PILCO Code Documentation v0.9

Marc Peter Deisenroth
Imperial College London, UK
Technische Universität Darmstadt, Germany
m.deisenroth@imperial.ac.uk

Andrew McHutchon
University of Cambridge, UK
ajm257@cam.ac.uk

Joe Hall
University of Cambridge, UK
jah215@cam.ac.uk

Carl Edward Rasmussen
University of Cambridge, UK
cer54@cam.ac.uk

May 27, 2016

Contents

Abstract

We describe a Matlab package for the PILCO policy search framework for data-efficient reinforcement learning. This package implements the PILCO learning framework in multiple different scenarios with continuous states and actions: pendulum swing-up, cart-pole swing-up, double-pendulum swing-up (with either one or two actuators), cart-double-pendulum swing-up, and unicycling. Results from some of these scenarios have been presented previously in [**?**, **?**]. The high-level steps of the PILCO algorithm, which are also implemented in this software package, are the following: Learn a Gaussian process (GP) model of the system dynamics, perform deterministic approximate inference for policy evaluation, update the policy parameters using exact gradient information, apply the learned controller to the system. The software package provides an interface that allows for setting up novel tasks without the need to be familiar with the intricate details of model learning, policy evaluation and improvement.

Chapter 1

Introduction

Reinforcement learning (RL) is a general paradigm for learning (optimal) policies for stochastic sequential decision making processes [**?**]. In many practical engineering applications, such as robotics and control, RL methods are difficult to apply: First, the state and action spaces are often continuous valued and high dimensional. Second, the number of interactions that can be performed with a real system is practically limited. Therefore, learning methods that efficiently extract valuable information from available data are important. Policy search methods have been playing an increasingly important role in robotics as they consider a simplified RL problem and search (optimal) policies in a constrained policy space [**?**, **?**], typically in an episodic set-up, i.e., a finite-horizon set-up with a fixed initial state (distribution).
We present the PILCO software package for data-efficient policy search that allows to learn (non)linear controllers with hundreds of parameters for high-dimensional systems. A key element is a learned probabilistic GP dynamics model. Uncertainty about the learned dynamics model (expressed by the GP posterior) is explicitly taken into account for multiple-step ahead predictions, policy evaluation, and policy improvement.

1.1 Intended Use

The intended use of this software package is to provide a relatively simple interface for practitioners who want to solve RL problems with continuous states and actions efficiently, i.e., without the need of excessively sized data sets. As PILCO is very data efficient, it has been successfully applied to robots to learn policies from scratch, i.e., without the need to provide demonstrations or other "informative" prior knowledge [**?**, **?**]. In this document, we intentionally hide the involved model learning and inference mechanisms to make the code more accessible. Details about inference and model learning can be found in [**?**].
This software package is *not* ideal for getting familiar with classical RL scenarios and algorithms (e.g., Q-learning, SARSA, TD-learning), which typically involve discrete states and actions. For this purpose, we refer to existing RL software packages, such as RLGlue, CLSquare[1], PIQLE[2], RL Toolbox[3], LibPG[4], or RLPy[5].

[1] http://www.ni.uos.de/index.php?id=70
[2] http://piqle.sourceforge.net/
[3] http://www.igi.tugraz.at/ril-toolbox/
[4] http://code.google.com/p/libpgrl/
[5] http://acl.mit.edu/RLPy/

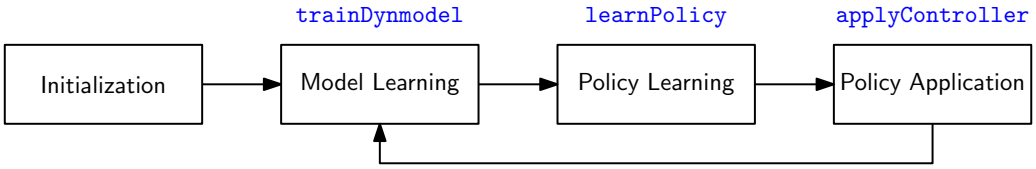

Figure 1.1: Main modules: After an initialization, the first module is responsible for training a GP from available data. The second module is used for policy learning, which consists of policy evaluation and improvement. The third module is responsible for applying the learned policy to the system, which can be either a simulated system or a real system, such as a robot. The collected data from this application is used for updating the model, and the cycle starts from the beginning.

1.2 Software Design and Implementation

The high-level modules 1) Model learning, 2) Policy learning, 3) Policy application are summarized in the following.

1.2.1 Model Learning

The forward model learned is a non-parametric, probabilistic Gaussian process [**?**]. The non-parametric property of the GP does not require an explicit task-dependent parametrization of the dynamics of the system. The probabilistic property of the GP reduces the effect of model errors.
Inputs to the GP are state-action pairs $(\boldsymbol{x}_t, \boldsymbol{u}_t)$, where t is a time index. Training targets are either successor states $\boldsymbol{x}_{t+1}$ or differences $\Delta_t = \boldsymbol{x}_{t+1} - \boldsymbol{x}_t$.
By default, a full GP model is trained by evidence maximization, where we penalize high signal-to-noise ratios in order to maintain numerical stability. There is an option to switch to sparse GPs in case the number of data points exceeds a particular threshold. We implemented the FITC/SPGP sparse GP method proposed by [**?**] for GP training and predictions at uncertain inputs.

1.2.2 Policy Learning

For policy learning, PILCO uses the learned GP forward model to compute approximate long-term predictions $p(\boldsymbol{x}_1|\pi), \ldots, p(\boldsymbol{x}_T|\pi)$ for a given controller π. To do so, we follow the analytic moment-matching approach proposed by [**?**], and approximate all $p(\boldsymbol{x}_t|\pi)$ by Gaussians $\mathcal{N}\big(\boldsymbol{x}_t \,|\, \boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t\big)$.
Policy Evaluation. Once the long-term predictions $p(\boldsymbol{x}_1|\pi), \ldots, p(\boldsymbol{x}_T|\pi)$ are computed, the expected long-term cost

$$J^\pi = \sum_{t=1}^{T} \mathbb{E}[c(\boldsymbol{x}_t)|\pi]\,, \qquad p(\boldsymbol{x}_0) = \mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)\,, \tag{1.1}$$

can be computed analytically for many cost functions c, e.g., polynomials, trigonometric functions, or Gaussian-shaped functions. In our implementation, we typically use a Gaussian-shaped cost function

$$c(\boldsymbol{x}) = 1 - \exp\big(-\tfrac{1}{2}\|\boldsymbol{x} - \boldsymbol{x}_{\text{target}}\|^2_{\boldsymbol{W}}/\sigma_c^2\big)\,,$$

where $\|\boldsymbol{x} - \boldsymbol{x}_{\text{target}}\|^2_{\boldsymbol{W}}$ is the Mahalanobis distance between $\boldsymbol{x}$ and $\boldsymbol{x}_{\text{target}}$, weighted by $\boldsymbol{W}$, σ_c is a scaling factor, and $\boldsymbol{x}_{\text{target}}$ is a target state.
Policy Improvement. To improve the policy, we use gradient-based Quasi-Newton optimization methods, such as BFGS. The required gradients of the expected long-term cost J^π with respect to the policy parameters are computed analytically, which allows for learning low-level policies with hundreds of parameters [**?**].

1.2.3 Policy Application

This module takes care of applying the learned controller to the (simulated) system. At each time step t, the learned controller π computes the corresponding control signal $\boldsymbol{u}_t$ from the current state $\boldsymbol{x}_t$. An ODE solver is used to determine the corresponding successor state $\boldsymbol{x}_{t+1}$. The module returns a trajectory of state-action pairs, from which the training inputs and targets for the GP model can be extracted.
If the controller is applied to a real system, such as a robot, this module is not necessary. Instead, state estimation and control computation need to be performed on the robot directly [**?**].

1.3 User Interface by Example

```
1  % 0. Initialization
2  settings;                    % load scenario-specific settings
3
4  for jj = 1:J                 % Initial J random rollouts
5    [xx, yy, realCost{jj}, latent{jj}] = ...
6      rollout(gaussian(mu0, S0), struct('maxU',policy.maxU), H, plant, cost);
7    x = [x; xx]; y = [y; yy]; % augment training sets for dynamics model
8  end
9
10 % Controlled learning (N iterations)
11 for j = 1:N
12   % 1. Train (GP) dynamics model
13   trainDynModel;
14
15   % 2. Learn policy
16   learnPolicy;
17
18   % 3. Apply controller to system
19   applyController;
20 end
```

In line 2, a scenario-specific settings script is executed. Here, we have to define the policy structure (e.g., parametrization, torque limits), the cost function, the dynamics model (e.g., definition of training inputs/targets), some details about the system/plant (e.g., sampling frequency) that are needed for the ODE solver, and some general parameters (e.g., prediction horizon, discount factor).
In lines 4–8, we create an initial set of trajectory rollouts by applying random actions to the system, starting from a state $\boldsymbol{x}_0$ sampled from $p(\boldsymbol{x}_0) = \mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{S}_0,)$. The `rollout` function in line 6 takes care of this. For each trajectory, the training inputs and targets are collected in the matrices `x` and `y`, respectively.
In lines 11–20, the three main modules are executed iteratively. First, the GP dynamics model is learned in the `trainDynModel` script, using the current training data `x,y` (line 13). Second, the policy is learned in the `learnPolicy` script, which updates the policy parameters using analytic policy evaluation and policy gradients. The third module, i.e., the application of the learned controller to the system, is encapsulated in the `applyController` script, which also augments the current data set `x,y` for training the GP model with the new trajectory.

1.4 Quick Start

If you want to try it out without diving into the details, navigate to `<pilco_root>/scenarios/cartPole` and execute the `cartPole_learn` script.

Chapter 2

Software Package Overview

This software package implements the PILCO reinforcement learning framework [**?**]. The package contains the following directories

- `base:` Root directory. Contains all other directories.
- `control:` Directory that implements several controllers.
- `doc:` Documention
- `gp:` Everything that has to do with Gaussian processes (training, predictions, sparse GPs etc.)
- `loss:` Several immediate cost functions
- `scenarios:` Different scenarios. Each scenario is packaged in a separate directory with all scenario-specific files
- `util:` Utility files
- `test:` Test functions (derivatives etc.)

2.1 Main Modules

The main modules of the PILCO framework and their interplay are visualized in Figure **??**. Each module is implemented in a separate script and can be found in `<pilco_root>/base`.
Let us have a look at the high-level functionality of the three main modules in Figure **??**:

1. `applyController`
   (a) determine start state
   (b) generate rollout
      i. compute control signal $\pi(\boldsymbol{x}_t)$
      ii. simulate dynamics (or apply control to real robot)
      iii. transition to state $\boldsymbol{x}_{t+1}$
2. `trainDynModel`
3. `learnPolicy`

(a) call gradient-based non-convex optimizer `minimize`: minimize `value` with respect to policy parameters $\boldsymbol{\theta}$

i. `propagated:` compute successor state distribution $p(\boldsymbol{x}_{t+1})$ and gradients $\partial p(\boldsymbol{x}_{t+1})/\partial \boldsymbol{\theta}$ with respect to the policy parameters and gradients $\partial p(\boldsymbol{x}_{t+1})/\partial p(\boldsymbol{x}_t)$ with respect to the previous state distribution $p(\boldsymbol{x}_t)$.

A. trigonometric augmentation of the state distribution $p(\boldsymbol{x}_t)$

B. compute distribution of preliminary (unsquashed) policy $p(\tilde{\pi}(\boldsymbol{x}_t))$

C. compute distribution of squashed (limited-amplitude) policy $p(\pi(\boldsymbol{x}_t)) = p(\boldsymbol{u}_t)$

D. determine successor state distribution $p(\boldsymbol{x}_{t+1})$ using GP prediction (`gp*`)

ii. `cost.fcn:` Scenario-specific function that computes the expected (immediate) cost $\mathbb{E}_{\boldsymbol{x}}[c(\boldsymbol{x})]$ and its partial derivatives $\partial \mathbb{E}_{\boldsymbol{x}}[c(\boldsymbol{x})]/\partial p(\boldsymbol{x})$

2.1.1 `applyController`

```
% 1. Generate trajectory rollout given the current policy
if isfield(plant,'constraint'), HH = maxH; else HH = H; end
[xx, yy, realCost{j+J}, latent{j}] = ...
  rollout(gaussian(mu0, S0), policy, HH, plant, cost);
disp(xx);                              % display states of observed trajectory
x = [x; xx]; y = [y; yy];                                % augment training set
if plotting.verbosity > 0
  if ~ishandle(3); figure(3); else set(0,'CurrentFigure',3); end
  hold on; plot(1:length(realCost{J+j}),realCost{J+j},'r'); drawnow;
end

% 2. Make many rollouts to test the controller quality
if plotting.verbosity > 1
  lat = cell(1,10);
  for i=1:10
    [~,~,~,lat{i}] = rollout(gaussian(mu0, S0), policy, HH, plant, cost);
  end

  if ~ishandle(4); figure(4); else set(0,'CurrentFigure',4); end; clf(4);

  ldyno = length(dyno);
  for i=1:ldyno       % plot the rollouts on top of predicted error bars

    subplot(ceil(ldyno/sqrt(ldyno)),ceil(sqrt(ldyno)),i); hold on;
    errorbar( 0:length(M{j}(i,:))-1, M{j}(i,:), ...
      2*sqrt(squeeze(Sigma{j}(i,i,:))) );
    for ii=1:10
      plot( 0:size(lat{ii}(:,dyno(i)),1)-1, lat{ii}(:,dyno(i)), 'r' );
    end
    plot( 0:size(latent{j}(:,dyno(i)),1)-1, latent{j}(:,dyno(i)),'g');
    axis tight
  end
  drawnow;
end

% 3. Save data
filename = [basename num2str(j) '_H' num2str(H)]; save(filename);
```

The script `applyController` executes the following high-level steps:

1. Generate a trajectory rollout by applying the current policy to the system (lines 1–4). The initial state is sampled from $p(\boldsymbol{x}_0) = \mathcal{N}(\texttt{mu0}, \texttt{S0})$, see line 4. This trajectory is used to augment the GP training set (line 6).

2. (optional) Generate more trajectories with different start states $\boldsymbol{x}_0 \sim p(\boldsymbol{x}_0)$ and plot a sample distribution of the trajectory distribution (lines 12– 33).

3. Save the entire workspace (line 36).

2.1.2 `trainDynModel`

```
1  % 1. Train GP dynamics model
2  Du = length(policy.maxU); Da = length(plant.angi); % no. of ctrl and angles
3  xaug = [x(:,dyno) x(:,end-Du-2*Da+1:end-Du)];     % x augmented with angles
4  dynmodel.inputs = [xaug(:,dyni) x(:,end-Du+1:end)];     % use dyni and ctrl
5  dynmodel.targets = y(:,dyno);
6  dynmodel.targets(:,difi) = dynmodel.targets(:,difi) - x(:,dyno(difi));
7
8  dynmodel = dynmodel.train(dynmodel, plant, trainOpt);  %  train dynamics GP
9
10 % display some hyperparameters
11 Xh = dynmodel.hyp;
12 % noise standard deviations
13 disp(['Learned noise std: ' num2str(exp(Xh(end,:)))]);
14 % signal-to-noise ratios (values > 500 can cause numerical problems)
15 disp(['SNRs             : ' num2str(exp(Xh(end-1,:)-Xh(end,:)))]);
```

The script that takes care of training the GP executes the following high-level steps:

1. Extract states and controls from `x`-matrix (lines 2–3)

2. Define the training inputs and targets of the GP (lines 4–6)

3. Train the GP (line 8)

4. Display GP hyper-parameters, the learned noise hyper-parameters, and the signal-to-noise ratios (lines 10–15). This information is very valuable for debugging purposes.

2.1.3 `learnPolicy`

```
% 1. Update the policy
opt.fh = 1;
[policy.p fX3] = minimize(policy.p, 'value', opt, mu0Sim, S0Sim, ...
  dynmodel, policy, plant, cost, H);

% (optional) Plot overall optimization progress
if exist('plotting', 'var') && isfield(plotting, 'verbosity') ...
    && plotting.verbosity > 1
  if ~ishandle(2); figure(2); else set(0,'CurrentFigure',2); end
  hold on; plot(fX3); drawnow;
  xlabel('line search iteration'); ylabel('function value')
end

```

```
14 % 2. Predict trajectory from p(x0) and compute cost trajectory
15 [M{j} Sigma{j}] = pred(policy, plant, dynmodel, mu0Sim(:,1), S0Sim, H);
16 [fantasy.mean{j} fantasy.std{j}] = ...
17   calcCost(cost, M{j}, Sigma{j}); % predict cost trajectory
18
19 % (optional) Plot predicted immediate costs (as a function of the time steps)
20 if exist('plotting', 'var') && isfield(plotting, 'verbosity') ...
21     && plotting.verbosity > 0
22   if ~ishandle(3); figure(3); else set(0,'CurrentFigure',3); end
23   clf(3); errorbar(0:H,fantasy.mean{j},2*fantasy.std{j}); drawnow;
24   xlabel('time step'); ylabel('immediate cost');
25 end
```

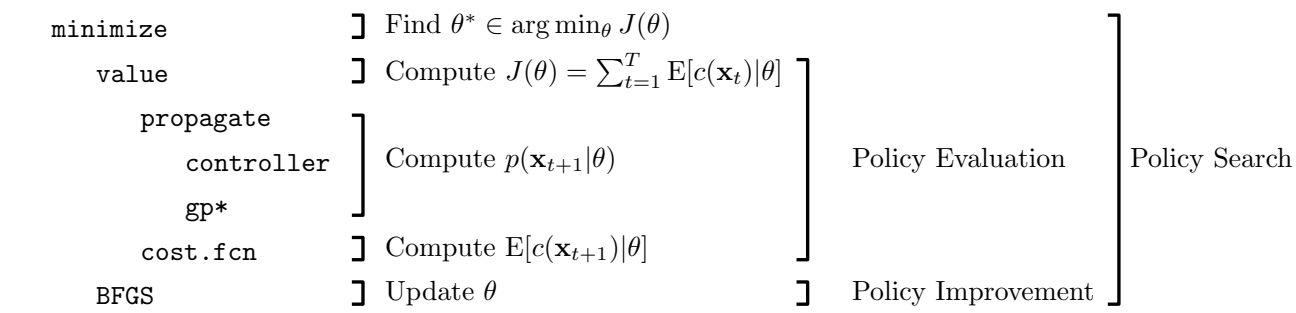

Figure 2.1: Functions being called from `learnPolicy.m` for learning the policy.

1. Learn the policy by calling `minimize`. Figure **??** depicts the functions that are called by `learnPolicy` in order to perform the policy search to find a good parameter set $\boldsymbol{\theta}^*$.
2. (optional) Plot overall optimization progress.
3. Long-term prediction of a state trajectory from $p(\boldsymbol{x}_0)$ using the learned policy (line 15) by calling `pred`. This prediction is equivalent to the last predicted trajectory during policy learning, i.e., the predicted state trajectory that belongs to the the learned controller.
4. The predicted state trajectory is used to compute the corresponding distribution over immediate costs (lines 16–17) by calling `calcCost`.
5. (optional) Plot the predicted immediate cost distribution as a function of the time steps (lines 19–25).

2.2 Working with a Real Robot

When you want to apply PILCO to a learning controller parameters for a real robot, only a few modifications are necessary. As policy learning is not real-time anyway, it does not make too much sense performing it on the robot directly. Therefore, the robot only needs to know about the learned policy, but nothing about the learned dynamics model.
Here is a list of modifications:

- An ODE does not need to be specified for simulating the system.

- All trajectory rollouts are executed directly on the robot.
- The module `applyController` needs to take care of generating a trajectory on the robot.
- For generating a trajectory using the robot, the probably least coding extensive approach is the following:
  1. Learn the dynamics model and the policy.
  2. Save the learned policy parameters in a file.
  3. Transfer the parameters to your robot
  4. Write a controller function in whatever programming language the robot needs.
  5. When the controller is applied, just map the measured state through the policy to obtain the desired control signal.
  6. Save the recorded trajectory in a file and make it available to PILCO and save them in `xx, yy`.

Here is a high-level code-snippet that explains the main steps.

```
% 0. Initialization
settings;                    % load scenario-specific settings

applyController_on_robot;    % collect data from robot

% Controlled learning (N iterations)
for j = 1:N
  % 1. Train (GP) dynamics model
  trainDynModel;

  % 2. Learn policy
  learnPolicy;

  % 3. Apply controller to system
  applyController_on_robot;
end
```

We successfully applied this procedure on different hardware platforms [**?**, **?**].

Chapter 3

Important Function Interfaces

The PILCO software package relies on several high-level functionalities with unified interfaces:

- Predicting with GPs when the test input is Gaussian distributed. We have implemented several versions of GPs (including sparse GPs), which perform these predictions. The generic interface is detailed in the following.
- Controller functions. With a unified interface, it is straightforward to swap between controllers in a learning scenario. We discuss the generic interface in this chapter.
- Cost functions. With this software package, we ship implementations of several cost functions. The interface of them is discussed in this chapter.

3.1 GP Predictions

Table **??** gives an overview of all implemented functions that are related to predicting with GPs at a Gaussian distributed test input $\boldsymbol{x}_* \sim \mathcal{N}(\boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*)$. We assume that the input dimension is D and the

Table 3.1: Overview of functions for GP predictions with Gaussian distributed test inputs.

	`[M S V]`	`[dMdm dSdm dVdm dMds dSds dVds]`	`[dMdP dSdP dVdP]`	sparse	prob. GP
`gp0`	✓	✗	✗	✗	✓
`gp0d`	✓	✓	✗	✗	✓
`gp1`	✓	✗	✗	✓	✓
`gp1d`	✓	✓	✗	✓	✓
`gp2`	✓	✗	✗	✗	✗
`gp2d`	✓	✓	✓	✗	✗

predictive dimension is E. All functions are in the directory `<pilco_root>/gp/`.
The convention in the function name is that a "`d`" indicates that derivatives are computed. For instance, `gp0d` computes the same function values as `gp0`, but it additionally computes some derivatives. We have three different categories of functions for GP predictions:

- `gp0`: The underlying model is a full probabilistic GP model. This model is used for implementing the standard GP dynamics model.

- `gp1`: The underlying model is a sparse GP model. In particular, we use the SPGP/FITC approach by Snelson and Ghahramani [**?**]. This model is used for implementing the GP dynamics when the data set is too large.
- `gp2`: The underlying model is a full "deterministic" GP model. The model differs from the full probabilistic model (`gp0`) by ignoring the posterior uncertainty about the underlying function. The model essentially consists of the mean function only. This makes it functionally equivalent to a radial-basis-function (RBF) network. This kind of model is used for implementing nonlinear controllers.

3.1.1 Input Arguments

For all functions `gp*`, the input arguments are the following:

1. `gpmodel`: Structure containing all relevant information
   - `.hyp`: log-hyper-parameters in a $(D+2) \times E$ matrix (D log-length scales, 1 log-signal-standard deviation, 1 log-noise-standard deviation per predictive dimension)
   - `.inputs`: training inputs in an $n \times D$ matrix
   - `.targets`: training targets in an $n \times E$ matrix
2. `m`: Mean of the state distribution $p(\boldsymbol{x})$, $(D \times 1)$
3. `s`: Covariance matrix of the state distribution $p(\boldsymbol{x})$, $(D \times D)$

3.1.2 Output Arguments

The `gp*` functions can be used to compute the mean and the covariance of the joint distribution $p(\boldsymbol{x}, f(\boldsymbol{x}))$, where $f \sim \mathcal{GP}$ and $\boldsymbol{x} \sim \mathcal{N}(\mathtt{m}, \mathtt{s})$.
All functions `gp*` predict the mean `M` and the covariance `S` of $p(f(\boldsymbol{x}))$ as well as $\mathtt{V}{=}\mathtt{s}^{-1}\mathrm{cov}[\boldsymbol{x}, f(\boldsymbol{x})]$. Note that `gp1*` compute these values using sparse GPs and `gp2*` use only the mean function of the GP, i.e., the posterior uncertainty about f is discarded.
For policy learning, we require from the *dynamics model* the following derivatives:

- `dMdm`: $\partial M/\partial m \in \mathbb{R}^{E \times D}$ The derivative of the mean of the prediction with respect to the mean of the input distribution.
- `dSdm`: $\partial S/\partial m \in \mathbb{R}^{E^2 \times D}$ The derivative of the covariance of the prediction with respect to the mean of the input distribution.
- `dVdm`: $\partial V/\partial m \in \mathbb{R}^{DE \times D}$ The derivative of V with respect to the mean of the input distribution.
- `dMds`: $\partial M/\partial s \in \mathbb{R}^{E \times D^2}$ The derivative of the mean of the prediction with respect to the covariance of the input distribution.
- `dSds`: $\partial S/\partial m \in \mathbb{R}^{E^2 \times D^2}$ The derivative of the covariance of the prediction with respect to the covariance of the input distribution.
- `dVds`: $\partial V/\partial m \in \mathbb{R}^{DE \times D^2}$ The derivative of V with respect to the covariance of the input distribution.

As `gp0d` and `gp1d` are the functions used to propagate uncertainties through a GP dynamics model, they all compute these derivatives, see Table **??**.
When we use `gp2*` as a convenient implementation of an RBF network *controller*, we additionally require the gradients of `M, S, V` with respect to the "parameters" of the GP, which are abbreviated by `P` in Table **??**. These parameters comprise the training inputs, the training targets, and the log-hyper-parameters:

- `dMdP`= $\{\partial M/\partial X, \partial M/\partial y, \partial M/\partial \theta\}$: The derivative of the mean prediction with respect to the training inputs X, the training targets y, and the log-hyper-parameters θ.
- `dSdP`= $\{\partial S/\partial X, \partial S/\partial y, \partial S/\partial \theta\}$: The derivative of the covariance of the prediction with respect to the training inputs X, the training targets y, and the log-hyper-parameters θ.
- `dVdP`= $\{\partial V/\partial X, \partial V/\partial y, \partial V/\partial \theta\}$: The derivative of `V` with respect to the training inputs X, the training targets y, and the log-hyper-parameters θ.

3.2 Controller

The control directory is located at `<pilco_root>/control`. The controllers compute the (unconstrained) control signals $\tilde{\pi}(\boldsymbol{x})$.
The generic function call is as follows, where `controller` is a generic name[1]:

```
function [M, S, V, dMdm, dSdm, dCdm, dMds, dSds, dVds, dMdp, dSdp, dVdp] ...
         = controller(policy, m, s)
```

3.2.1 Interface

Let us explain the interface in more detail

3.2.1.1 Input Arguments

All controllers expect the following inputs

1. `policy`: A struct with the following fields
   - `policy.fcn`: A function handle to `controller`. This is not needed by the controller function itself, but by other functions that call `controller`.
   - `policy.p`: The policy parameters. Everything that is in this field is considered a free parameter and optimized during policy learning.
   - `policy.<>`: Other arguments the controller function requires.
2. `m`: $\mathbb{E}[\boldsymbol{x}] \in \mathbb{R}^D$ The mean of the state distribution $p(\boldsymbol{x})$.
3. `s`: $\mathbb{V}[\boldsymbol{x}] \in \mathbb{R}^{D\times D}$ The covariance matrix of the state distribution $p(\boldsymbol{x})$.

[1] We have implemented two controller functions: `conlin` and `congp`

3.2.1.2 Output Arguments

All controller functions are expected to compute

1. `M:` $\mathbb{E}[\tilde{\pi}(\boldsymbol{x})] \in \mathbb{R}^F$ The mean of the predicted (unconstrained) control signal
2. `S:` $\mathbb{V}[\tilde{\pi}(\boldsymbol{x})] \in \mathbb{R}^{F \times F}$ The covariance matrix of the predicted (unconstrained) control signal
3. `V:` $\mathbb{V}(\boldsymbol{x})^{-1}\text{cov}[\boldsymbol{x}, \tilde{\pi}(\boldsymbol{x})]$ The cross-covariance between the (input) state $\boldsymbol{x}$ and the control signal $\tilde{\pi}(\boldsymbol{x})$, pre-multiplied with $\mathbb{V}(\boldsymbol{x})^{-1}$, the inverse of the covariance matrix of $p(\boldsymbol{x})$. We do not compute $\text{cov}[\boldsymbol{x}, \tilde{\pi}(\boldsymbol{x})]$ because of numerical reasons.
4. Gradients. The gradients of all output arguments with respect to all input arguments are computed:
   - `dMdm:` $\partial M / \partial m \in \mathbb{R}^{F \times D}$ The derivative of the mean of the predicted control with respect to the mean of the state distribution.
   - `dSdm:` $\partial S / \partial m \in \mathbb{R}^{F^2 \times D}$ The derivative of the covariance of the predicted control with respect to the mean of the state distribution.
   - `dVdm:` $\partial V / \partial m \in \mathbb{R}^{DF \times D}$ The derivative of V with respect to the mean of the state distribution.
   - `dMds:` $\partial M / \partial s \in \mathbb{R}^{F \times D^2}$ The derivative of the mean of the predicted control with respect to the covariance of the state distribution.
   - `dSds:` $\partial S / \partial m \in \mathbb{R}^{F^2 \times D^2}$ The derivative of the covariance of the predicted control with respect to the covariance of the state distribution.
   - `dVds:` $\partial V / \partial m \in \mathbb{R}^{DF \times D^2}$ The derivative of V with respect to the covariance of the state distribution.
   - `dMdp:` $\partial M / \partial \theta \in \mathbb{R}^{F \times |\boldsymbol{\theta}|}$ The derivative of the mean of the predicted control with respect to the policy parameters $\boldsymbol{\theta}$.
   - `dSdp:` $\partial S / \partial \theta \in \mathbb{R}^{F^2 \times |\boldsymbol{\theta}|}$ The derivative of the covariance of the predicted control with respect to the policy parameters $\boldsymbol{\theta}$.
   - `dVdp:` $\partial V / \partial \theta \in \mathbb{R}^{DF \times |\boldsymbol{\theta}|}$ The derivative of V with respect to the policy parameters $\boldsymbol{\theta}$.

3.3 Cost Functions

Any (generic) cost function is supposed to compute the expected (immediate) cost $\mathbb{E}[c(\boldsymbol{x})]$ and the corresponding variance $\mathbb{V}[c(\boldsymbol{x})]$ for a Gaussian distributed state $\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{m}, \boldsymbol{S})$.
Cost functions have to be written for each scenario. Example cost functions can be found in `<pilco_root>/scenarios/*`.

3.3.1 Interface for Scenario-specific Cost Functions

```
1 function [L, dLdm, dLds, S] = loss(cost, m, s)
```

Input Arguments

```
cost            cost structure
  .p            parameters that are required to compute the cost,
                e.g., length of pendulum                          [P x  1 ]
  .expl         (optional) exploration parameter
  .target       target state                                      [D x  1 ]
m               mean of state distribution                        [D x  1 ]
s               covariance matrix for the state distribution      [D x  D ]
```

We only listed typical fields of the `cost` structure. It is possible to add more information. `cost.expl` allows for UCB-type exploration, in which case the returned cost `L` should be computed according to

$$L(\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{x}}[c(\boldsymbol{x})] + \kappa\sqrt{\mathbb{V}_{\boldsymbol{x}}[c(\boldsymbol{x})]}\,, \tag{3.1}$$

where κ is an exploration parameter stored in `cost.expl`. Exploration is encouraged for $\kappa < 0$ and discouraged for $\kappa > 0$. By default, exploration is disabled, i.e., `cost.expl=0`. A target state can be passed in via `cost.target`, but could also be hard-coded in the cost function.

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]
S     variance of cost                                            [1 x  1 ]
```

Note that the expected cost $L = \mathbb{E}[c(\boldsymbol{x})]$ can take care of UCB-type exploration, see Equation (**??**). The gradients of `L` with respect to the mean (`dLdm`) and covariance (`dLds`) of the input distribution are required for policy learning.

3.3.2 General Building Blocks

We have implemented some generic building blocks that can be called by the scenario-specific cost functions. In the following, we detail the computation of a saturating cost function `<pilco_root>/loss/lossSat.m` and a quadratic cost function `<pilco_root>/loss/lossQuad.m`.

3.3.2.1 Saturating Cost

`lossSat` computes the expectation and variance of a saturating cost

$$1 - \exp\big(-\tfrac{1}{2}(\boldsymbol{x}-\boldsymbol{z})^\top \boldsymbol{W}(\boldsymbol{x}-\boldsymbol{z})\big) \in [0,1]$$

and their derivatives, where $\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{m}, \boldsymbol{S})$, and a is a normalizing constant. The matrix $\boldsymbol{W}$ is never inverted and plays the role of a precision matrix. Moreover, it is straightforward to eliminate the influence of state variables in the cost function by setting the corresponding values in $\boldsymbol{W}$ to 0.

```
1 function [L, dLdm, dLds, S, dSdm, dSds, C, dCdm, dCds] = lossSat(cost, m, s)
```

Input arguments:

```
cost
  .z:     target state                                          [D x 1]
  .W:     weight matrix                                         [D x D]
m         mean of input distribution                            [D x 1]
s         covariance matrix of input distribution               [D x D]
```

Output arguments:

```
L             expected loss                              [1   x    1 ]
dLdm          derivative of L wrt input mean             [1   x    D ]
dLds          derivative of L wrt input covariance       [1   x   D^2]
S             variance of loss                           [1   x    1 ]
dSdm          derivative of S wrt input mean             [1   x    D ]
dSds          derivative of S wrt input covariance       [1   x   D^2]
C             inv(S) times input-output covariance       [D   x    1 ]
dCdm          derivative of C wrt input mean             [D   x    D ]
dCds          derivative of C wrt input covariance       [D   x   D^2]
```

Implementation

```
2  % some precomputations
3  D = length(m); % get state dimension
4
5  % set some defaults if necessary
6  if isfield(cost,'W'); W = cost.W; else W = eye(D); end
7  if isfield(cost,'z'); z = cost.z; else z = zeros(D,1); end
8
9  SW = s*W;
10 iSpW = W/(eye(D)+SW);
```

In lines 6–7, we check whether the weight matrix $\boldsymbol{W}$ and the state $\boldsymbol{z}$ exist. Their default values are $\boldsymbol{I}$ and $\mathbf{0}$, respectively. Lines 9–10 do some pre-computations of matrices that will be frequently used afterwards.

```
11 % 1. Expected cost
12 L = -exp(-(m-z)'*iSpW*(m-z)/2)/sqrt(det(eye(D)+SW)); % in interval [-1,0]
13
14 % 1a. derivatives of expected cost
15 if nargout > 1
16   dLdm = -L*(m-z)'*iSpW;  % wrt input mean
17   dLds = L*(iSpW*(m-z)*(m-z)'-eye(D))*iSpW/2;  % wrt input covariance matrix
18 end
```

In lines 11–18, we compute the expected cost $L = \mathbb{E}[c(\boldsymbol{x})]$ and its derivatives with respect to the mean and the covariance matrix of the input distribution. A detailed derivation can be found in [**?**]. Note that at the moment, L$\in [-1, 0]$ (line 12).

```
19 % 2. Variance of cost
20 if nargout > 3
21   i2SpW = W/(eye(D)+2*SW);
22   r2 = exp(-(m-z)'*i2SpW*(m-z))/sqrt(det(eye(D)+2*SW));
```

```
23    S = r2 - L^2;
24    if S < 1e-12; S=0; end % for numerical reasons
25  end
26
27  % 2a. derivatives of variance of cost
28  if nargout > 4
29    % wrt input mean
30    dSdm = -2*r2*(m-z)'*i2SpW-2*L*dLdm;
31    % wrt input covariance matrix
32    dSds = r2*(2*i2SpW*(m-z)*(m-z)'-eye(D))*i2SpW-2*L*dLds;
33  end
```

In lines 19–33, we compute the variance $\mathbb{V}[c(\boldsymbol{x})]$ of the cost and its derivatives with respect to the mean and the covariance matrix of the input distribution. A detailed derivation can be found in [**?**]. If the variance $\mathbb{V}[c(\boldsymbol{x})] < 10^{-12}$, we set it to 0 for numerical reasons (line 24).

```
34  % 3. inv(s)*cov(x,L)
35  if nargout > 6
36      t = W*z - iSpW*(SW*z+m);
37      C = L*t;
38      dCdm = t*dLdm - L*iSpW;
39      dCds = -L*(bsxfun(@times,iSpW,permute(t,[3,2,1])) + ...
40                                      bsxfun(@times,permute(iSpW,[1,3,2]),t'))/2;
41      dCds = bsxfun(@times,t,dLds(:)') + reshape(dCds,D,D^2);
42  end
```

If required, lines 34–42 compute $\boldsymbol{S}^{-1}\text{cov}[\boldsymbol{x}, c(\boldsymbol{x})]$ and the corresponding derivatives with respect to the mean and the covariance of the (Gaussian) state distribution $p(\boldsymbol{x})$.

```
43  L = 1+L; % bring cost to the interval [0,1]
```

Line 43 brings the expected cost `L` to the interval $[0, 1]$.

3.3.2.2 Quadratic Cost

`lossQuad` computes the expectation and variance of a quadratic cost

$$c(\boldsymbol{x}) = (\boldsymbol{x} - \boldsymbol{z})^\top \boldsymbol{W}(\boldsymbol{x} - \boldsymbol{z})$$

and their derivatives with respect to the mean and covariance matrix of the (Gaussian) input distribution $p(\boldsymbol{x})$.

```
1  function [L, dLdm, dLds, S, dSdm, dSds, C, dCdm, dCds] = lossQuad(cost, m, S)
```

Input arguments

```
cost
  .z:     target state                                          [D x 1]
  .W:     weight matrix                                         [D x D]
m         mean of input distribution                            [D x 1]
s         covariance matrix of input distribution               [D x D]
```

Output arguments

L	expected loss	[1 x 1]
dLdm	derivative of L wrt input mean	[1 x D]
dLds	derivative of L wrt input covariance	[1 x D^2]
S	variance of loss	[1 x 1]
dSdm	derivative of S wrt input mean	[1 x D]
dSds	derivative of S wrt input covariance	[1 x D^2]
C	inv(S) times input-output covariance	[D x 1]
dCdm	derivative of C wrt input mean	[D x D]
dCds	derivative of C wrt input covariance	[D x D^2]

Implementation

```
2  D = length(m); % get state dimension
3
4  % set some defaults if necessary
5  if isfield(cost,'W'); W = cost.W; else W = eye(D); end
6  if isfield(cost,'z'); z = cost.z; else z = zeros(D,1); end
```

In lines 5–6, we check whether the weight matrix $\boldsymbol{W}$ and the state $\boldsymbol{z}$ exist. Their default values are $\boldsymbol{I}$ and $\boldsymbol{0}$, respectively.

```
7  % 1. expected cost
8  L = S(:)'*W(:) + (z-m)'*W*(z-m);
9
10 % 1a. derivatives of expected cost
11 if nargout > 1
12   dLdm = 2*(m-z)'*W; % wrt input mean
13   dLds = W';         % wrt input covariance matrix
14 end
```

In lines 7–14, we compute the expected cost $L = \mathbb{E}[c(\boldsymbol{x})]$ and its derivatives with respect to the mean and the covariance matrix of the input distribution. A detailed derivation can be found in [**?**].

```
15 % 2. variance of cost
16 if nargout > 3
17   S = trace(W*S*(W + W')*S) + (z-m)'*(W + W')*S*(W + W')*(z-m);
18   if S < 1e-12; S = 0; end % for numerical reasons
19 end
20
21 % 2a. derivatives of variance of cost
22 if nargout > 4
23   % wrt input mean
24   dSdm = -(2*(W+W')*S*(W+W)*(z-m))';
25   % wrt input covariance matrix
26   dSds = W'*S'*(W + W')'+(W + W')'*S'*W' + (W + W')*(z-m)*((W + W')*(z-m))';
27 end
```

In lines 15–27, we compute the variance $\mathbb{V}[c(\boldsymbol{x})]$ of the cost and its derivatives with respect to the mean and the covariance matrix of the input distribution. A detailed derivation can be found in [**?**]. If the variance $\mathbb{V}[c(\boldsymbol{x})] < 10^{-12}$, we set it to 0 for numerical reasons (line 18).

```
28 % 3. inv(s) times IO covariance with derivatives
29 if nargout > 6
30     C = 2*W*(m-z);
31     dCdm = 2*W;
32     dCds = zeros(D,D^2);
33 end
```

If required, lines 28–33 compute $\boldsymbol{S}^{-1}\mathrm{cov}[\boldsymbol{x}, c(\boldsymbol{x})]$ and the corresponding derivatives with respect to the mean and the covariance of the (Gaussian) state distribution $p(\boldsymbol{x})$.

Chapter 4

How to Create Your Own Scenario

In this chapter, we explain in sufficient detail how to set up a new scenario by going step-by-step through the cart-pole scenario, which can be found in `<pilco_root>/scenarios/cartPole`.

4.1 Necessary Files

For each scenario, we need the following set of files, which are specific to this scenario. In the cart-pole case, these files are the following:

- `settings_cp.m`: A file that contains scenario-specific settings and initializations
- `loss_cp.m`: A cost function
- `dynamics_cp.m`: A file that implements the ODE, which governs the dynamics.[1]
- `learn_cp.m`: A file that puts everything together
- (optional) visualization

4.2 ODE Dynamics

In the following, we briefly describe the interface and the functionality of the cart-pole dynamics. The PILCO code assumes by default that the dynamics model is given by an ODE that is solved numerically using ODE45 (see `<pilco_root>/util/simulate.m` for more details).

```
1 function dz = dynamics_cp(t, z, f)
```

Input arguments:

```
t     current time step (called from ODE solver)
z     state                                              [4 x 1]
f     (optional): force f(t)
```

The input arguments are as follows:

- `t`: The current time.

[1] When working with a real robot, this file is not needed.

- `z`: The state. It is assumed that the state $\boldsymbol{z}$ is given as follows:

$$\boldsymbol{z} = [x, \dot{x}, \dot{\theta}, \theta]\,,$$

  where x is the position of the cart (given in m), $\dot{x}$ is the cart's velocity (in m/s), $\dot{\theta}$ is the pendulum's angular velocity in rad/s, and θ is the pendulum's angle in rad. For the angle θ, we chose 0 rad to be the angle when the pendulum hangs downward.

- `f`: The applied force to the cart.

Output arguments:

```
dz    if 3 input arguments:      state derivative wrt time
      if only 2 input arguments: total mechanical energy
```

The function returns either $\dot{\boldsymbol{z}}$, if three input arguments are given, or the total mechanical energy with two input arguments. The total mechanical energy can be used to verify that the system preserves energy (here, the friction coefficient in line 5 needs to be set to 0).

```
2 l = 0.5;  % [m]      length of pendulum
3 m = 0.5;  % [kg]     mass of pendulum
4 M = 0.5;  % [kg]     mass of cart
5 b = 0.1;  % [N/m/s]  coefficient of friction between cart and ground
6 g = 9.82; % [m/s^2]  acceleration of gravity
```

In lines 2–6, the parameters of the cart-pole system are defined: the length of the pendulum `l` (line 2), the mass of the pendulum `m` (line 3), the mass of the cart `M` (line 4), the coefficient of friction between cart and ground `b` (line 5), and the acceleration of gravity `g` (line 6).

```
7  if nargin==3
8    dz = zeros(4,1);
9    dz(1) = z(2);
10   dz(2) = ( 2*m*l*z(3)^2*sin(z(4)) + 3*m*g*sin(z(4))*cos(z(4)) ...
11           + 4*f(t) - 4*b*z(2) )/( 4*(M+m)-3*m*cos(z(4))^2 );
12   dz(3) = (-3*m*l*z(3)^2*sin(z(4))*cos(z(4)) - 6*(M+m)*g*sin(z(4)) ...
13           - 6*(f(t)-b*z(2))*cos(z(4)) )/( 4*l*(m+M)-3*m*l*cos(z(4))^2 );
14   dz(4) = z(3);
15 else
16   dz = (M+m)*z(2)^2/2 + 1/6*m*l^2*z(3)^2 + m*l*(z(2)*z(3)-g)*cos(z(4))/2;
17 end
```

Lines 7–17 compute either $\dot{\boldsymbol{z}}$ (lines 8–14) or the total mechanical energy (line 16). A principled derivation of the system dynamics and the mechanical energy can be found in [**?**].

4.3 Scenario-specific Settings

4.3.1 Adding Paths

```
1 rand('seed',1); randn('seed',1); format short; format compact;
2 % include some paths
```

```
3  try
4    rd = '../../';
5    addpath([rd 'base'],[rd 'util'],[rd 'gp'],[rd 'control'],[rd 'loss']);
6  catch
7  end
```

First, we include (relative) paths to the directories required for learning and initialize the random seed to make the experiments reproducible. The setting of the random seeds (line 1) will cause some warnings in newer versions of MATLAB, but it is backwards compatible to MATLAB 2007.

4.3.2 Indices

```
8  % 1. Define state and important indices
9
10 % 1a. Full state representation (including all augmentations)
11 %
12 %  1  x          cart position
13 %  2  v          cart velocity
14 %  3  dtheta     angular velocity
15 %  4  theta      angle of the pendulum
16 %  5  sin(theta) complex representation ...
17 %  6  cos(theta) of theta
18 %  7  u          force applied to cart
19 %
20
21 % 1b. Important indices
22 % odei  indicies for the ode solver
23 % augi  indicies for variables augmented to the ode variables
24 % dyno  indicies for the output from the dynamics model and indicies to loss
25 % angi  indicies for variables treated as angles (using sin/cos representation)
26 % dyni  indicies for inputs to the dynamics model
27 % poli  indicies for the inputs to the policy
28 % difi  indicies for training targets that are differences (rather than values)
29
30 odei = [1 2 3 4];            % varibles for the ode solver
31 augi = [];                   % variables to be augmented
32 dyno = [1 2 3 4];            % variables to be predicted (and known to loss)
33 angi = [4];                  % angle variables
34 dyni = [1 2 3 5 6];          % variables that serve as inputs to the dynamics GP
35 poli = [1 2 3 5 6];          % variables that serve as inputs to the policy
36 difi = [1 2 3 4];            % variables that are learned via differences
```

We now define important state indices that are required by the code that does the actual learning. We assume that the state is given as

$$\boldsymbol{x} = [x, \dot{x}, \dot{\theta}, \theta]^\top ,$$

where x is the position of the cart, $\dot{x}$ the corresponding velocity, and $\dot{\theta}$ and θ are the angular velocity and the angle of the pendulum.
The ODE-solver requires to know what parts of the state are used for the forward dynamics. These indices are captured by `odei` (line 30).
The predictive dimensions of the dynamics GP model are stored in `dyno` (line 32).

The indices in `angi` (line 33) indicate which variables are angles. We represent these angle variables in the complex plane

$$\theta \mapsto [\sin\theta, \cos\theta]^\top$$

to be able to exploit the wrap-around condition $\theta \equiv \theta + 2k\pi$, $k \in \mathbb{Z}$. With this augmentation, we define the auxiliary state vector, i.e., the state vector augmented with the complex representation of the angle, as

$$\boldsymbol{x} = [x, \dot{x}, \dot{\theta}, \theta, \sin\theta, \cos\theta]^\top . \tag{4.1}$$

The `dyni` indices (line 34) describe which variables from the auxiliary state vector in Equation (**??**) are used as the training inputs for the GP dynamics model. Note that we use the complex representation $[\sin\theta, \cos\theta]$ instead of θ, i.e., we no longer need θ in the inputs of the dynamics GP.
The `poli` indices (line 35) describe which state variables from the auxiliary state vector in Equation (**??**) are used as inputs to the policy.
The `difi` indices (line 36) are a subset of `dyno` and contain the indices of the state variables for which the GP training targets are differences

$$\Delta_t = \boldsymbol{x}_{t+1} - \boldsymbol{x}_t$$

instead of $\boldsymbol{x}_{t+1}$. Using differences as training targets encodes an implicit prior mean function $m(\boldsymbol{x}) = \boldsymbol{x}$. This means that when leaving the training data, the GP predictions do not fall back to 0 but they remain constant. A practical implication is that learning differences Δ_t generalizes better across different parts of the state space. From a learning point of view, training a GP on differences is much simpler than training it on absolute values: The function to be learned does not vary so much, i.e., we do not need so many data points in the end. From a robotics point of view, robot dynamics are typically relative to the current state, they do not so much depend on absolute coordinates.

4.3.3 General Settings

```
37 % 2. Set up the scenario
38 dt = 0.10;                          % [s] sampling time
39 T = 4.0;                            % [s] initial prediction horizon time
40 H = ceil(T/dt);                     % prediction steps (optimization horizon)
41 mu0 = [0 0 0 0]';                   % initial state mean
42 S0 = diag([0.1 0.1 0.1 0.1].^2);    % initial state covariance
43 N = 15;                             % number controller optimizations
44 J = 1;                              % initial J trajectories of length H
45 K = 1;                              % no. of initial states for which we optimize
46 nc = 100;                           % number of controller basis functions
```

`dt` is the sampling time, i.e., $1/dt$ is the sampling frequency.
`T` is the length of the prediction horizon in seconds.
`H`= T/dt is the length of the prediction horizon in time steps.
`mu0` is the mean of the distribution $p(\boldsymbol{x}_0)$ of the initial state. Here, `mu0`= $\mathbf{0}$ encodes that the cart is in the middle of the track (with zero velocity) with the pendulum hanging down (with zero angular velocity).
`S0` is the covariance matrix of $p(\boldsymbol{x}_0)$.
`N` is the number of times the loop in Figure **??** is executed.

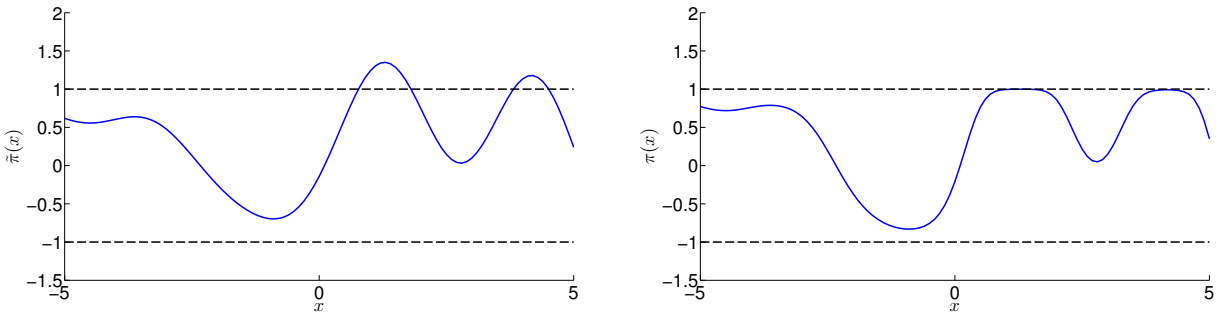

Figure 4.1: Preliminary policy $\tilde{\pi}$ and squashed policy π. The squashing function ensures that the control signals $\boldsymbol{u} = \pi(\boldsymbol{x})$ do not exceed the values $\pm \boldsymbol{u}_{\max}$.

`J` is the number of initial random rollouts, i.e., rollouts with a random policy. These rollouts are used to collect an initial data set for training the first GP dynamics model. Usually, we set this to 1.
`K` is the number of initial states for which the policy is learned. The code can manage initial state distributions of the form

$$p(\boldsymbol{x}_0) \propto \sum_{i=1}^{K} \mathcal{N}(\boldsymbol{\mu}_0^{(i)}, \boldsymbol{\Sigma}_0)\,, \tag{4.2}$$

which corresponds to a distributions with different means $\boldsymbol{\mu}_0^{(i)}$ but shared covariance matrices Σ_0.
`nc` is the number of basis functions of the policy. In this scenario, we use a nonlinear policy of the form

$$\pi(\boldsymbol{x}) = u_{\max}\sigma\tilde{\pi}(\boldsymbol{x}) \tag{4.3}$$

$$\tilde{\pi}(\boldsymbol{x}) = \sum_{i=1}^{\mathtt{nc}} w_i \exp\big(-\tfrac{1}{2}(\boldsymbol{x}-\boldsymbol{c}_i)^\top \boldsymbol{W}(\boldsymbol{x}-\boldsymbol{c}_i)\big)\,, \tag{4.4}$$

where σ is a squashing function, which maps its argument to the interval $[-1, 1]$, $\boldsymbol{c}_i$ are the locations of the Gaussian-shaped basis functions, and $\boldsymbol{W}$ is a (shared) weight matrix.

4.3.4 Plant Structure

```
47 % 3. Plant structure
48 plant.dynamics = @dynamics_cp;                % dynamics ode function
49 plant.noise = diag(ones(1,4)*0.01.^2);        % measurement noise
50 plant.dt = dt;
51 plant.ctrl = @zoh;                            % controler is zero order hold
52 plant.odei = odei;
53 plant.augi = augi;
54 plant.angi = angi;
55 plant.poli = poli;
56 plant.dyno = dyno;
57 plant.dyni = dyni;
58 plant.difi = difi;
59 plant.prop = @propagated;
```

`plant.dynamics` requires a function handle to the function that implements the ODE for simulating the system.
`plant.noise` contains the measurement noise covariance matrix. We assume the noise is zero-mean Gaussian. The noise is added to the (latent) state during a trajectory rollout (see `base/rollout.m`).
`plant.dt` is the sampling time, i.e., $1/dt$ is the sampling frequency.
`plant.ctrl` is the controller to be applied. Here, `@zoh` implements a zero-order-hold controller. Other options are `@foh` (first-order-hold) and `@lag` (first-order-lag). For more information, have a look at `base/simulate.m`.
`plant.odei`–`plant.difi` copy the indices defined earlier into the `plant` structure.
`plant.prop` requires a function handle to the function that computes $p(\boldsymbol{x}_{t+1})$ from $p(\boldsymbol{x}_t)$, i.e., a one-step (probabilistic) prediction. In this software package, we implemented a fairly generic function called `propagated`, which computes the predictive state distribution $p(\boldsymbol{x}_{t+1})$ and the partial derivatives that are required for gradient-based policy search. For details about the gradients, we refer to [**?**].

4.3.5 Policy Structure

```
60 % 4. Policy structure
61 policy.fcn = @(policy,m,s)conCat(@congp,@gSat,policy,m,s);% controller
62                                                          % representation
63 policy.maxU = 10;                                        % max. amplitude of
64                                                          % control
65 [mm ss cc] = gTrig(mu0, S0, plant.angi);                 % represent angles
66 mm = [mu0; mm]; cc = S0*cc; ss = [S0 cc; cc' ss];        % in complex plane
67 policy.p.inputs = gaussian(mm(poli), ss(poli,poli), nc)'; % init. location of
68                                                          % basis functions
69 policy.p.targets = 0.1*randn(nc, length(policy.maxU));   % init. policy targets
70                                                          % (close to zero)
71 policy.p.hyp = log([1 1 1 0.7 0.7 1 0.01])';            % initialize policy
72                                                          % hyper-parameters
```

The policy we use in this example is the nonlinear policy given in Equations (**??**)–(**??**). The policy function handle is stored in `policy.fcn` (line 61). In this particular example, the policy is a concatenation of two functions: the RBF controller (`@congp`, see `ctrl/congp.m`), which is parametrized as the mean of a GP with a squared exponential covariance function, and a squashing function σ (`@gSat`, see `<pilco_root>/util/gSat.m`), defined as

$$\sigma(x) = u_{\max}\frac{9\sin x + \sin(3x)}{8}, \tag{4.5}$$

which is the third-order Fourier series expansion of a trapezoidal wave, normalized to the interval $[-u_{\max}, -u_{\max}]$.
`policy.maxU` (line 63) defines the maximum force value $u_{\max}$ in Equation (**??**). We assume that $u \in [-u_{\max}, -u_{\max}]$.
In lines 65–66, we augment the original state by $[\sin\theta, \cos\theta]$ by means of `gTrig.m`, where the indices of the angles θ are stored in `plant.angi`. We compute a Gaussian approximation to the joint distribution $p(\boldsymbol{x}, \sin\theta, \cos\theta)$. The representation of angles θ by means of $[\sin\theta, \cos\theta]$ avoids discontinuities and automatically takes care of the "wrap-around condition", i.e., $\theta \equiv \theta + 2k\pi, k \in \mathbb{Z}$.
The following lines are used to initialize the policy parameters. The policy parameters are generally stored in `policy.p`. We distinguish between three types of policy parameters:

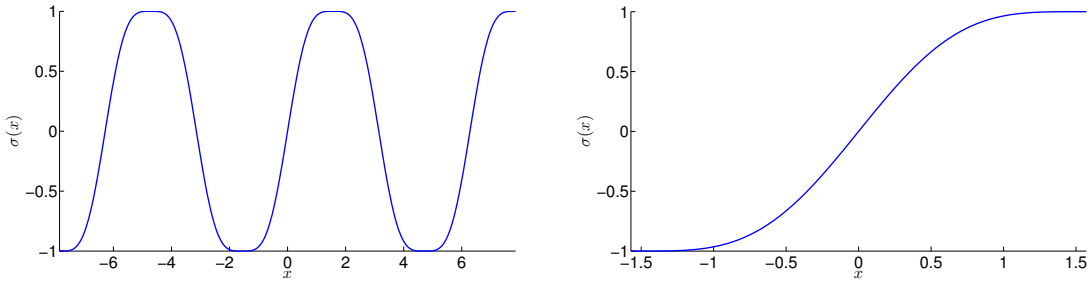

Figure 4.2: Squashing function.

- `policy.p.inputs`: These values play the role of the training inputs of a GP and correspond to the centers $\boldsymbol{c}_i$ of the policy in Equation (**??**). We sample the initial locations of the centers from the initial state distribution $p(\boldsymbol{x}_0)$. We compute this initial state distribution in lines 65–66, where we account for possible angle representations (`plant.angi`) of the state. If `plant.angi` is empty, lines 65–66 do not do anything and `mm`=`mu0` and `ss`=`S0`.

- `policy.p.targets`: These values play the role of GP training targets are initialized to values close to zero.

- `policy.p.hyp`: These values play the role of the GP log-hyper-parameters: log-length-scales, log-signal-standard-deviation, and log-noise-standard-deviation. We initialize the policy hyper-parameters as follows:

  - Length-scales: The length-scales weight the dimensions of the state that are passed to the policy (see `poli`). In our case these are: $x, \dot{x}, \dot{\theta}, \sin\theta, \cos\theta$. We initialize the first three length-scales to 1. These values largely depend on the scale of the input data. In our example, the cart position and velocity are measured in m and m/s, respectively, the angular velocity is measured in rad/s. The last two length-scales scale trigonometric values, i.e., $\sin(\theta)$ and $\cos(\theta)$. Since these trigonometric functions map their argument into the interval $[-1, 1]$, we choose a length-scale of 0.7, which is somewhat smaller than unity.
  - Signal variance: We set the signal variance of the controller $\tilde{\pi}$ to 1. Note that we squash $\tilde{\pi}$ through σ, see Equation (**??**). To exploit the full potential of the squashing function σ, it is sufficient to cover the domain $[-\pi/2, \pi/2]$. Therefore, we initialize the signal variance to 1.
  - Noise variance: The noise variance is only important only important as a relative factor to the signal variance. This ratio essentially determines how smooth the policy is. We initialize the noise variance to 0.01^2.

4.3.6 Cost Function Structure

In the following, we set up a structure for the immediate cost function.

```
73 % 5. Set up the cost structure
74 cost.fcn = @loss_cp;                    % cost function
75 cost.gamma = 1;                         % discount factor
76 cost.p = 0.5;                           % length of pendulum
77 cost.width = 0.25;                      % cost function width
```