FEBOL Documentation

Release 0.1

Louis Dressel

Nov 22, 2018

Contents:

CHAPTER 1

Basic Types

1.1 Search Domain

```
# puts target at (tx, ty)
m = SearchDomain(side_length, tx, ty)

# puts target at random location within the square domain
m = SearchDomain(side_length)
```

1.2 LocTuple

```
const LocTuple = NTuple{2, Float64}
```

1.3 Pose

```
const Pose = NTuple{3, Float64}
```

1.4 Action

```
const Action = NTuple{3, Float64}
```

CHAPTER 2

Vehicle

Each instance of `Vehicle` has the following fields:

```
x::Float64
y::Float64
heading::Float64    # east of north (degrees)
max_step::Float64   # max distance vehicle can go per unit time (meters)
sensor::Sensor
```

There are several constructors. Below is the default:

```
v = Vehicle(x::Real, y::Real, h::Real, ms::Real, s::Sensor)
```

If you just give it a starting location, `heading` is set to 0, `max_step` is set to 2.0, and the `sensor` is defaulted to `BearingOnly(10.0)` (a bearing-only sensor with noise std deviation of 10 deg).

```
v = Vehicle(x::Real, y::Real)
```

Alternatively, you can pass the sensor in as well, with the omitted variables as above:

```
v = Vehicle(x::Real, y::Real, s::Sensor)
```

CHAPTER 3

Sensors

The abstract `Sensor` type describes the sensing model of the vehicle. Originally, the only sensor type was bearing only, but this has been expanded to consider other sensing modalities.

3.1 BearingOnly

```
BearingOnly(noise_sigma)
```

3.2 DirOmni

The `DirOmni` sensor combines a directional antenna with an omni-directional antenna.

3.3 FOV

The `FOV` sensor is a "field-of-view" sensor. The observed value 1 suggests the source is in the vehicle's field of view, and 0 suggests the source is not.

```
region_probs = [(60.0,0.9), (120.0, 0.5), (180.0,0.1)]
sensor = FOV(region_probs)
v = Vehicle(50,50, sensor)
```

3.4 Custom Sensors

You can make your own sensors.

```
NewSensor <: Sensor
```

You must implement the `observe` function, which returns an observation (of type `Float64`).

```
observe(tx::LocTuple, s::NewSensor, p::Pose)
```

If you want the particle filter to work, you need to define an observation model.

```
O(s::NewSensor, theta::LocTuple, p::Pose, o::Float64)
```

If you want the discrete filter to work, you need to define a discretized version, and a function that converts an observation (Float64) into a discretized version (Int)

```
obs2bin(o::Float64, s::NewSensor)    # returns an int
O(s::NewSensor, theta::LocTuple, p::Pose, o::Int)
```

CHAPTER 4

Filters

A filter is something that maintains a belief over the search space and updates it given new observations and vehicle locations.

Note that each filter maintains a belief, which is a questionable design decision. In reality, a belief is something separate, fed into a filter to be updated. However, the belief representation (discrete, Gaussian, etc) depends heavily on the filtering being applied. In short, it just seems easier to maintain a single filter type rather than worry about a separate belief.

Note that each filter has its own sensor, even though the vehicle also has a sensor. The filtering updates use the filter's sensor, and the observations actually received come from the vehicle's sensor. This distinction allows you to test the effect of unmodeled sensor noise. In this case, the vehicle's sensor might have noise that is not accounted for in the filter's model, which can affect localization.

4.1 Discrete Filter

The discrete filter type, `DF`, has the following fields

```
b::Matrix{Float64}        # the actual discrete belief
n::Int64                  # number of cells per side
cell_size::Float64        # width of each cell, in meters
sensor<:Sensor            # sensor model used in filtering
obs_list                  # list of observations
```

The matrix `b` is the probability distribution over possible target locations. The weight in a cell is the probability that the target is in that cell.

The `obs_list` field exists for greedy control based on mutual information. Computing mutual information requires integrating over possible observations. However, if you are using a different controller you can ignore this field.

The constructor for a discrete filter is

```
DF(m::SearchDomain, n::Int, s::Sensor, obs_list=0:0)
```

where `n` is the number of cells per side.

4.2 Particle Filter

The particle filter is based on ParticleFilters.jl. Its constructor is

```
PF(m::Model, n::Int, obs_list)
```

The `Model` type contains information that is used in the particle filter update. The type and constructors are

```
struct Model{V <: Vehicle, S <: Sensor, M <: MotionModel}
    x::V
    sensor::S
    motion_model::M
end
Model(x::Vehicle) = Model(x, x.sensor)
Model(x::Vehicle, s::Sensor) = Model(x, s, NoMotion())
```

4.3 Extended Kalman Fiter

```
EKF(m::SearchDomain)
```

4.4 Unscented Kalman Fiter

```
UKF(m::SearchDomain)
```

4.5 Gaussian Fiter

The `GaussianFilter` abstract type is a child of `AbstractFilter` and a parent of `EKF` and `UKF`. I've thought about calling this `KalmanFilter` instead, but that could be ambiguous—someone could think this refers to a specific KF, rather than an abstract type.

The `GaussianFilter` abstract type covers utilities that both `EKF` and `UKF` use. The most important of these is the `Initializer` abstract type. Each `EKF` and `UKF` instance contains an `Initializer` subtype that determines how the filter estimate should be initialized.

The default initializer is a `NaiveInitializer` sets the estimate to be the center of the search domain and uses a large initial covariance.

Another initializer is the `LSInitializer`, or least squares initializer. After taking `min_obs_num` observations, this initializer sets the mean to the point in the search domain yielding the smallest sum of least square differences between observed and expected observations. The code below shows how to initialize an instance of `LSInitializer` and modify some of its important fields:

```
lsi = LSInitializer(m::SearchDomain)
lsi.Sigma = 1e3*eye(2)
lsi.min_obs_num = 5
```

4.6 Custom Filters

The code below is a template for creating your own filter type. You must extend the `AbstractFilter` type and implement the following functions.

```
type CustomFilter <: AbstractFilter
end

function update!(f::CustomFilter, p::Pose, o::Float64)
    # update the belief in the filter.
end

function centroid(f::CustomFilter)
    # return the centroid of the filter's belief
end

function entropy(f::CustomFilter)
    # return the entropy of the filter's belief
end

function reset!(f::CustomFilter)
    # reset the filter to a uniform prior
end
```

CHAPTER 5

Policies

5.1 RandomPolicy

A `RandomPolicy` simply moves the vehicle in a random direction.

```
RandomPolicy()
```

5.2 GreedyPolicy

A `GreedyPolicy` moves the agent in the direction that minimizes the expected entropy after moving.

```
GreedyPolicy(x::Vehicle, n::Int)
```

The integer `n` denotes how many actions should be considered. If `n=6`, then the agent considers the expected entropy given 6 different directions, spaced an even 60 degrees apart.

5.3 CirclePolicy

A `CirclePolicy` moves the agent perpendicularly to the last recorded bearing measurement, which ends up drawing a circle around the source. The constructor is as follows:

```
CirclePolicy()
```

The `CirclePolicy` implicitly assumes that the sensor is of `BearingOnly` type.

5.4 Custom Policy

You can create your own policies by extending the abstract `Policy` class and implementing the `action` function. Below is an example. Remember that to extend `FEBOL`'s `action` function, you must import it instead of just relying on `using`:

```
using FEBOL
import FEBOL.action

type CustomPolicy <: Policy
end

function action(m::SearchDomain, x::Vehicle, o::Float64, f::AbstractFilter, p::CustomPolicy)
    # your policy code
    # must return action (2-tuple of Float64s)
end
```

Feel free to take advantage of the `normalize` function to ensure your action's norm is equal to the maximum distance the vehicle can take per time step:

```
normalize(a::Action, x::Vehicle)
```

CHAPTER 6

Simulations

6.1 Quick Simulations

If you've just implemented a sensor, filter, or policy, you might want to run it through a quick simulation to make sure everything works. You can simply call

```
simulate(m, x, f, p, n_steps=10)
```

where `m` is a SearchDomain, `x` is a Vehicle, `f` is a filter, and `p` is a policy. If everything works, no error will be thrown.

6.2 Simulations with SimUnit

To specify costs and termination conditions, use the SimUnit type.

```
simulate(m::SearchDomain, su::SimUnit)
```

This returns the total cost of the simulated run (a float).

6.3 Batch Simulations

To evaluate a SimUnit over the course of various simulations, you can provide a number of simulations, `n_sims`, to `simulate`:

```
simulate(m::SearchDomain, su::SimUnit, n_sims::Int)
```

At the beginning of each simulation, the target is started in a random location. The return value is a vector of cost values. This vector is of length `n_sims` and has one cost per simulation.

If we want to compare different filters and policies, we can provide a vector of SimUnits to `simulate`:

```
simulate(m::SearchDomain, vsu::Vector{SimUnit}, n_sims::Int)
```

A total of `n_sims` simulations is run per SimUnit. Once a new (random) target location is selected, all SimUnits are run once. The return value is a matrix with one row for each simulation and one column for each sim unit. In each simulation (a row), each simulation unit is tested with the same target location. The values in this matrix correspond to the total cost/reward accumulated druing the simulations.

6.4 Parallel Simulations

To devote `n` cores to running simulations, you must start Julia with the following command

```
julia -p n
```

To run simulations in parallel, use the `parsim` function, which takes the same arguments as the `simulate` function for batch simulations:

```
parsim(m::SearchDomain, su::SimUnit, n_sims::Int)
parsim(m::SearchDomain, vsu::Vector{SimUnit}, n_sims::Int)
```

6.5 Under the Hood

CHAPTER 7

SimUnit

The SimUnit type stores most of the things needed to run a simulation. This includes the filter, policy, cost model, and termination condition. The cost model and termination condition are discussed in more detail later on this page.

A SimUnit has the following fields:

```
type SimUnit
    x::Vehicle
    f::AbstractFilter
    p::Policy
    cm::CostModel
    tc::TerminationCondition
end
```

Below are the constructors for the SimUnit type. At a minimum, it needs a vehicle, filter, and policy. If no cost model is provided, it defaults to ConstantCost(1.0). If no termination condition is provided, it defaults to StepThreshold(10).

```
SimUnit(x, f, p)            # default termination and cost
SimUnit(x, f, p, tc)        # default cost
SimUnit(x, f, p, tc, cm)    # fully defined
```

7.1 Cost Model

The abstract CostModel type handles how costs are applied throughout the simulations. Two cost models are provided:

To define your own cost model, you must extend the abstract CostModel type and implement the get_action_cost function.

```
type CustomCost <: CostModel
    # whatever fields you need for get_action_cost
end
```

(continues on next page)

(continued from previous page)

```
function get_action_cost(a::Action, cm::CostModel)
    # return a Float64 describing cost
end
```

For an example, let's examine the `ConstantCost` model, which applies the same cost at each step. This cost might represent the time each step takes. Therefore, a simulated trajectory's cost would simulate how much time it took. The `ConstantCost` model is defined as follows,

```
type ConstantCost <: CostModel
    value::Float64
end

function get_action_cost(a::Action, cc::ConstantCost)
    return cc.value
end
```

The `MoveAndRotateCost` is a more complex example.

```
type MoveAndRotateCost <: CostModel
    speed::Float64
    time_per_rotation::Float64
end

function get_action_cost(a::Action, marc::MoveAndRotateCost)
    dx = a[1]
    dy = a[2]
    dist = sqrt(dx*dx + dy*dy)
    return (dist / marc.speed) + marc.time_per_rotation
end
```

7.2 Termination Condition

The abstract `TerminationCondition` type determines when an individual simulation should be terminated.

To define your own termination condition, you must extend the abstract `TerminationCondition` type and implement the `is_complete` function.

```
type CustomTC <: TerminationCondition
    # whatever fields you need
end

function is_complete(f::AbstractFilter, ctc::CustomTC, step_count::Int)
    # return true if termination condition reached, false if not
end
```

The `step_count` argument is passed in by the thing. (Clarify if it starts at one or zero.) You can define the `is_complete` function for a specific kind of filter if you only plan on using one filter.

The `StepThreshold` is provided. It terminates after a specified number of steps has been simulated.

```
type StepThreshold <: TerminationCondition
    value::Int
end
function is_complete(f::DF, st::StepThreshold, step_count::Int)
```

(continues on next page)

(continued from previous page)

```
    ret_val = false
    if step_count >= st.value
        ret_val = true
    end
    return ret_val
end
```

The `MaxNormThreshold` termination condition is also provided. The implementation is below

```
type MaxNormThreshold <: TerminationCondition
    value::Float64
end

function is_complete(f::DF, mnt::MaxNormThreshold, ::Int)
    ret_val = false
    if maximum(f.b) > mnt.value
        ret_val = true
    end
    return ret_val
end
```

CHAPTER 8

Visualizatons

Recall that visualizations require the FEBOLPlots.jl package. To install this package, you must call the following in Julia:

```
Pkg.clone("https://github.com/dressel/FEBOLPlots.jl.git")
```

Once the package has been installed, you must include the statement `using FEBOLPLots` whenever using one of its functions. The most useful functions will be `visualize` and `gif`.

8.1 Visualize Function

The `visualize` function allows you to plot out several steps. A simple version can be called with

```
visualize(m, x, f, p, n_steps=10; pause_time=0.3)
```

where `m` is a SearchDomain, `x` is a Vehicle, `f` is a filter, and `p` is a policy.

A different version allows you to pass in `SimUnit`:

```
visualize(m::SearchDomain, su::SimUnit; pause_time=0.3)
```

8.2 Creating GIFs

```
gif(m, x, f, p, num_steps)
```

```
gif(m::SearchDomain, x::Vehicle, f::AbstractFilter, p::Policy, num_steps::Int=10,
↪filename="out.gif"; seconds_per_step=0.5, show_mean=false, show_cov=false, show_
↪path=false)
```

CHAPTER 9

Things to change and modify

- Rename this to be more general
- Add an altitude to Vehicle
- filter should have its own noise model

CHAPTER 10

Indices and tables

- genindex
- modindex
- search