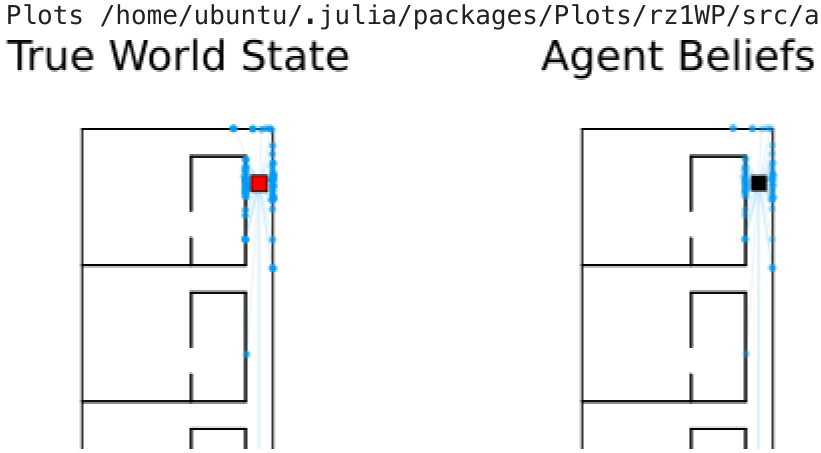
```
# Roll out the trace another 100 steps
robust_rollout_tr_KR, _ = Gen.update(robust_rollout_tr_KR, (140, PARAMS),
trace_to_path_image(robust_rollout_tr_KR; goalobj=goalobj, kidnapped_at=[4
```

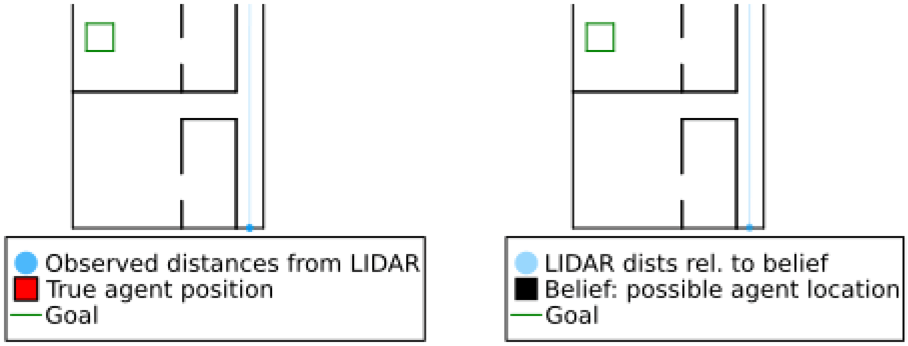


In [50]:

```
trace_to_gif(robust_rollout_tr_KR; goalobj=goalobj, fps=10, kidnapped_at=
```

```
┌ Info: Saved animation to /home/ubuntu/Developer/tasks2D/notebooks_clean/k
idnapping_recovery.gif.gif
└ @ Plots /home/ubuntu/.julia/packages/Plots/rz1WP/src/animation.jl:156
```



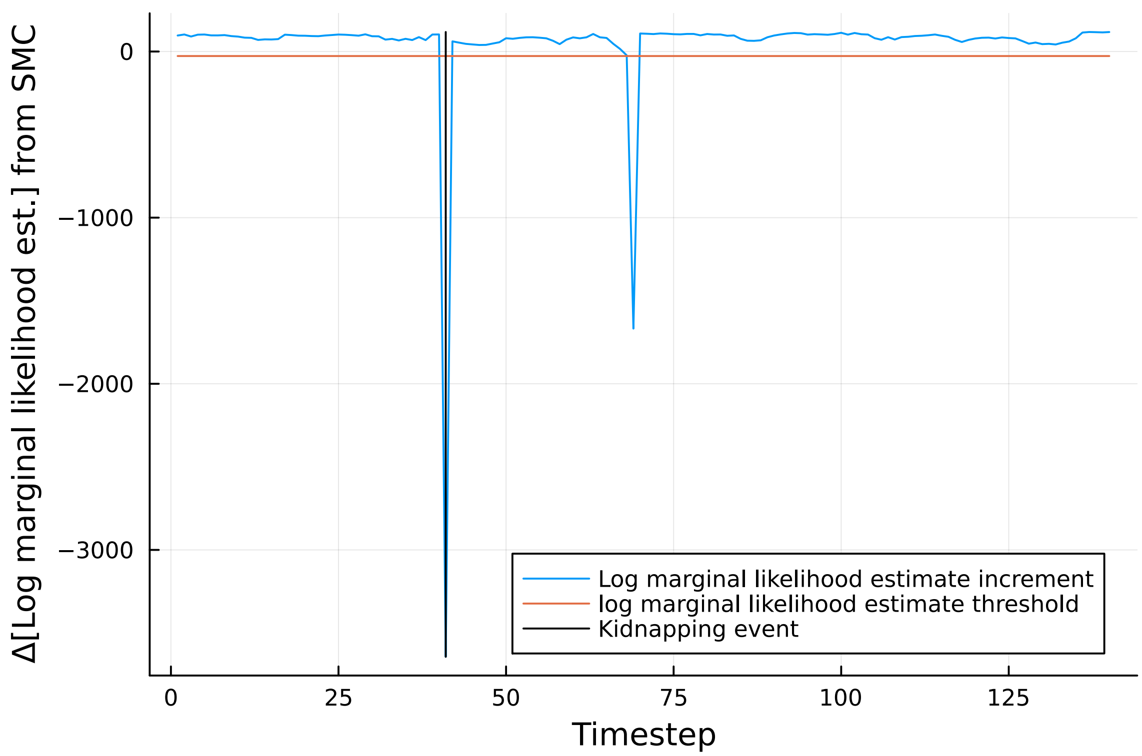


Plotting the log marginal likelihood estimates from each timestep:

In [51]:

```
lml_ests = [GenParticleFilters.get_lml_est(pfst)
    for (pfst, _) in GenPOMDPs.controllerstate_sequence(robust_rollout_tr
]
println(length(lml_ests))
deltas = [x - y for (x, y) in zip(lml_ests[2:end], lml_ests[1:end-1])]
p = plot(1:length(deltas), deltas, label="Log marginal likelihood estimat
plot!(1:length(deltas), [logpy_threshold for _ in 1:length(deltas)], labe
plot!([41, 41], [minimum(deltas)-.1, maximum(deltas)+.1], label="Kidnappi
```

141



In [52]:

Plotting the runtime and effectiveness of each controller

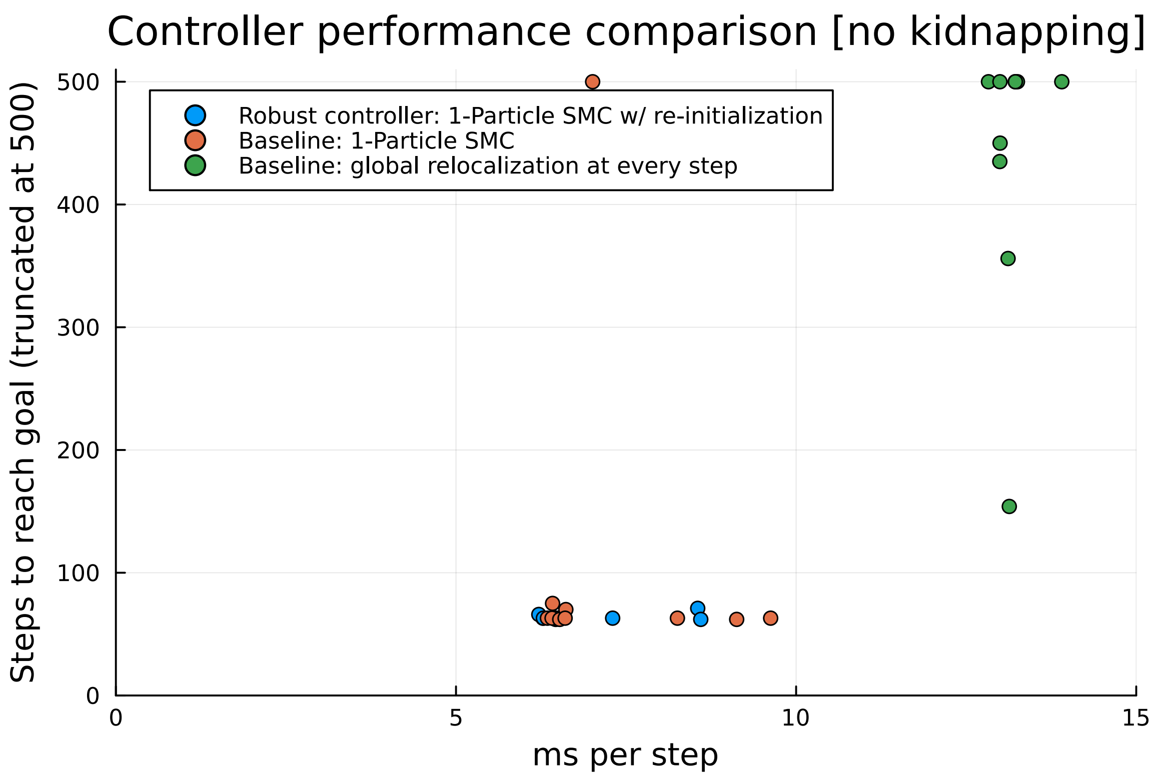
In [53]:
```
includet("KidnappedRobot/measure_performance.jl") # Loads `take_measuremen
```

In [54]:
```
robust_costs_runtimes = [take_measurement(rollout_model) for _=1:10];
```

In [55]:
```
baseline1_costs_runtimes = [take_measurement(baseline_rollout_model) for _
```

In [56]:
```
baselinepf_costs_runtimes = [take_measurement(baseline_pf_rollout_model) f
```

In [57]:
```
plot(;
    title="Controller performance comparison [no kidnapping]",
    ylabel="Steps to reach goal (truncated at 500)",
    xlabel="ms per step",
    ylims=(0, 510),
    xlims=(0,15)
)
scatter!(map(x->x[2]*1000, robust_costs_runtimes), map(x->x[1], robust_cos
scatter!(map(x->x[2]*1000, baselinepf_costs_runtimes), map(x->x[1], basel
scatter!(map(x->x[2]*1000, baseline1_costs_runtimes), map(x->x[1], baselin
```



In [58]:
```
robust_costs_runtimes_KR = [take_measurement_KR(rollout_model) for _=1:10
```

In [59]:
```
baseline1_costs_runtimes_KR = [take_measurement_KR(baseline_rollout_model
```

In [60]:
```
baselinepf_costs_runtimes_KR = [take_measurement_KR(baseline_pf_rollout_mo
```

In [61]:
```
plot(;
    title="Controller performance comparison [with kidnapping]"
```