helicopterRL Code Documentation

Pedro Mediano
Imperial College London, UK
pedro.martinez-mediano13@imperial.ac.uk

August 8, 2014

1 Introduction

This is the documentation for the **helicopterRL** project. More information in
https://github.com/pmediano/helicopterRL
http://wp.doc.ic.ac.uk/sml/student-projects/

2 Main function

2.1 helicopter_learn_loop.m

Summary: Script to learn a controller for the helicopter problem

High-level steps

1. Initialize variables and compile RL-Glue trainer
2. Generate random trajectories to speed up learning
3. Apply learning loop (model learning, policy learning, policy application)

Code

```
% 1. Initialization
clear all; close all;
settings_hc;                    % Load scenario-specific settings

use_reward_model = false;
use_previous_history = true;

% Modify the trainer source files and compile for the desired MDP
whichMDP = 0;
setenv('tMDP', num2str(whichMDP));
cd([trainerDir 'consoleTrainerJavaHelicopter']);
!sed "s/whichTrainingMDP = [0-9]/whichTrainingMDP = ${tMDP}/" <src/↩
    consoleTrainerBackup >src/consoleTrainer.java
!make

% Get a suitable name for the history file
cd([pilcoDir 'scenarios/helicopter']);
```

```
getHistoryFilename;


% 2. Collect data with random policy to start learning, or start from the
% latest history file
if ~use_previous_history
    cd([trainerDir 'testTrainerJavaHelicopter']);
    !sed "s/whichTrainingMDP = [0-9]/whichTrainingMDP = ${tMDP}/" <src/testTrainerBackup >src/testTrainer.java
    !make
    !bash run.bash &
    cd([agentDir 'randomAgentMatlab']);
    delete(['randomDataMDP' num2str(whichMDP) '.mat']);
    theRandomAgent = random_agent_matlab(whichMDP, codecDir, agentDir);
    runAgent(theRandomAgent);

    load([agentDir 'randomAgentMatlab/randomDataMDP' num2str(whichMDP) '.mat']);

    delta_n = min(n_init, size(random_data.x, 1));
    x = random_data.x(1:delta_n,:);
    y = random_data.y(1:delta_n,:);
    r_x = random_data.r_x(1:delta_n,:);
    r_y = random_data.r_y(1:delta_n,:);
    H = 5;
else
    load(historyFilename);
    x = history{end}.x;
    y = history{end}.y;
    r_x = history{end}.r_x;
    r_y = history{end}.r_y;
    dynmodel = history{end}.dynmodel;
    if use_reward_model
        if ~isfield(history{end}, 'reward_dynmodel')
            disp('Requested reward model, but no reward model in previous history. Abort.');
            return
        else
            reward_dynmodel = history{end}.reward_dynmodel;
        end
    end
    policy = history{end}.policy;
    H = history{end}.H;
    last_size = history{end}.steps;
end


% Start learning loop. Feel free to Ctrl-C at any time, history file is
% saved automatically and you can resume later using
% |use_previous_history = true;|
cd([pilcoDir 'scenarios/helicopter']);
j = 1;
while true

    % 3. Train GP's and specify cost function

    % 3.1. Train GP for the helicopter dynamics
    trainHelicopterModel
```

```
    if exist('history', 'var'); history{end+1}.dynmodel = dynmodel; else history{1}.dynmodel = dynmodel; end

    % 3.2. Train GP model to predict reward to use as cost function, or
    % set parameters of user-defined cost function
    if use_reward_model
        trainRewardModel;
        cost.fcn = @Rbased_loss_hc;
        cost.rewardgpmodel = reward_dynmodel;
        history{end}.reward_dynmodel = reward_dynmodel;
    else
        cost.width = 1;
        cost.fcn = @loss_hc;
    end

    % 4. Learn policy and save structure
    H = H + 5;
    learnPolicy;
    policy.date = clock;

    history{end}.policy = policy;
    history{end}.H = H;

    % 5. Run the simulator with the latest policy until trajectory is
    % longer than lower limit. If it isn't aggregate more datasets.
    last_size = 0;
    min_last_size = H;
    run_counter = 0;
    while last_size < min_last_size && run_counter < 10
      cd([trainerDir 'consoleTrainerJavaHelicopter']);
      !bash run.bash &
      cd([pilcoDir 'scenarios/helicopter']);
      theAgent = helicopter_agent(policy, codecDir, pilcoDir);
      runAgent(theAgent);

        aux_newdata = load('GPHistory.mat');
      newdata = [newdata; newdata.helicopter_agent_struct];
      last_size = size(newdata,1) - 1;

        run_counter = run_counter + 1;
    end

    % 6. Get new data from the last trajectory. If the last trajectory
    % was long enough, select data
    if last_size < 5990
        delta_n = min(last_size, max_last_size) + 1;
        x = [x; newdata(1:delta_n-1,1:16)];
        y = [y; newdata(2:delta_n,1:12)];
        r_x = [r_x; newdata(2:delta_n,1:12)];
        r_y = [r_y; newdata(2:delta_n, 17)];
    else
        [dynIdx, rewardIdx] = selectData(history{end}, max_last_size);
        x = [x; history{end}.trajectory(dynIdx, 1:16)];
        y = [y; history{end}.trajectory(dynIdx+1, 1:12)];
        r_x = [r_x; history{end}.trajectory(rewardIdx, 1:12)];
        r_y = [r_y; history{end}.trajectory(rewardIdx, 17)];
    end

```

```

    history{end}.steps = last_size;
    history{end}.x = x;
    history{end}.y = y;
    history{end}.r_x = r_x;
    history{end}.r_y = r_y;
    history{end}.trajectory = newdata;

    save(historyFilename, 'history');

    disp(['Length of the last trajectory is ' num2str(last_size)]);

    % 7. Plot NLPD of model when following the trajectory
    drawNLPDplots;

    j = j + 1;

end
```

2.2 settings_hc.m

Summary: Script to set up the helicopter scenario

High-Level Steps

1. Define state and important indices
2. Set up scenario
3. Set up the plant structure
4. Set up the policy structure
5. Set up the cost structure
6. Set up the GP dynamics model structure
7. Parameters for policy optimization
8. Plotting verbosity
9. Some array initializations

Code

```
warning('on','all'); format short; format compact

% Include some paths
try
  rd = '../../';
  addpath([rd 'base'],[rd 'util'],[rd 'gp'],[rd 'control'],[rd 'loss']);
catch
end

basename = 'helicopter_';     % filename used for saving data
trainerDir = '/home/pmediano/Downloads/RL/helicopter/trainers/';
agentDir = '/home/pmediano/Downloads/RL/helicopter/agents/';
setenv('AGENTDIR', agentDir);
codecDir = '/home/pmediano/Downloads/RL/helicopter/src/';
pilcoDir = '/home/pmediano/Downloads/RL/pilcoV0.9/';
```

```
addpath([agentDir 'GPAgentMatlab']);

rand('seed',1); randn('seed',1); format short; format compact;

% 1. Define state and important indices

% 1a. State representation
%  1   u_err     forward velocity
%  2   v_err     sideways velocity
%  3   w_err     downward velocity
%  4   x_err     forward error
%  5   y_err     sideways error
%  6   z_err     downward error
%  7   p_err     angular rate around forward axis
%  8   q_err     angular rate around sideways (to the right) axis
%  9   r_err     angular rate around vertical (downward) axis
%  10  qx_err     quaternion entries, x,y,z,w   q = [ sin(theta/2) * axis; cos(theta/2)],
%  11 qy_err         where axis = axis of rotation; theta is amount of rotation around that axis
%  12 qz_err         [recall: any rotation can be represented by a single rotation around some axis]

observationIdx = 1:12;
nVar = 12;

%  Action representation
%  13 longitudinal cyclic pitch
%  14 latitudinal (left-right) cyclic pitch
%  15 main rotor collective pitch
%  16 tail rotor collective pitch

actionIdx = 13:16;
nU = 4;
maxU = [1, 1, 1, 1];

%  17 reward

rewardIdx = 17;

% 1b. Important indices

odei = observationIdx;        % indicies for the ode solver
augi = [];                    % indicies for variables augmented to the ode variables
dyno = observationIdx;        % indicies for the output from the dynamics model and indicies to loss
angi = [];                    % indicies for variables treated as angles (using sin/cos representation)
dyni = observationIdx;        % indicies for inputs to the dynamics model
poli = observationIdx;        % indicies for variables that serve as inputs to the policy
difi = observationIdx;        % indicies for training targets that are differences (rather than values)

% 2. Set up the scenario
mu0 = zeros([nVar 1]);              % initial state mean (column vector)
% S0 = diag(zeros([1 nVar]));         % initial state covariance
S0 = 0.0001*eye(nVar);
```

```
mu0Sim(odei,:) = mu0; S0Sim(odei,odei) = S0;        % Specify initial state distribution -
mu0Sim = mu0Sim(dyno); S0Sim = S0Sim(dyno,dyno);    % in this case, the origin.

n_init = 100;                      % no. of initial data points (computed with random policy)
max_last_size = 100;               % max no. of data points added to the dataset in each iteration
N = 20;                            % max no. of controller optimizations

% 3. Set up the plant structure
%plant.ctrl = @zoh;                   % controler is zero-order-hold
plant.odei = odei;                  % indices to the varibles for the ode solver
plant.augi = augi;                  % indices of augmented variables
plant.angi = angi;
plant.poli = poli;
plant.dyno = dyno;
plant.dyni = dyni;
plant.difi = difi;
plant.prop = @propagated;   % handle to function that propagates state over time



% 4. Set up the policy structure

policy.fcn = @(policy,m,s)conCat(@conlin,@gSat,policy,m,s);% controller
                                                        % representation
policy.maxU = maxU;                                   % max. amplitude of
                                                        % actions
[mm, ss, cc] = gTrig(mu0, S0, plant.angi);               % represent angles
mm = [mu0; mm]; cc = S0*cc; ss = [S0 cc; cc' ss];        % in complex plane

% Uncomment the following lines if policy is @conlin
policy.p.w = 1e-2*randn(length(policy.maxU),length(poli));  % weight matrix
policy.p.b = zeros(length(policy.maxU),1);                   % bias


% Uncomment the following lines if policy is @congp
% nc = 100;                              % size of controller training set
% policy.p.inputs = gaussian(mm(poli), ss(poli,poli), nc)'; % init. location of
%                                                          % basis functions
% policy.p.targets = 0.1*randn(nc, length(policy.maxU));    % init. policy targets
%                                                          % (close to zero)
% policy.p.hyp = ...                                       % initialize policy
%  repmat(log([1 1 1 1 1 1 1 1 1 1 1 1 1 0.01]'), 1, 4);   % hyper-parameters


% 5. Set up the cost structure
cost.fcn = @loss_hc;                          % cost function
cost.gamma = 1;                               % discount factor
cost.width = 1;                             % cost function width
cost.expl = 0;                                % exploration parameter (UCB)
cost.angle = plant.angi;                      % index of angle (for cost function)
cost.target = zeros(nVar, 1);                     % target state

% Alternatively, define a function to translate RL-Glue rewards to cost
reward2loss = @(r) -exp(r/5);

```

```

% 6. Set up the GP dynamics model structure
dynmodel.fcn = @gp1d;                  % function for GP predictions
dynmodel.train = @train;               % function to train dynamics model
dynmodel.induce = zeros(5000,0,1);      % shared inducing inputs (sparse GP)
trainOpt = [300 300];                  % defines the max. number of line searches
                                       % when training the GP dynamics models
                                       % trainOpt(1): full GP,
                                       % trainOpt(2): sparse GP (FITC)

% 7. Parameters for policy optimization
opt.length = 50;                          % max. number of line searches
opt.MFEPLS = 7;                           % max. number of function evaluations
                                           % per line search
opt.verbosity = 3;                         % verbosity: specifies how much
                                           % information is displayed during
                                           % policy learning. Options: 0-3

% 8. Plotting verbosity
plotting.verbosity = 3;              % 0: no plots
                                     % 1: some plots
                                     % 2: all plots

% 9. Some initializations
fantasy.mean = cell(1,N); fantasy.std = cell(1,N);
realCost = cell(1,N); M = cell(N,1); Sigma = cell(N,1);
newdata = [];
```

3 Agents

3.1 random_agent_matlab.m

Summary: Random policy agent used to collect initial data. Contains the functions needed to communicate with the RL-Glue Core.

Contents

1. **helicopter_agent**: set useful paths and function handles
2. **helicopter_agent_init**: initialize data structure
3. **helicopter_agent_start**: take the first step
4. **helicopter_agent_step**: take a step
5. **helicopter_agent_message**: communicate between trainer and agent
6. **helicopter_agent_end**: finish episode
7. **helicopter_agent_cleanup**: save data structure

Code

```
function theAgent=random_agent_matlab(MDP, codec_base, agent_base)
% Add paths and fill agent structure

    addpath([codec_base 'agent'], [codec_base 'glue'], codec_base);
```

```

    theAgent.agent_init=@helicopter_agent_init;
    theAgent.agent_start=@helicopter_agent_start;
    theAgent.agent_step=@helicopter_agent_step;
    theAgent.agent_end=@helicopter_agent_end;
    theAgent.agent_cleanup=@helicopter_agent_cleanup;
    theAgent.agent_message=@helicopter_agent_message;

    global whichMDP;
    whichMDP = MDP;
    global agentDir;
    agentDir = agent_base;

end

function helicopter_agent_init(taskSpec)
% Initialize agent
% Note that helicopter_agent_struct is created and saved in |start| and
% |end|, different from what is done in |GPAgentMatlab|
end

function theAction=helicopter_agent_start(theObservation)
% Take the first step of the agent as the episode starts, and store data

    global helicopter_agent_struct;
    global timeStep;
    helicopter_agent_struct = zeros(2,17);
    timeStep = 1;

    theAction = org.rlcommunity.rlglue.codec.types.Action();
  theAction.doubleArray = rand(4,1)-1;

    helicopter_agent_struct(timeStep, 1:12) = theObservation.doubleArray;
    helicopter_agent_struct(timeStep, 13:16) = theAction.doubleArray;
    helicopter_agent_struct(timeStep, 17) = 0;

end

function theAction=helicopter_agent_step(theReward, theObservation)
% Take a step and store data

  global helicopter_agent_struct;
    global timeStep;

    theAction = org.rlcommunity.rlglue.codec.types.Action();
    theAction.doubleArray = 2*rand(4,1)-1;

    timeStep = timeStep + 1;
    helicopter_agent_struct(timeStep, 1:12) = theObservation.doubleArray;
    helicopter_agent_struct(timeStep, 13:16) = theAction.doubleArray;
    helicopter_agent_struct(timeStep, 17) = theReward;

end

function helicopter_agent_end(theReward)
% An episode ends and save the data

    global helicopter_agent_struct;
```

```
    global timeStep;
    global whichMDP;
    global agentDir;

    basename = [agentDir 'randomAgentMatlab/'];
    filename = sprintf('randomDataMDP%i.mat', whichMDP);
    fullname = strcat(basename, filename);
    if exist(fullname, 'file')
        load(fullname);
        random_data.x = [random_data.x; helicopter_agent_struct(1:end-1,1:16)];
        random_data.y = [random_data.y; helicopter_agent_struct(2:end,1:12)];
        random_data.r_x = [random_data.r_x; helicopter_agent_struct(2:end,1:12)];
        random_data.r_y = [random_data.r_y; helicopter_agent_struct(2:end,17)];
        random_data.all = [random_data.all; helicopter_agent_struct];
    else
        random_data.x = helicopter_agent_struct(1:end-1,1:16);
        random_data.y = helicopter_agent_struct(2:end,1:12);
        random_data.r_x = helicopter_agent_struct(2:end,1:12);
        random_data.r_y = helicopter_agent_struct(2:end, 17);
        random_data.all = helicopter_agent_struct;
    end
    if timeStep > 1
        save(fullname, 'random_data');
    end
end

function returnMessage=helicopter_agent_message(theMessageJavaObject)
% Custom function for trainer-agent communication

    inMessage=char(theMessageJavaObject);
    global whichMDP;
  if strcmp(inMessage,'What is your name?')==1
    returnMessage='My name is random_agent_matlab';
    elseif strcmp(inMessage,'Which MDP?')==1
        returnMessage=sprintf('I am doing MDP = %i', whichMDP);
    else
        returnMessage='I don\''t know how to respond to your message';
  end
end

function helicopter_agent_cleanup()
% Agent cleanup

end
```

3.2 helicopter_agent.m

Summary: Helicopter control agent. Contains the functions needed to communicate with the RL-Glue Core.

Contents

1. `helicopter_agent`: set useful paths and function handles
2. `helicopter_agent_init`: initialize data structure
3. `helicopter_agent_start`: take the first step

4. helicopter_agent_step: take a step
5. helicopter_agent_message: communicate between trainer and agent
6. helicopter_agent_end: finish episode
7. helicopter_agent_cleanup: save data structure

Code

```
function theAgent=helicopter_agent(policy_input, codec_base, pilco_root)
% Add paths and fill agent structure

    global policy
    policy = policy_input;

    addpath([codec_base 'agent'], [codec_base 'glue'], codec_base);

    addpath([pilco_root 'base'],[pilco_root 'util'],[pilco_root 'gp'],[pilco_root '↩
        control'],[pilco_root 'loss']);

    theAgent.agent_init=@helicopter_agent_init;
    theAgent.agent_start=@helicopter_agent_start;
    theAgent.agent_step=@helicopter_agent_step;
    theAgent.agent_end=@helicopter_agent_end;
    theAgent.agent_cleanup=@helicopter_agent_cleanup;
    theAgent.agent_message=@helicopter_agent_message;

end

function helicopter_agent_init(taskSpec)
% This is a persistent struct we will use to store the data collected for
% the next learning iteration
global helicopter_agent_struct;
helicopter_agent_struct = zeros(2,17);

end

function theAction=helicopter_agent_start(theObservation)
% Take the first step of the agent as the episode starts, and store data

    global helicopter_agent_struct;
    global timeStep;
    global policy
    timeStep = 1;

    theAction = org.rlcommunity.rlglue.codec.types.Action();
  theAction.doubleArray = policy.fcn(policy, theObservation.doubleArray, zeros(length↩
      (theObservation.doubleArray)));

    helicopter_agent_struct(timeStep, 1:12) = theObservation.doubleArray;
    helicopter_agent_struct(timeStep, 13:16) = theAction.doubleArray;
    helicopter_agent_struct(timeStep, 17) = 0;

end

function theAction=helicopter_agent_step(theReward, theObservation)
% Take a step and store data

```

```matlab
    global helicopter_agent_struct;
    global timeStep;
    global policy;

    theAction = org.rlcommunity.rlglue.codec.types.Action();
    theAction.doubleArray = policy.fcn(policy, theObservation.doubleArray, zeros(length(theObservation.doubleArray)));

    timeStep = timeStep + 1;
    helicopter_agent_struct(timeStep, 1:12) = theObservation.doubleArray;
    helicopter_agent_struct(timeStep, 13:16) = theAction.doubleArray;
    helicopter_agent_struct(timeStep, 17) = theReward;

end

function helicopter_agent_end(theReward)
    % An episode ends
end

function returnMessage=helicopter_agent_message(theMessageJavaObject)
% Custom function for trainer-agent communication

    inMessage=char(theMessageJavaObject);
    global policy;

    if strcmp(inMessage,'what is your name?')==1
    returnMessage='my name is helicopter_agent, Matlab edition!';
    elseif strcmp(inMessage, 'when')==1
        % Print policy training timestamp
        returnMessage = num2str(policy.date);
    else
        returnMessage='I don\''t know how to respond to your message';
    end

end

function helicopter_agent_cleanup()
% On cleanup, save the collected data to a MAT-file

    global helicopter_agent_struct;
    save('GPHistory', 'helicopter_agent_struct');
end
```

4 Model evaluation functions

4.1 checkModel.m

Summary: Compute goodness-of-fit measures to assess the predictive power of the model. If no test dataset is provided, measure on train dataset.

Input arguments:

```
model            model to be assessed
test\_x           (optional) test inputs to try model on
test\_y           (optional) test outputs to try model on
```

Code

```
function checkModel( model, test_x, test_y )

    if nargin == 1; x = model.inputs; y = model.targets;
    else x = test_x; y = test_y; end;

    % 1. Gather information about the model and the test dataset
    [m, n] = size(x);
    [m1, d] = size(y);
    if m1~=m; return; end;
    LP = zeros(m, 1);
    sampleSize = 1000;
    sampleLP = zeros(m, 1);
    predictions = zeros(m, d);

    % 2. Compute model and optimal NLPD
    for j=1:m
        [mu, sigma] = model.fcn(model, x(j,:)', zeros(n));
        %disp(mu);
        LP(j) = NLPD(y(j,:)', mu, sigma, d);
        sample = mvnrnd(mu, sigma, sampleSize);
        aux = cellfun(@(v) NLPD(v', mu, sigma, d), num2cell(sample, 2));
        sampleLP(j) = mean(aux);
        predictions(j, :) = mu;
    end
    fprintf('NLPD of model\n \t True \t Optimal \n Mean %f  %f \n Std  %f  %f\n',mean(LP), mean(sampleLP), std(LP), std(sampleLP));

    % 3. Compute relative error
    diff = abs((predictions-y)./y);
    relDiff = mean(100*diff)';
    fprintf('Mean relative difference of each variable (%%):\n');
    disp(relDiff);
    fprintf('Total average relative difference is %f%%\n', mean(relDiff));


    % 4. Check SNR's
    hyp = model.hyp;
    SNR = exp(hyp(end-1,:)-hyp(end,:));
    if any(SNR > 500)
        fprintf('SNR is greater than 500 for variables %s\n', num2str(find(SNR>500)));
    else
        fprintf('All SNRs are less than 500.\n');
    end

    % 5. Check log-signal-std
    lsstd = hyp(end-1, :);
    if any(lsstd<log(0.1))
        low = lsstd<log(0.1);
        fprintf('LSSTD is lower than log(0.1) for\n variables \t%s\n values \t%s\n', num2str(find(low)), num2str(lsstd(low)));
    end
    if any(lsstd>log(10))
        high = lsstd>log(10);
```

```
        fprintf('LSSTD is higher than log(10) for\n variables \t%s\n values \t%s\n', num2str(find(high)), num2str(lsstd(high)));
    end
    if all(and(lsstd>log(0.1), lsstd<log(10)))
        fprintf('All log-signal-std are within safe range.\n');
    end

    % 6. Check lengthscales
    lengthScale = exp(hyp(1:end-2,:));
    inputStd = repmat(std(model.inputs)',[1,d]);
    fprintf('LengthScale/std ratios:\n');
    disp(lengthScale./inputStd);


end
```

4.2 drawNLPDplots.m

Summary: Script to assess predictive power of a model. Computes NLPD and RelDiff of model predictions along the trajectory generated by the policy trained on it.

High-level steps

1. Initialize empty matrices
2. Loop over the trajectory gathering statistics
3. Draw plots

Code

```
try
    LP = zeros(last_size, 1);
    sampleLP = zeros(last_size, 1);
    sampleLPstd = zeros(last_size, 1);
    relDiff = zeros(last_size, 1);
    sampleSize = 500;
    fprintf(1, '\nCalculating NLPD and relative differences of predictions and experiment:\nStep:     ');
    for k=1:last_size
        [mu, sigma] = dynmodel.fcn(dynmodel, newdata(k,1:16)', 0.0001*eye(16));
        mu(difi) = mu(difi) + newdata(k, difi)';
        LP(k) = NLPD(newdata(k+1,1:12)', mu, sigma, 12);
        sample = mvnrnd(mu, sigma, sampleSize);
        aux = cellfun(@(v) NLPD(v', mu, sigma, 12), num2cell(sample, 2));
        sampleLP(k) = mean(aux);
        sampleLPstd(k) = std(aux);
        relDiff(k) = 100*mean((mu - newdata(k+1,1:12)')./newdata(k+1,1:12)');
        fprintf(1, '\b\b\b\b%4i', k);
    end
    aux = 1:last_size;
    figure(10)
    plot(aux, LP, 'k-',  ...
      aux, sampleLP, 'g-', ...
      aux, sampleLP + sampleLPstd, 'r:', ...
```

```
      aux, sampleLP - sampleLPstd, 'r:'  ); drawnow;
      xlabel('Steps');
      ylabel('NLPD');
    legend('Real', 'Optimal', '1-sigma belt');

    figure(11)
    plot(relDiff); drawnow;
    xlabel('Steps');
    ylabel('Relative Difference (%)');

catch ME
    disp('Error computing NLPD');
    disp('Exception:');
    disp(ME);
    disp('Covariance matrix:');
    disp(sigma);
end
```

4.3 evaluateHistory.m

Summary: Evaluate the performance of the policies in a certain learning history.

Input arguments: whichMDP: number of the MDP to test policies on history: cell array, each cell contains a policy to be evaluated

Output arguments: res: matrix with the average and std of steps survived by each policy

High-level steps

1. Set up configuration for helicopter scenario (paths, variables, etc)
2. Compile java trainer to run desired MDP
3. Evaluate each policy in history file \verb@sample@ times and return statisics

Code

```
function res=evaluateHistory(whichMDP, history)

    % 1. Set up helicopter configuration
    settings_hc;

    n_iter = numel(history);
    res = zeros([n_iter, 5]);

    % 2. Compile java trainer
    setenv('tMDP', num2str(whichMDP));
    cd([trainerDir 'consoleTrainerJavaHelicopter']);
    !sed "s/whichTrainingMDP = [0-9]/whichTrainingMDP = ${tMDP}/" <src/consoleTrainerBackup >src/consoleTrainer.java
    !make

    % 3. Loop over history and evaluate policies
    for j=1:n_iter
```

```
        theAgent = helicopter_agent(history{j}.policy, codecDir, pilcoDir);
        res(j, 1) = size(history{j}.dynmodel.targets, 1);
        [res(j, 2), res(j, 3), res(j, 4), res(j, 5)] = evaluatePolicy(theAgent, trainerDir, pilcoDir);
    end

end

function [mean_time, std_time, mean_reward, std_reward]=evaluatePolicy(theAgent, trainerDir, pilcoDir, nb_samples)
% Auxiliar function for evaluateHistory.

    if nargin < 4
        sample = 10;    % Number of trajectories to generate
    else
        sample = nb_samples;
    end
    steps = zeros([sample, 1]);
    rewards = zeros([sample, 1]);

    for j=1:sample
        cd([trainerDir 'consoleTrainerJavaHelicopter']);
        !bash run.bash &
        cd([pilcoDir 'scenarios/helicopter']);
        runAgent(theAgent);
        newdata = load('GPHistory.mat');
        steps(j) = size(newdata.helicopter_agent_struct, 1) - 1;
        rewards(j) = mean(newdata.helicopter_agent_struct(:, 17));
    end

    mean_time = mean(steps);
    std_time = std(steps);
    mean_reward = mean(rewards);
    std_reward = std(rewards);

end
```

4.4 NLPD.m

Summary: Computes the Negative Log Predictive Density for a multivariate gaussian distribution.

Input arguments: x observation mu mean vector of the distribution sigma covariance matrix of the distribution d (optional) dimensionality of input space

Output arguments: lp computed NLPD

Code

```
function [ lp ] = NLPD(x, mu, sigma, d)

    if nargout==3; d = numel(x); end;

```

```
    diff = x-mu;
    lp = 0.5*(log(det(sigma)) + diff'*(sigma\diff) + d*1.837877066409345);

end
```

5 Helper functions

5.1 getHistoryFilename.m

Summary: Script to find a name for the history file to be used. If user set `use_previous_history=true`, the name of the latest existing history file is returned. If `use_previous_history=false`, return a name for a new file.

Code

```
historyFilename = sprintf('historyMDP%i.mat', whichMDP);

if ~use_previous_history && ~exist(historyFilename, 'file')
    disp(['Creating first history file for this MDP, ' historyFilename]);

elseif use_previous_history && ~exist(historyFilename, 'file')          % If the user requests a previous history but
                                                                         % there are no saved histories, notify
    disp(['No previous history to load. Generating random data. Saving history in ' historyFilename]);
    use_previous_history = false;
else                                                                     % If a history file exists, add a subindex
    history_nb = 2;
    historyFilename = sprintf('historyMDP%i.%i.mat', whichMDP, history_nb);
    if use_previous_history && ~exist(historyFilename, 'file')
        historyFilename = sprintf('historyMDP%i.mat', whichMDP);
        disp(['Using data from file ' historyFilename]);
    else
        while exist(historyFilename, 'file')                             % Find the latest history file for this MDP
            history_nb = history_nb + 1;
            historyFilename = sprintf('historyMDP%i.%i.mat', whichMDP, history_nb);
        end
        if use_previous_history                                          % If user requested previous history, go back
                                                                         % one step to get the latest existing file
            history_nb = history_nb - 1;
            historyFilename = sprintf('historyMDP%i.%i.mat', whichMDP, history_nb);
            disp(['Using history from the latest file, ' historyFilename]);
        else
            disp(['Creating new file ' historyFilename]);
        end
```

```
    end
end
```

5.2 loss_dp.m

Summary: Double-Pendulum loss function; the loss is $1 - \exp(-0.5 * d^2 * a)$, where $a > 0$ and d^2 is the squared difference between the actual and desired position of the tip of the outer pendulum. The mean and the variance of the loss are computed by averaging over the Gaussian distribution of the state $p(x) = \mathcal{N}(m, s)$ with mean m and covariance matrix s. Derivatives of these quantities are computed when desired.

```
function [L, dLdm, dLds, S2] = loss\_dp(cost, m, s)
```

Input arguments:

```
cost            cost structure
  .width        array of widths of the cost (summed together)
  .expl         (optional) exploration parameter
  .angle        (optional) array of angle indices
  .target       target state                              [D x  1 ]
m               mean of state distribution                [D x  1 ]
s               covariance matrix for the state distribution   [D x  D ]
```

Output arguments:

```
L     expected cost                                       [1 x  1 ]
dLdm  derivative of expected cost wrt. state mean vector  [1 x  D ]
dLds  derivative of expected cost wrt. state covariance matrix [1 x D^2]
S2    variance of cost                                    [1 x  1 ]
```

High-Level Steps

1. Precomputations
2. Define static penalty as distance from target setpoint
3. Calculate loss

```
function [L, dLdm, dLds, S2] = loss_hc(cost, m, s)
```

Code

```
if isfield(cost,'width'); cw = cost.width; else cw = 1; end
if ~isfield(cost,'expl') || isempty(cost.expl); b = 0; else b =  cost.expl; end


% 1. Some precomputations
```

```
D0 = size(s,2); D = D0;                                  % state dimension
D1 = D0 + 2*length(cost.angle);           % state dimension (with sin/cos)

M = zeros(D1,1); M(1:D0) = m; S = zeros(D1); S(1:D0,1:D0) = s;
Mdm = [eye(D0); zeros(D1-D0,D0)]; Sdm = zeros(D1*D1,D0);
Mds = zeros(D1,D0*D0); Sds = kron(Mdm,Mdm);

% 2. Define static penalty as distance from target setpoint
Q = diag(ones(numel(m), 1));

target = [cost.target(:); gTrig(cost.target(:), 0*s, cost.angle)];
% 3. Calculate loss
L = 0; dLdm = zeros(1,D0); dLds = zeros(1,D0*D0); S2 = 0;
for i = 1:length(cw)                        % scale mixture of immediate costs
    cost.z = target; cost.W = Q/cw(i)^2;
  [r, rdM, rdS, s2, s2dM, s2dS] = lossSat(cost, M, S);

  L = L + r; S2 = S2 + s2;
  dLdm = dLdm + rdM(:)'*Mdm + rdS(:)'*Sdm;
  dLds = dLds + rdM(:)'*Mds + rdS(:)'*Sds;

  if (b~=0 || ~isempty(b)) && abs(s2)>1e-12
    L = L + b*sqrt(s2);
    dLdm = dLdm + b/sqrt(s2) * ( s2dM(:)'*Mdm + s2dS(:)'*Sdm )/2;
    dLds = dLds + b/sqrt(s2) * ( s2dM(:)'*Mds + s2dS(:)'*Sds )/2;
  end
end

% normalize
n = length(cw); L = L/n; dLdm = dLdm/n; dLds = dLds/n; S2 = S2/n;
```

Rbased_loss_hc.m

```
function [L, dLdm, dLds, S2] = Rbased\_loss\_hc(cost, m, s)
```

Input arguments:

```
cost            cost structure
 .rewardgpmodel GP to predict reward given state
m               mean of state distribution                  [D x  1 ]
s               covariance matrix for the state distribution    [D x  D ]
```

Output arguments:

```
L     expected cost                                          [1 x  1 ]
dLdm  derivative of expected cost wrt. state mean vector        [1 x  D ]
dLds  derivative of expected cost wrt. state covariance matrix  [1 x D^2]
S2    variance of cost                                       [1 x  1 ]
```

```
function [L, dLdm, dLds, S2] = Rbased_loss_hc(cost, m, s)
```

Code

```
[L, S2, ~, dLdm, ~, ~, dLds, ~, ~] = gp1d(cost.rewardgpmodel, m, s);
```

```
end
```

5.3 trainHelicopterModel.m

Summary: Script to learn the dynamics model. If SNR is too high, white noise is added to the GP targets and model is trained again.

High-level-steps

1. Set up GP structure
2. Train one GP for each variable separately and merge results into the complete GP
3. Check length-scales

Code

```
% 1. Set up GP inputs and targets
Du = length(policy.maxU); Da = length(plant.angi); % no. of ctrl and angles
xaug = [x(:,dyno) x(:,end-Du-2*Da+1:end-Du)];     % x augmented with angles
dynmodel.inputs = [xaug(:,dyni) x(:,end-Du+1:end)];     % use dyni and ctrl
dynmodel.targets = y(:,dyno);
dynmodel.targets(:,difi) = dynmodel.targets(:,difi) - x(:,dyno(difi));

dynmodel.hyp = zeros([nVar+nU+2, nVar]);

% 2. For each variable, train a separate GP until SNR falls below the limit
for k=1:nVar
    disp(['Training GP for variable ' num2str(k)]);
    auxDynmodel.fcn = dynmodel.fcn;
    auxDynmodel.train = dynmodel.train;
    auxDynmodel.induce = dynmodel.induce;
    auxDynmodel.inputs = dynmodel.inputs;
    auxDynmodel.targets = dynmodel.targets(:,observationIdx(k));
    auxDynmodel.hyp = dynmodel.hyp(:,k);

    auxDynmodel = auxDynmodel.train(auxDynmodel, plant, trainOpt);

    Xh = auxDynmodel.hyp;
    SNR = exp(Xh(end-1)-Xh(end));
    disp(['SNR = ' num2str(SNR)]);
    while SNR>500
        disp('SNR too high. Add noise and re-train.');
        std_target = std(auxDynmodel.targets);
        auxDynmodel.targets = auxDynmodel.targets + 0.01*std_target*randn(size(↩
            auxDynmodel.targets));
        auxDynmodel = auxDynmodel.train(auxDynmodel, plant, trainOpt);
```

```
        Xh = auxDynmodel.hyp;
        SNR = exp(Xh(end-1)-Xh(end));
        disp(['Now SNR is ' num2str(SNR)]);
    end

%   2.1. Merge the trained hyper-parameters with the complete GP
    dynmodel.hyp(:,k) = auxDynmodel.hyp;

end

% 3. Show the result of various tests based on the model's length-scales
Xh = dynmodel.hyp;
log_signal_std = (Xh(end-1,:)<log(0.1)) + (Xh(end-1,:)>log(10));
if any(log_signal_std)
    fprintf('Helicopter model log-signal-std hyperparameter is out of range for variables %s \n', num2str(find(log_signal_std)));
end

input_std = std(dynmodel.inputs);
if any(Xh(1:16,:) > 100*repmat(input_std',[1, nVar]))
    fprintf('Length-scales are much bigger than input std\n');
end

if any(Xh(1:16,:) < 0.01*repmat(input_std',[1, nVar]))
    fprintf('Length-scales are much smaller than input std\n');
end
```

5.4 trainRewardModel.m

Summary: Script to train the reward model.

High-level steps

1. Set up GP structure and train
2. Add noise and re-train while SNR is high
3. Check length-scales of the model

Code

```
% 1. Set up and train reward GP structure
clear reward_dynmodel;
reward_dynmodel.fcn = @gp1d;
reward_dynmodel.train = @train;
reward_dynmodel.induce = zeros(5000,0,1);
reward_dynmodel.inputs = r_x;
reward_dynmodel.targets = reward2loss(r_y);

reward_dynmodel = reward_dynmodel.train(reward_dynmodel);  %  train dynamics GP

% 2. Add noise and train again while SNR is high
r_hyp = reward_dynmodel.hyp;
highSNR = any(exp(r_hyp(end-1,:)-r_hyp(end,:)) > 500);

```

```
while highSNR
    target_std = std(reward_dynmodel.targets);
    reward_dynmodel.targets = reward_dynmodel.targets + 0.01*target_std*randn(size(↩
        reward_dynmodel.targets));
    reward_dynmodel = reward_dynmodel.train(reward_dynmodel);  %  train dynamics GP
    r_hyp = reward_dynmodel.hyp;
    highSNR = exp(r_hyp(end-1)-r_hyp(end)) > 500;
    disp(['Reward model SNR = ' num2str(exp(r_hyp(end-1)-r_hyp(end)))]);
end

% 3. Check length-scales
if r_hyp(end-1) > log(10)
    fprintf('Reward log-signal-std hyperparameter is %.4f, greater than log(10)\n', ↩
        r_hyp(end-1));
end

if r_hyp(end-1) < log(0.1)
    fprintf('Reward log-signal-std hyperparameter is %.4f, less than log(0.1)\n', ↩
        r_hyp(end-1));
end

input_std = std(reward_dynmodel.inputs)';
fprintf('Reward input lengthscales and input std:\n');
disp(num2str([exp(r_hyp(1:12)), input_std]));
```