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

September 20, 2013
Contents

1 Introduction 1
   1.1 Intended Use ........................................ 1
   1.2 Software Design and Implementation .................. 2
      1.2.1 Model Learning ................................ 2
      1.2.2 Policy Learning ............................... 2
      1.2.3 Policy Application ............................ 3
   1.3 User Interface by Example ............................ 4
   1.4 Quick Start ........................................ 4

2 Software Package Overview 5
   2.1 Main Modules ....................................... 5
      2.1.1 applyController .............................. 6
      2.1.2 trainDynModel ............................... 7
      2.1.3 learnPolicy .................................. 7
   2.2 Working with a Real Robot ........................... 8

3 Important Function Interfaces 10
   3.1 GP Predictions ..................................... 10
      3.1.1 Input Arguments .............................. 11
      3.1.2 Output Arguments ............................ 11
   3.2 Controller ......................................... 12
      3.2.1 Interface .................................... 12
   3.3 Cost Functions .................................... 13
      3.3.1 Interface for Scenario-specific Cost Functions .. 13
      3.3.2 General Building Blocks ....................... 14

4 How to Create Your Own Scenario 19
   4.1 Necessary Files ................................... 19
   4.2 ODE Dynamics ...................................... 19
   4.3 Scenario-specific Settings ......................... 20
      4.3.1 Adding Paths ................................ 20
      4.3.2 Indices ...................................... 21
      4.3.3 General Settings ............................ 22
      4.3.4 Plant Structure .............................. 23
      4.3.5 Policy Structure ............................. 24
      4.3.6 Cost Function Structure ...................... 25
      4.3.7 GP Dynamics Model Structure ................ 27
4.3.8 Optimization Parameters (Policy Learning) ................................. 27
4.3.9 Plotting Parameters ................................................................. 28
4.3.10 Allocating Variables ............................................................... 28
4.4 Cost Function ....................................................................... 28
4.4.1 Interface ................................................................. 28
4.5 Visualization ................................................................. 31
4.6 Main Function ................................................................. 32
4.6.1 Screen Prints and Visualization ........................................... 33

5 Implemented Scenarios ................................................................. 38
5.1 Pendulum Swing-up ............................................................... 38
5.2 Double Pendulum Swing-up with a Single Actuator (Pendubot) .............. 39
5.3 Double Pendulum Swing-up with Two Actuators ................................ 41
5.4 Cart-Pole Swing-up ............................................................... 41
5.5 Cart-Doube Pendulum Swing-up .................................................. 42
5.6 Unicycling ................................................................. 44
5.6.1 Method ................................................................. 45
5.6.2 Wheel FBD ............................................................... 46
5.6.3 Frame FBD ............................................................... 46
5.6.4 Turntable FBD ............................................................. 48
5.6.5 Eliminating Internal Forces .................................................. 48

6 Testing and Debugging ................................................................. 50
6.1 Gradient Checks for the Controller Function ................................... 50
6.1.1 Interface ................................................................. 50
6.2 Gradient Checks for the Cost Function ......................................... 51
6.2.1 Interface ................................................................. 51
6.3 Gradient Checks for the GP Prediction Function ............................... 52
6.3.1 Interface ................................................................. 52
6.4 Gradient Checks for the State Propagation Function ........................... 53
6.4.1 Interface ................................................................. 53
6.5 Gradient Checks for Policy Evaluation ......................................... 53
6.5.1 Interface ................................................................. 53

7 Code and Auto-generated Documentation of the Main Functions ................. 55
7.1 Base Directory ........................................................................ 55
7.1.1 applyController.m .......................................................... 55
7.1.2 propagate.m ............................................................... 56
7.1.3 rollout.m ................................................................. 58
7.1.4 trainDynModel.m .......................................................... 60
7.1.5 value.m ................................................................. 61
7.2 Control Directory ................................................................... 62
7.2.1 concat.m ..................................................................... 62
7.2.2 cong.m ..................................................................... 64
7.2.3 conlin.m ..................................................................... 66
7.3 GP Directory ........................................................................ 68
7.3.1 train.m ..................................................................... 68
7.3.2 hypCurb.m .................................................................... 70
<table>
<thead>
<tr>
<th>Section</th>
<th>File</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3.3</td>
<td>fitc.m</td>
<td>71</td>
</tr>
<tr>
<td>7.3.4</td>
<td>gp0.m</td>
<td>73</td>
</tr>
<tr>
<td>7.3.5</td>
<td>gp1.m</td>
<td>75</td>
</tr>
<tr>
<td>7.3.6</td>
<td>gp2.m</td>
<td>78</td>
</tr>
</tbody>
</table>
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 [3, 4]. 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 [10]. 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 [1, 3], 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 [3, 4]. 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 [2].

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 RLG, CLSquare1, PIQLE2, RL Toolbox3, LibPG4, or RLPy5.

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/
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 [7]. 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 \((x_t, u_t)\), where \(t\) is a time index. Training targets are either successor states \(x_{t+1}\) or differences \(\Delta_t = x_{t+1} - x_t\).

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 [8] 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 \(p(x_1|\pi), \ldots, p(x_T|\pi)\) for a given controller \(\pi\). To do so, we follow the analytic moment-matching approach proposed by [6], and approximate all \(p(x_t|\pi)\) by Gaussians \(\mathcal{N}(x_t|\mu_t, \Sigma_t)\).

**Policy Evaluation.** Once the long-term predictions \(p(x_1|\pi), \ldots, p(x_T|\pi)\) are computed, the expected long-term cost

\[
J^\pi = \sum_{t=1}^T \mathbb{E}[c(x_t)|\pi], \quad p(x_0) = \mathcal{N}(\mu_0, \Sigma_0),
\]

(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

\[
c(x) = 1 - \exp \left(-\frac{1}{2} \|x - x_{\text{target}}\|^2_W/\sigma_c^2\right),
\]

where \(\|x - x_{\text{target}}\|^2_W\) is the Mahalanobis distance between \(x\) and \(x_{\text{target}}\), weighted by \(W\), \(\sigma_c\) is a scaling factor, and \(x_{\text{target}}\) 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 \(J^\pi\) with respect to the policy parameters are computed analytically, which allows for learning low-level policies with hundreds of parameters [3].
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 $\pi$ computes the corresponding control signal $u_t$ from the current state $x_t$. An ODE solver is used to determine the corresponding successor state $x_{t+1}$. 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 [4].
1.3 User Interface by Example

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

for jj = 1:J % Initial J random rollouts
    [xx, yy, realCost{jj}, latent{jj}] = ...
    rollout(gaussian(mu0, S0), struct('maxU', policy.maxU), H, plant, cost);
    x = [x; xx]; y = [y; yy]; % augment training sets for dynamics model
end

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

    % 2. Learn policy
    learnPolicy;

    % 3. Apply controller to system
    applyController;
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 \( x_0 \) sampled from \( p(x_0) = \mathcal{N}(\mu_0, S_0) \). 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 [3]. The package contains the following directories

- **base**: Root directory. Contains all other directories.
- **control**: Directory that implements several controllers.
- **doc**: Documentation
- **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 1.1. 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 1.1:

1. **applyController**
   - (a) determine start state
   - (b) generate rollout
     - i. compute control signal \( \pi(x_t) \)
     - ii. simulate dynamics (or apply control to real robot)
     - iii. transition to state \( x_{t+1} \)

2. **trainDynModel**

3. **learnPolicy**
(a) call gradient-based non-convex optimizer \texttt{minimize}: minimize \texttt{value} with respect to policy parameters \( \theta \)

i. \texttt{propagated}: compute successor state distribution \( p(x_{t+1}) \) and gradients \( \partial p(x_{t+1})/\partial \theta \) with respect to the policy parameters and gradients \( \partial p(x_{t+1})/\partial p(x_t) \) with respect to the previous state distribution \( p(x_t) \).

A. trigonometric augmentation of the state distribution \( p(x_t) \)

B. compute distribution of preliminary (unsquashed) policy \( p(\tilde{\pi}(x_t)) \)

C. compute distribution of squashed (limited-amplitude) policy \( p(\pi(x_t)) = p(u_t) \)

D. determine successor state distribution \( p(x_{t+1}) \) using GP prediction (\texttt{gp*})

ii. \texttt{cost.fcn}: Scenario-specific function that computes the expected (immediate) cost \( E_x[c(x)] \) and its partial derivatives \( \partial E_x[c(x)]/\partial p(x) \)

2.1.1 applyController

\begin{verbatim}
% 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);
end
ldyno = length(dyno);       % plot the rollouts on top of predicted error bars
for i=1:ldyno
    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(ii)),1)-1, lat{ii}(:,dyno(ii)),'r');
    end
    plot(0:size(latent{j}(:,dyno(i)),1)-1, latent{j}(:,dyno(i)),'g');
    axis tight
    drawnow;
end

% 3. Save data
filename = [basename num2str(j) '.H' num2str(H)]; save(filename);
\end{verbatim}

The script \texttt{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 \( p(x_0) = \mathcal{N}(\mu_0, S_0) \), see line 4. This trajectory is used to augment the GP training set (line 6).

2. (optional) Generate more trajectories with different start states \( x_0 \sim p(x_0) \) and plot a sample distribution of the trajectory distribution (lines 12–33).

3. Save the entire workspace (line 36).

2.1.2 trainDynModel

```matlab
% 1. Train GP dynamics model
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 = dynmodel.train(dynmodel, plant, trainOpt); % train dynamics GP

% display some hyperparameters
Xh = dynmodel.hyp;
% noise standard deviations
disp(['Learned noise std: ' num2str(exp(Xh(end,:)))]);
% signal-to-noise ratios (values > 500 can cause numerical problems)
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

```matlab
% 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
```
learnPolicy

minimize
value

propagate
controller
gp*
cost.fcn
BFGS

Find $\theta^* \in \arg \min_{\theta} J(\theta)$
Compute $J(\theta) = \sum_{t=1}^{T} E[c(x_t)|\theta]$ 
Compute $p(x_{t+1}|\theta)$
Compute $E[c(x_{t+1})|\theta]$
Update $\theta$

Policy Evaluation
Policy Search
Policy Improvement

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

1. Learn the policy by calling minimize. Figure 2.1 depicts the functions that are called by learnPolicy in order to perform the policy search to find a good parameter set $\theta^*$.

2. (optional) Plot overall optimization progress.

3. Long-term prediction of a state trajectory from $p(x_0)$ 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 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.

```matlab
% 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 [4, 3].
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 3.1 gives an overview of all implemented functions that are related to predicting with GPs at a Gaussian distributed test input $x_* \sim \mathcal{N}(\mu_*, \Sigma_*)$. We assume that the input dimension is $D$ and the predictive dimension is $E$. All functions are in the directory $<\text{pilco\_root}>/\text{gp}/$.

The convention in the function name is that a “d” indicates that derivatives are computed. For instance, $\text{gp0d}$ computes the same function values as $\text{gp0}$, but it additionally computes some derivatives.

We have three different categories of functions for GP predictions:

- $\text{gp0}$: The underlying model is a full probabilistic GP model. This model is used for implementing the standard GP dynamics model.

<table>
<thead>
<tr>
<th>Function</th>
<th>Sparse</th>
<th>Prob. GP</th>
</tr>
</thead>
<tbody>
<tr>
<td>gp0</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>gp0d</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>gp1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>gp1d</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>gp2</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>gp2d</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
• **gp1**: The underlying model is a sparse GP model. In particular, we use the SPGP/FITC approach by Snelson and Ghahramani [8]. 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 $p(x)$, $(D \times 1)$

3. **s**: Covariance matrix of the state distribution $p(x)$, $(D \times D)$

### 3.1.2 Output Arguments

The gp* functions can be used to compute the mean and the covariance of the joint distribution $p(x, f(x))$, where $f \sim \mathcal{GP}$ and $x \sim \mathcal{N}(m, s)$.

All functions gp* predict the mean $\mathbf{M}$ and the covariance $\mathbf{S}$ of $p(f(x))$ as well as $\mathbf{V} = s^{-1} \text{cov}[x, f(x)]$. 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**: $\partial S / \partial s \in \mathbb{R}^{E^2 \times D^2}$ The derivative of the covariance of the prediction with respect to the covariance of the input distribution.
- **dVds**: $\partial V / \partial s \in \mathbb{R}^{DE \times D^2}$ 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 3.1.

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 3.1. These parameters comprise the training inputs, the training targets, and the log-hyper-parameters:

- $dM/dP = \{\partial M/\partial X, \partial M/\partial y, \partial M/\partial \theta\}$: The derivative of the mean prediction with respect to the training inputs $X$, the training targets $y$, and the log-hyper-parameters $\theta$.
- $dS/dP = \{\partial S/\partial X, \partial S/\partial y, \partial S/\partial \theta\}$: The derivative of the covariance of the prediction with respect to the training inputs $X$, the training targets $y$, and the log-hyper-parameters $\theta$.
- $dV/dP = \{\partial V/\partial X, \partial V/\partial y, \partial V/\partial \theta\}$: The derivative of $V$ with respect to the training inputs $X$, the training targets $y$, and the log-hyper-parameters $\theta$.

### 3.2 Controller

The control directory is located at <pilco_root>/control. The controllers compute the (unconstrained) control signals $\tilde{\pi}(x)$.

The generic function call is as follows, where controller is a generic name:\footnote{We have implemented two controller functions: conlin and congp}:

```latex
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**: $E[x] \in \mathbb{R}^D$ The mean of the state distribution $p(x)$.

3. **s**: $V[x] \in \mathbb{R}^{D \times D}$ The covariance matrix of the state distribution $p(x)$.
3.2.1.2 Output Arguments

All controller functions are expected to compute

1. \( \mathbf{M}: \mathbb{E}[\tilde{\pi}(x)] \in \mathbb{R}^F \) The mean of the predicted (unconstrained) control signal

2. \( \mathbf{S}: \mathbb{V}[\tilde{\pi}(x)] \in \mathbb{R}^{F \times F} \) The covariance matrix of the predicted (unconstrained) control signal

3. \( \mathbf{V}: \mathbb{V}(x) \cdot \text{cov}[x, \tilde{\pi}(x)] \) The cross-covariance between the (input) state \( x \) and the control signal \( \tilde{\pi}(x) \), pre-multiplied with \( \mathbb{V}(x)^{-1} \), the inverse of the covariance matrix of \( p(x) \). We do not compute \( \text{cov}[x, \tilde{\pi}(x)] \) because of numerical reasons.

4. Gradients. The gradients of all output arguments with respect to all input arguments are computed:

   - \( \hat{d}\mathbf{M}/\partial \mathbf{m} \in \mathbb{R}^{F \times D} \) The derivative of the mean of the predicted control with respect to the mean of the state distribution.
   - \( \hat{d}\mathbf{S}/\partial \mathbf{m} \in \mathbb{R}^{F^2 \times D} \) The derivative of the covariance of the predicted control with respect to the mean of the state distribution.
   - \( \hat{d}\mathbf{V}/\partial \mathbf{m} \in \mathbb{R}^{DF \times D} \) The derivative of \( \mathbf{V} \) with respect to the mean of the state distribution.
   - \( \hat{d}\mathbf{M}/\partial \mathbf{s} \in \mathbb{R}^{F \times D^2} \) The derivative of the mean of the predicted control with respect to the covariance of the state distribution.
   - \( \hat{d}\mathbf{S}/\partial \mathbf{s} \in \mathbb{R}^{F^2 \times D^2} \) The derivative of the covariance of the predicted control with respect to the covariance of the state distribution.
   - \( \hat{d}\mathbf{V}/\partial \mathbf{s} \in \mathbb{R}^{DF \times D^2} \) The derivative of \( \mathbf{V} \) with respect to the covariance of the state distribution.
   - \( \hat{d}\mathbf{M}/\partial \mathbf{\theta} \in \mathbb{R}^{F \times |\mathbf{\theta}|} \) The derivative of the mean of the predicted control with respect to the policy parameters \( \mathbf{\theta} \).
   - \( \hat{d}\mathbf{S}/\partial \mathbf{\theta} \in \mathbb{R}^{F^2 \times |\mathbf{\theta}|} \) The derivative of the covariance of the predicted control with respect to the policy parameters \( \mathbf{\theta} \).
   - \( \hat{d}\mathbf{V}/\partial \mathbf{\theta} \in \mathbb{R}^{DF \times |\mathbf{\theta}|} \) The derivative of \( \mathbf{V} \) with respect to the policy parameters \( \mathbf{\theta} \).

3.3 Cost Functions

Any (generic) cost function is supposed to compute the expected (immediate) cost \( \mathbb{E}[c(x)] \) and the corresponding variance \( \mathbb{V}[c(x)] \) for a Gaussian distributed state \( x \sim \mathcal{N}(m, S) \).

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

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

1
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

\[
L(x) = \mathbb{E}_x[c(x)] + \kappa \sqrt{\mathbb{V}_x[c(x)]},
\]

where \( \kappa \) 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]

dCdm, dCds are not used.

Note that the expected cost \( L = \mathbb{E}[c(x)] \) can take care of UCB-type exploration, see Equation (3.1). 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

\[
1 - \exp \left( -\frac{1}{2}(x - z)^\top W(x - z) \right) \in [0,1]
\]

and their derivatives, where \( x \sim \mathcal{N}(m, S) \), and \( a \) is a normalizing constant. The matrix \( W \) 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 \( W \) 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 SW = s*W;
9 iSpW = W/(eye(D)+SW);

In lines 6–7, we check whether the weight matrix W and the state z exist. Their default values are I and 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 \( L = \mathbb{E}[c(x)] \) and its derivatives with respect to the mean and the covariance matrix of the input distribution. A detailed derivation can be found in [2]. 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));
In lines 19–33, we compute the variance $V[c(x)]$ 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 [2]. If the variance $V[c(x)] < 10^{-12}$, we set it to 0 for numerical reasons (line 24).

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

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

### 3.3.2.2 Quadratic Cost

costQuad computes the expectation and variance of a quadratic cost

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

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

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

**Input arguments**

- `cost`: target state
- `W`: weight matrix
- `m`: mean of input distribution
- `S`: covariance matrix of input distribution

$D = \text{size}(S, 2)$
Output arguments

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>expected loss</td>
<td>[1 x 1]</td>
</tr>
<tr>
<td>dLdm</td>
<td>derivative of L wrt input mean</td>
<td>[1 x D]</td>
</tr>
<tr>
<td>dLds</td>
<td>derivative of L wrt input covariance</td>
<td>[1 x D^2]</td>
</tr>
<tr>
<td>S</td>
<td>variance of loss</td>
<td>[1 x 1]</td>
</tr>
<tr>
<td>dSdm</td>
<td>derivative of S wrt input mean</td>
<td>[1 x D]</td>
</tr>
<tr>
<td>dSds</td>
<td>derivative of S wrt input covariance</td>
<td>[1 x D^2]</td>
</tr>
<tr>
<td>C</td>
<td>inv(S) times input-output covariance</td>
<td>[D x 1]</td>
</tr>
<tr>
<td>dCdm</td>
<td>derivative of C wrt input mean</td>
<td>[D x D]</td>
</tr>
<tr>
<td>dCds</td>
<td>derivative of C wrt input covariance</td>
<td>[D x D^2]</td>
</tr>
</tbody>
</table>

Implementation

```matlab
D = length(m);  % get state dimension

% set some defaults if necessary
if isfield(cost, 'W'); W = cost.W; else W = eye(D); end
if isfield(cost, 'z'); z = cost.z; else z = zeros(D,1); end

% 1. expected cost
L = S(:)'*W(:) + (z-m)'*W*(z-m);

% 1a. derivatives of expected cost
if nargout > 1
dLdm = 2*(m-z)'*W;  % wrt input mean
dLds = W';          % wrt input covariance matrix
end

% 2. variance of cost
if nargout > 3
S = trace(W*S*(W + W')*S) + (z-m)'*(W + W')*S*(W + W')*(z-m);
if S < 1e-12; S = 0; end  % for numerical reasons
end

% 2a. derivatives of variance of cost
if nargout > 4
dSdm = -(2*(W+W')*S*(W+W)@(z-m))';
dSds = W'*S'*(W + W')' + (W + W')'*S'*W' + (W + W')*(z-m)*((W + W')*(z-m))';
end
```

In lines 1–14, we compute the expected cost $L = \mathbb{E}[c(x)]$ and its derivatives with respect to the mean and the covariance matrix of the input distribution. A detailed derivation can be found in [2].

In lines 15–27, we compute the variance $\mathbb{V}[c(x)]$ 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 [2]. If the variance $\mathbb{V}[c(x)] < 10^{-12}$, we set it to 0 for numerical reasons (line 18).
If required, lines 28–33 compute $S^{-1} \text{cov}[x, c(x)]$ and the corresponding derivatives with respect to the mean and the covariance of the (Gaussian) state distribution $p(x)$. 
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.
- `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).

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

**Input arguments:**

- `t`: current time step (called from ODE solver)
- `z`: state
- `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 \( z \) is given as follows:

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

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 \( \theta \) is the pendulum’s angle in rad. For the angle \( \theta \), 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 \( \dot{z} \), 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).

\[
l = 0.5; \quad \text{length of pendulum}
\]
\[
m = 0.5; \quad \text{mass of pendulum}
\]
\[
M = 0.5; \quad \text{mass of cart}
\]
\[
b = 0.1; \quad \text{coefficient of friction between cart and ground}
\]
\[
g = 9.82; \quad \text{acceleration of gravity}
\]

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).

\[
\text{if nargin==3}\]
\[
dz = \text{zeros}(4,1);
\]
\[
dz(1) = z(2);
\]
\[
dz(2) = \left( 2 m s l s z(3)^2 \sin(z(4)) + 3 m s g s \sin(z(4)) \cos(z(4)) \right) \ldots
\]
\[
+ 4 s f(t) - 4 b s z(2) ) / ( 4 s (M+m) - 3 m s \cos(z(4))^2 ) ;
\]
\[
dz(3) = (-3 m s l s z(3)^2 \sin(z(4)) \cos(z(4)) - 6 s (M+m) s \sin(z(4)) \ldots
\]
\[
- 6 s (f(t) - b s z(2)) s \cos(z(4)) ) / ( 4 s (m+m) - 3 m s \cos(z(4))^2 ) ;
\]
\[
dz(4) = z(3);
\]
\[
\text{else}\]
\[
dz = (M+m) s z(2)^2/2 + 1/6 m s l s 2 s z(3)^2 + m s (z(2) s z(3) - g) s \cos(z(4)) / 2 ;
\]
\[
\text{end}\]

Lines 7–17 compute either \( \dot{z} \) (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 [2].

### 4.3 Scenario-specific Settings

#### 4.3.1 Adding Paths

\[
\text{rand(’seed’,1); randn(’seed’,1); format short; format compact;}
\]
\[
\% include some paths
\]
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

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

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

where \(x\) is the position of the cart, \(\dot{x}\) the corresponding velocity, and \(\dot{\theta}\) and \(\theta\) 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 \texttt{odei} (line 30).

The predictive dimensions of the dynamics GP model are stored in \texttt{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

\[ x = [x, \dot{x}, \dot{\theta}, \theta, \sin \theta, \cos \theta]^{\top}. \] (4.1)

The dyni indices (line 34) describe which variables from the auxiliary state vector in Equation (4.1) are used as the training inputs for the GP dynamics model. Note that we use the complex representation \([\sin \theta, \cos \theta]\) instead of \(\theta\), i.e., we no longer need \(\theta\) in the inputs of the dynamics GP.

The poli indices (line 35) describe which state variables from the auxiliary state vector in Equation (4.1) 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

\[ \Delta_t = x_{t+1} - x_t \]

instead of \(x_{t+1}\). Using differences as training targets encodes an implicit prior mean function \(m(x) = x\). 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 \(\Delta_t\) 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

% 2. Set up the scenario

\begin{verbatim}
% [s] sampling time
dt = 0.10;
% [s] initial prediction horizon time
T = 4.0;
% prediction steps (optimization horizon)
H = ceil(T/dt);
% initial state mean
mu0 = [0 0 0 0]';
% initial state covariance
S0 = diag([0.1 0.1 0.1 0.1].^2);
% number controller optimizations
N = 15;
% initial J trajectories of length H
J = 1;
% no. of initial states for which we optimize
K = 1;
% number of controller basis functions
nc = 100;
\end{verbatim}

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 \(p(x_0)\) of the initial state. Here, \(mu0=0\) 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 \(p(x_0)\).
N is the number of times the loop in Figure 1.1 is executed.
Figure 4.1: Preliminary policy $\tilde{\pi}$ and squashed policy $\pi$. The squashing function ensures that the control signals $u = \pi(x)$ do not exceed the values $\pm u_{\text{max}}$.

$J$ 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

$$p(x_0) \propto \sum_{i=1}^{K} N(\mu_0^{(i)}, \Sigma_0),$$  \hspace{1cm} (4.2)

which corresponds to a distribution with different means $\mu_0^{(i)}$ but shared covariance matrices $\Sigma_0$.

$nc$ is the number of basis functions of the policy. In this scenario, we use a nonlinear policy of the form

$$\pi(x) = u_{\text{max}} \sigma \tilde{\pi}(x)$$

$$\tilde{\pi}(x) = \sum_{i=1}^{nc} w_i \exp \left( -\frac{1}{2} (x - c_i)^\top W (x - c_i) \right),$$  \hspace{1cm} (4.4)

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

4.3.4 Plant Structure

```matlab
% 3. Plant structure
plant.dynamics = @dynamics_cp;  % dynamics ode function
plant.noise = diag(ones(1,4)*0.01.^2);  % measurement noise
plant.dt = dt;  % controller is zero order hold
plant.odei = odei;
plant.augi = augi;
plant.angi = angi;
plant.poli = poli;
plant.dyno = dyno;
plant.dyni = dyni;
plant.difi = difi;
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., $1/\text{dt}$ is the sampling frequency.

plant.ctrl is the controller to be applied. Here, $\mathtt{@zoh}$ implements a zero-order-hold controller. Other options are $\mathtt{@foh}$ (first-order-hold) and $\mathtt{@lag}$ (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 $p(x_{t+1})$ from $p(x_t)$, 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 $p(x_{t+1})$ and the partial derivatives that are required for gradient-based policy search. For details about the gradients, we refer to [3].

4.3.5 Policy Structure

The policy we use in this example is the nonlinear policy given in Equations (4.3)-(4.4). The policy function handle is stored in \texttt{policy.fcn} (line 61). In this particular example, the policy is a concatenation of two functions: the RBF controller ($\mathtt{@congp}$, see \texttt{ctrl/congp.m}), which is parametrized as the mean of a GP with a squared exponential covariance function, and a squashing function $\sigma$ ($\mathtt{@gSat}$, see \texttt{<pilco_root>/util/gSat.m}), defined as

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

which is the third-order Fourier series expansion of a trapezoidal wave, normalized to the interval $[-u_{\text{max}}, u_{\text{max}}]$.

policy.maxU (line 63) defines the maximum force value $u_{\text{max}}$ in Equation (4.3). We assume that $u \in [-u_{\text{max}}, -u_{\text{max}}]$.

In lines 65–66, we augment the original state by $[\sin \theta, \cos \theta]$ by means of \texttt{gTrig.m}, where the indices of the angles $\theta$ are stored in plant.angi. We compute a Gaussian approximation to the joint distribution $p(x, \sin \theta, \cos \theta)$. The representation of angles $\theta$ 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:
• `policy.p.inputs`: These values play the role of the training inputs of a GP and correspond to the centers \( c_i \) of the policy in Equation (4.4). We sample the initial locations of the centers from the initial state distribution \( p(x_0) \). 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 `policy`). 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 \( \sigma \), see Equation (4.3). To exploit the full potential of the squashing function \( \sigma \), 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 0.01².

### 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
```
In the saturating cost function, see Equation (4.6), \( \text{cost.width} \) determines how far away from the target a cost \( c < 1 \) can be attained.

In line 74, we store the function handle \( @\text{loss.cp} \) in \texttt{cost.fcn}. The function \texttt{loss.cp} implements a saturating cost function (an unnormalized Gaussian subtracted from 1) with spread \( \sigma_c \), i.e.,

\[
    c(x) = 1 - \exp \left( -\frac{1}{2\sigma_c^2} \| x - x_{\text{target}} \|^2 \right) \in [0, 1]
\]  

(4.6)

where \( x_{\text{target}} \) is a target state.

We set the discount factor \texttt{cost.gamma} to 1 (line 75) as we look at a finite-horizon problem.

The following parameters are specific to the cost function:

- \texttt{\( \hat{\text{cost.p} \)}} (line 76) is the length of the pendulum. This length is needed to compute the Euclidean distance of the tip of the pendulum from the desired location in the inverted position.

- \texttt{\( \hat{\text{cost.width} \)}} (line 77) is the spread/width \( \sigma_c \) of the cost function. Looking at the cost function in (4.6) and the target state \( [x, \dot{x}, \dot{\theta}, \theta] = [0, *, *, 2k\pi + \pi], k \in \mathbb{Z} \), the factor 0.25 essentially encodes that the pendulum has to be above horizontal, such that a cost substantially different from 1 is incurred (the length of the pendulum is 0.5 m). Figure 4.3 illustrates a simplified scenario with \( \sigma_c = 1/2 \). For \( c \approx 1 \), it can get difficult to obtain any useful gradients. As a rule of thumb, one can set \( \text{cost.width} = \| \mu_0 - x_{\text{target}} \| / 10 \).

- \texttt{\( \hat{\text{cost.expl} \)}} (line 78) is a UCB-type exploration parameter. Negative values encourage exploration, positive values encourage the policy staying in regions with good predictive performance. We set the value to 0 in order to disable any kind of additional exploration or exploitation.

- \texttt{\( \hat{\text{cost.angle} \)}} (line 79) tells the cost function, which indices of the state are angles. In the cost function, we also represent angles in the complex plane.

- \texttt{\( \hat{\text{cost.target} \)}} (line 80) defines the target state \( x_{\text{target}} \). Here, the target state is defined as the cart being in the middle of the track and the pendulum upright, without any velocity or angular velocity.
4.3.7 GP Dynamics Model Structure

In the following, we set up the structure dynmodel for the GP dynamics model.

```
% 6. Dynamics model structure
dynmodel.fcn = @gp1d;           % function for GP predictions
dynmodel.train = @train;        % function to train dynamics model
dynmodel.induce = zeros(300,0,1); % shared inducing inputs (sparse GP)
trainOpt = [300 500];          % defines the max. number of line searches

% when training the GP dynamics models
% trainOpt(1): full GP,
% trainOpt(2): sparse GP (FITC)
```

We generally assume that the model uses a squared exponential covariance, a Gaussian likelihood, and a zero prior mean. Therefore, these parameters are not explicitly specified here.

dynmodel.fcn (line 82) contains a function handle to gp1d, which can predict with (sparse) GPs at uncertain inputs. If the GP is not sparse but full, gp1d calls gp0d, which implements GP predictions at uncertain inputs with the full GP model.

dynmodel.train (line 83) contains a function handle to train, which is responsible for GP training.

dynmodel.induce (line 84) is optional and tells us when to switch from full GPs to sparse GPs. dynmodel.induce is a tensor of the form $a \times b \times c$, where $a$ is the number of inducing inputs, $b = 0$ tells us that there are no inducing inputs yet, and $c$ is either 1 or the number of predictive dimensions, which corresponds to the number of indices stored in dyno. For $c = 1$, the inducing inputs are shared among all GPs. Otherwise, sets of inducing inputs are separately learned for each predictive dimension.

trainOpt (line 85) contains the number of line searches for GP hyper-parameter training used by the full GP (first number) and the sparse GP (second parameter).

4.3.8 Optimization Parameters (Policy Learning)

In the following lines, we define (optional) parameters for policy learning. Generally, we use an adapted version of `minimize.m`, a non-convex gradient-based optimizer. These optional parameters are stored in a structure opt.

```
% 7. Parameters for policy optimization
opt.length = 150;             % max. number of line searches
opt.MFEPLS = 30;              % max. number of function evaluations
opt.verbosity = 1;            % per line search
                          % verbosity: specifies how much
                          % information is displayed during
                          % policy learning. Options: 0–3
```

opt.length (line 90) sets the maximum number of line searches after which the optimizer returns the best parameter set so far.

opt.MFEPLS (line 91) is the maximum number of function evaluations per line search. Either the line search succeeds by finding a parameter set with a gradient close to 0 or it does not succeed and aborts after opt.MFEPLS many function (and gradient) evaluations.

opt.verbosity (line 93) regulates the verbosity of the optimization procedure. Verbosity ranges from 0 (no information displayed) to 3 (visualize the line search and the computed gradients).

---

2If the size of the training set exceeds $a$, the full GP automatically switches to its sparse approximation.

3http://www.gaussianprocess.org/gpml/code/matlab/util/minimize.m

4This is great for debugging.
4.3.9 Plotting Parameters

plotting.verbosity (line 97) is an easy way of controlling how much information is visualized during policy learning.

4.3.10 Allocating Variables

In lines 100-103, we simply initialize a few array that are used to store data during the learning process.

4.4 Cost Function

In the following, we describe how to define a cost function for the cart-pole swing-up scenario. The cost function is stored in loss_cp.m and implements the cost

\[
c(x) = \frac{1}{\#\text{cost.cw}} \sum_{i=1}^{\#\text{cost.cw}} \left(1 - \exp \left( -\frac{1}{2\sigma_c^{(i)}} \| x - x_{\text{target}} \|^2 \right) \right)
\] (4.7)

which is a generalized version of Equation (4.6). In particular, \text{cost.cw} can be an array of different widths \( \sigma_c^{(i)} \), which is used to compute a cost function \( c(x) \) as a mixture of cost functions with different widths.

The mean and the variance of the cost \( c(x) \) are computed by averaging over the Gaussian state distribution \( p(x) = \mathcal{N}(x|m, S) \) with mean \( m \) and covariance matrix \( S \). Derivatives of the expected cost and the cost variance with respect to the mean and the covariance of the input distribution are computed when desired.

4.4.1 Interface

\begin{verbatim}
function [L, dLdm, dLds, S] = loss_cp(cost, m, s)
\end{verbatim}

Input arguments:

- cost cost structure
- .p length of pendulum
- .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]
S variance of cost [1 x 1]

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

In lines 2–3, we check whether a scaling factor (array) and an exploration parameter exist. Default values are 1 (no scaling) and 0 (no exploration), respectively.

4 % 1. Some precomputations
5 D0 = size(s,2); % state dimension
6 D1 = D0 + 2*length(cost.angle); % state dimension (with sin/cos)
7
8 M = zeros(D1,1); M(1:D0) = m; S = zeros(D1); S(1:D0,1:D0) = s;
9 Mdm = [eye(D0); zeros(D1-D0,D0)]; Sdm = zeros(D1*D1,D0);
10 Mds = zeros(D1,D0*D0); Sds = kron(Mdm,Mdm);

In line 5, the dimension of the state is determined.
In line 6, the dimension of the state is augmented to account for potential angles in the state, which require a representation on the unit circle via \( \sin \theta \) and \( \cos \theta \). Therefore, the (fully) augmented state variable is then given as
\[
j = [x, \dot{x}, \dot{\theta}, \theta, \sin \theta, \cos \theta]^\top.
\] (4.8)

Lines 8–10 initialize the output arguments to 0.
In the following lines, the distance \( x - x_{\text{target}} \) is computed.

11 % 2. Define static penalty as distance from target setpoint
12 ell = cost.p; % pendulum length
13 Q = zeros(D1); Q([1 D0+1],[1 D0+1]) = [1 ell]’*[1 ell]; Q(D0+2,D0+2) = ell^2;

Line 12 stores the pendulum length in \( \ell_1 \).
In line 12, the matrix \( Q \) is computed, such that \((j - j_{\text{target}})^\top Q(j - j_{\text{target}})\) is the squared Euclidean distance between the tip of the pendulum in the current state and the target state. For \( x_{\text{target}} = [0, \pi, \pi, \pi]^\top \), i.e., the pendulum is balanced in the inverted position in the middle of the track, the Euclidean distance is given as
\[
\|x - x_{\text{target}}\|^2 = x^2 + 2\ell \sin \theta + 2l^2 + 2l^2 \cos \theta = (j - j_{\text{target}})^\top Q(j - j_{\text{target}}),
\] (4.9)
\[
Q = \begin{bmatrix}
1 & 0 & 0 & 0 & l & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
l & 0 & 0 & 0 & l^2 & 0 \\
0 & 0 & 0 & 0 & 0 & l^2 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}.
\] (4.10)
Note that at this point only the $Q$-matrix is determined.

This block is only executed if angles are present (the check is performed in line 15). First (line 17), the target state $x_{\text{target}}$ is augmented to

$$
\dot{j}_{\text{target}} = [x_{\text{target}}, \dot{x}_{\text{target}}, \dot{\theta}_{\text{target}}, \dot{\theta}_{\text{target}}, \sin(\theta_{\text{target}}), \cos(\theta_{\text{target}})]^\top.
$$

In line 21, the state distribution $p(x)$ is probabilistically augmented to $p(j)$, where $j$ is given in Equation (4.1). Note that $p(j)$ cannot be computed analytically. Instead, we compute the mean and covariance of $p(j)$ exactly and approximate $p(j)$ by a Gaussian. This probabilistic augmentation and the corresponding derivatives with respect to the mean and covariance of the state distribution are computed by <pilco_root>/util/gTrig.m.

Lines 28–38 compute the derivatives of the mean and covariance of $p(j)$ and the cross-covariance $\text{cov}[x, j]$ with respect to the mean and covariance of $p(x)$ using the chain rule.
\[
\frac{dLds}{dLds + b/sqrt(s2) * (s2dM(:, :) * Mds + s2dS(:, :) * Sds)) / 2};
\]

After all the pre-computations, in lines 40–58, the expected cost for Equation (4.7) is finally computed: For all widths of the cost structure (line 42), we compute the mean and the variance of the saturating cost in Equation (4.6), including the derivatives with respect to the mean and the covariance of \( p(j) \), see line 44. For these computations, the function loss/lossSat.m is called.

Lines 47–48 compute the derivatives of the expected cost and the variance of the cost with respect to the mean and covariance matrix of the state distribution \( p(x) \) by applying the chain rule.

If exploration is desired (line 50), we add \( \kappa \sqrt{V[c]\{c(x)\]} \) to the \( E[c\{c(x)\}] \) to allow for UCB-type exploration, see Equation (3.1)

### 4.5 Visualization

The following lines of code display a sequence of images (video) of a cart-pole trajectory and can be found in `<pilco_root>/scenarios/cartPole/draw_rollout_cp.m`.

At each time step \( r \) of the most recent trajectory (stored in \( xx \)), an image of the current cart-pole state is drawn by repeatedly calling draw_cp. We distinguish between two modes (if-else statement): Either we draw a trajectory after having learned a policy (if statement), or we draw a trajectory from a random rollout, i.e., before a policy is learned (else statement). In the first case, draw_cp also visualizes the long-term predictive means and variances of the tip of the pendulum, which are not given otherwise.

The draw_cp function plots the cart-pole system with reward, applied force, and predictive uncertainty of the tip of the pendulum. We just describe the interface in the following. The code is available at `<pilco_root>/scenarios/cartPole/draw_cp.m`.

```
function draw_cp(x, theta, force, cost, text1, text2, M, S)
```

Input arguments:
4.6 Main Function

The main function executes the following high-level steps.

1. Load scenario-specific setting
2. Create J initial trajectories by applying random controls
3. Controlled learning:
   (a) Train dynamics model
   (b) Learn policy
   (c) Apply policy to system

1 % 1. Initialization
2 clear all; close all;
3 settings_cp;  % load scenario-specific settings
4 basename = 'cartPole_';  % filename used for saving data

Lines 1–4 load scenario-specific settings (line 3) and define a basename for data that is stored throughout the execution.

5 % 2. Initial J random rollouts
6 for jj = 1:J
7   [xx, yy, realCost{jj}, latent{jj}] = ... 
8   rollout(gaussian(mu0, S0), struct('maxU',policy.maxU), H, plant, cost);
9   x = [x; xx]; y = [y; yy];  % augment training sets for dynamics model
10  if plotting.verbosity > 0;  % visualization of trajectory
11    if ~ishandle(1); figure(1); else set(0,'CurrentFigure',1); end; clf(1);
12    draw_rollout_cp;
13  end
14 end
15
16 mu0Sim(odei,:) = mu0; S0Sim(odei.odei) = S0;
17 mu0Sim = mu0Sim(dyno); S0Sim = S0Sim(dyno.dyno);

To kick off learning, we need to create an initial (small) data set that can be learned for learning the first GP dynamics model. For this, we generate J trajectories of length H by applying random control
signals using `<pilco_root>/base/rollout.m`, (lines 7–8). Generally, the training data for the GP is stored in x and y (line 9). If desired, the trajectories of the cart-pole system are visualized (lines 10–13). In lines 17–18, we define variables `mu0Sim` and `S0Sim`, which are used subsequently.

The actual learning happens in lines 19–29, where PILCO performs N (controlled) iterations of dynamics model learning (line 21), policy search (line 22), and controller application to the system (line 23). If desired, the trajectories of the cart-pole system are visualized (lines 25–28).

### 4.6.1 Screen Prints and Visualization

When training the GP dynamics model, a typical feedback in the command window looks as follows:

```plaintext
Train hyper-parameters of full GP ...
GP 1/4
 Initial Function Value 1.853671e+01
  linesearch #    31; value -8.942969e+01
GP 2/4
 Initial Function Value 3.115190e+01
  linesearch #    34; value -4.210842e+01
GP 3/4
 Initial Function Value 8.045295e+01
  linesearch #    30; value 4.742728e+00
GP 4/4
 Initial Function Value -3.771471e+00
  linesearch #    37; value -6.971879e+01
Learned noise std:  0.016818  0.016432  0.04385  0.019182
SNRs :  28.91172  116.4612  112.2714  49.12984
```

In this cart-pole example, we train four GPs (one for each predictive dimension stored in `dyno`). The hyper-parameters are learned after 30–40 line searches. Positive values are generally and indicator that the learned model is not so good at predicting this target dimension. This might be due to sparse data and/or very nonlinear dynamics. At the end, the learned noise standard deviations are displayed, together with the signal-to-noise ratios (SNRs). The SNRs are computed as

\[
SNR = \frac{\sqrt{\sigma_f^2}}{\sqrt{\sigma_{\text{noise}}^2}}
\]
where $\sigma_f^2$ is the variance of the underlying function and $\sigma_{\text{noise}}^2$ is the inferred noise variance. High SNRs ($> 500$) are penalized during GP training in `<pilco_root>/gp/hypCurb.m` to improve numerical stability. If there are SNRs $> 500$, it might be worthwhile adding some more noise to the GP training targets.

Figure 4.4 displays a typical screen output during policy optimization, showing the overall progress of policy learning (top subplot) and the progress of the current line search (bottom subplot). The following lines are simultaneously displayed in the command window to show the current progress of learning:

```
Initial Function Value 3.910902e+01
linesearch #  150;  value 3.316620e+01
```

If `opt.verbosity < 3`, i.e., we are not interested in permanently displaying the gradients as in Figure 4.4, it is possible to display the overall optimization progress at the end of each policy search by setting `plotting.verbosity=2`. The corresponding figure is given in Figure 4.5. This figure visually indicates whether the policy search was close to convergence. If it frequently happens that the curve does not flatten out, it might be worthwhile increasing the value of `opt.length` in the scenario-specific settings file (here: `settings.cp.m`).

Figure 4.4: Typical display during policy learning: The top subplot shows the overall progress of policy learning as a function of the number of line searches. In this example, the initial policy caused a total expected cost $J(\theta)$ of 39.11; the policy after 150 line search searches caused an expected cost of 33.16. The bottom subplot shows the progress of the current line search as a function of the distance in line search direction. The function values (+) and the corresponding gradients are visualized. For more detailed information about the visualization and the verbosity of the output (`opt.verbosity`), we refer to `<pilco_root>/util/minimize.m.`
Figure 4.5: Overall progress of a policy search with 150 line searches.

Figure 4.6 shows the predicted and the incurred immediate cost when applying the policy. Initially, the cost is at unity, which means that the initial state is far from the target area. After around 6 time steps, the predictive uncertainty in the cost increases, which means that the predictive state distribution substantially overlaps with the target area. In Figure 4.6 the reason is that the predictive state increases very quickly during the first time steps.

Figure 4.7 shows two sets of example trajectories for each predictive dimension. One set is generated in the early stages of learning (Figure 4.7(a)), the other one is generated in the final stages of learning (Figure 4.7(b)). The green trajectory is stored in $\mathbf{x}$ and will be used to augment the training data set for the GP model. The red trajectories are generated to (a) indicate whether the green trajectory is a “typical” trajectory or an outlier, (b) show the quality of the long-term predictive state distributions, whose 95% predictive confidence intervals are indicated by the blue error bars, (c) give an intuition how sensitive the currently learned controller is to different initial states $\mathbf{x}_0 \sim p(\mathbf{x}_0)$. As shown in Figure 4.7(a), in the early stages of learning, the controller is very sensitive to the initial conditions, i.e., the controlled trajectories vary a lot. A good controller is robust to the uncertain initial conditions and leads to very similar controlled trajectories as shown in Figure 4.7(b).

Figure 4.8 shows a snapshot of a visualization of a trajectory of the cart-pole system. The cart runs on an (infinite) track, the pendulum is freely swinging. The cross denotes the target location for balancing the tip of the pendulum, and the blue ellipse represents the 95% confidence bound of a $t$-step ahead prediction of the location of the tip of the pendulum when applying the learned policy. We also display the applied force (green bar), whose values $u_{\text{max}}$ and $-u_{\text{max}}$ correspond to a green bar being either full to the right or left side. Here, “full” means at the end of the black line that represents the track. Moreover, we show the incurred immediate reward (negative cost) of the current state of the system. The immediate reward is represented by a yellow bar, whose maximum is at the right end of the black bar. We also display the number of total trials (this includes the random initial trials), the prediction horizon $T$, and the total experience after this trial.
Figure 4.6: Predicted immediate cost (blue error bar) and corresponding incurred cost when applying the policy (red) for each time step of the prediction horizon.

Table 4.1: `plotting.verbosity` Overview.

<table>
<thead>
<tr>
<th><code>plotting.verbosity</code></th>
<th>Figure 4.5</th>
<th>Figure 4.6</th>
<th>Figure 4.7</th>
<th>Figure 4.8</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>plotting.verbosity=0</code></td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
</tr>
<tr>
<td><code>plotting.verbosity=1</code></td>
<td>☒</td>
<td>☑</td>
<td>☒</td>
<td>☑</td>
</tr>
<tr>
<td><code>plotting.verbosity=2</code></td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
</tbody>
</table>
Figure 4.7: Example trajectories. For each predictive dimension (dyno), ten state trajectories are displayed, which occurred when the same policy $\pi(\theta^*)$ is applied to the system. Note that the initial state differs as it is sampled from the start state distribution, i.e., $x_0 \sim p(x_0)$. The green trajectory will be used in the next learning iteration to update the GP training set; the other (red) trajectories are generated to get an impression of the quality of the long-term predictive state distributions, the 95% confidence intervals of which are shown by the blue error bars.

Figure 4.8: Visualization of a trajectory of the cart-pole swing-up during learning. The cart runs on an (infinite) track, the pendulum is freely swinging. The cross denotes the target location for balancing the tip of the pendulum, and the blue ellipse represents the 95% confidence bound of a $t$-step ahead prediction of the location of the tip of the pendulum when applying the learned policy. We also display the applied force (green bar), whose values $u_{\text{max}}$ and $-u_{\text{max}}$ correspond to a green bar being either full to the right or left side. Here, “full” means at the end of the black line that represents the track. Moreover, we show the incurred immediate reward (negative cost) of the current state of the system. The immediate reward is represented by a yellow bar, whose maximum is at the right end of the black bar. We also display the number of total trials (this includes the random initial trials), the prediction horizon $T$, and the total experience after this trial.
Chapter 5

Implemented Scenarios

In the following, we introduce the scenarios that are shipped with this software package, and detail the derivation of the corresponding equations of motion, taken from [2]. All scenarios have their own folder in `<pilco_root>/scenarios/`.

<table>
<thead>
<tr>
<th>Task</th>
<th>State Space</th>
<th>Control Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pendulum</td>
<td>$\mathbb{R}^2$</td>
<td>$\mathbb{R}$</td>
</tr>
<tr>
<td>Pendubot</td>
<td>$\mathbb{R}^4$</td>
<td>$\mathbb{R}$</td>
</tr>
<tr>
<td>Double Pendulum</td>
<td>$\mathbb{R}^4$</td>
<td>$\mathbb{R}^2$</td>
</tr>
<tr>
<td>Cart-Pole</td>
<td>$\mathbb{R}^4$</td>
<td>$\mathbb{R}$</td>
</tr>
<tr>
<td>Cart-Double Pendulum</td>
<td>$\mathbb{R}^6$</td>
<td>$\mathbb{R}$</td>
</tr>
<tr>
<td>Unicycling</td>
<td>$\mathbb{R}^{12}$</td>
<td>$\mathbb{R}^2$</td>
</tr>
</tbody>
</table>

Table 5.1 gives an overview of the corresponding state and control dimensions.

5.1 Pendulum Swing-up

The pendulum shown in Figure 5.1 possesses a mass $m$ and a length $l$. The pendulum angle $\varphi$ is measured anti-clockwise from hanging down. A torque $u$ can be applied to the pendulum. Typical values are: $m = 1\text{ kg}$ and $l = 1\text{ m}$.

The coordinates $x$ and $y$ of the midpoint of the pendulum are

\[
x = \frac{1}{2} l \sin \varphi, \\
y = -\frac{1}{2} l \cos \varphi,
\]

and the squared velocity of the midpoint of the pendulum is

\[
v^2 = \dot{x}^2 + \dot{y}^2 = \frac{1}{4} l^2 \dot{\varphi}^2.
\]

Figure 5.1: Pendulum.
We derive the equations of motion via the system Lagrangian $L$, which is the difference between kinetic energy $T$ and potential energy $V$ and given by

$$L = T - V = \frac{1}{2}mv^2 + \frac{1}{2}I\dot{\phi}^2 + \frac{1}{2}mlg \cos \phi,$$

(5.1)

where $g = 9.82 \text{ m/s}^2$ is the acceleration of gravity and $I = \frac{1}{12}ml^2$ is the moment of inertia of a pendulum around the pendulum midpoint.

The equations of motion can generally be derived from a set of equations defined through

$$\frac{d}{dt} \frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = Q_i,$$

where $Q_i$ are the non-conservative forces and $q_i$ and $\dot{q}_i$ are the state variables of the system. In our case,

$$\frac{\partial L}{\partial \dot{\phi}} = \frac{1}{4}ml^2 \ddot{\phi} + I\dot{\phi}$$

$$\frac{\partial L}{\partial \phi} = -\frac{1}{2}mlg \sin \phi$$

yield

$$\ddot{\phi} \left( \frac{1}{4}ml^2 + I \right) + \frac{1}{2}mlg \sin \phi = u - b\dot{\phi},$$

where $b$ is a friction coefficient. Collecting both variables $z = [\dot{\phi}, \phi]^T$ the equations of motion can be conveniently expressed as two coupled ordinary differential equations

$$\frac{dz}{dt} = \begin{bmatrix} u - bz_1 - \frac{1}{2}mlg \sin z_2 \\ \frac{1}{4}ml^2 + I \\ z_1 \end{bmatrix},$$

which can be simulated numerically.

### 5.2 Double Pendulum Swing-up with a Single Actuator (Pendubot)

The Pendubot in Figure 5.2 is a two-link (mass $m_2$ and $m_3$ and length $l_2$ and $l_3$ respectively), underactuated robot as described by [9]. The first joint exerts torque, but the second joint cannot. The system has four continuous state variables: two joint positions and two joint velocities. The angles of the joints, $\theta_2$ and $\theta_3$, are measured anti-clockwise from upright. An applied external torque $u$ controls the first joint. Typical values are: $m_2 = 0.5 \text{ kg}$, $m_3 = 0.5 \text{ kg}$, $l_2 = 0.6 \text{ m}$, $l_3 = 0.6 \text{ m}$. 
The Cartesian coordinates \( x_2, y_2 \) and \( x_3, y_3 \) of the midpoints of the pendulum elements are

\[
\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} -\frac{1}{2}l_2 \sin \theta_2 \\ \frac{l_2}{2} \cos \theta_2 \end{bmatrix}, \quad \begin{bmatrix} x_3 \\ y_3 \end{bmatrix} = \begin{bmatrix} -l_2 \sin \theta_2 - \frac{1}{2}l_3 \sin \theta_3 \\ l_2 \cos \theta_2 + \frac{1}{2}l_3 \cos \theta_3 \end{bmatrix},
\]

and the squared velocities of the pendulum midpoints are

\[
\begin{align*}
v_2^2 &= x_2^2 + y_2^2 = \frac{1}{4}l_2^2 \dot{\theta}_2^2, \\
v_3^2 &= x_3^2 + y_3^2 = l_2^2 \dot{\theta}_2^2 + \frac{1}{4}l_3^2 \dot{\theta}_3^2 + l_2l_3 \dot{\theta}_2 \dot{\theta}_3 \cos(\theta_2 - \theta_3).
\end{align*}
\]

The system Lagrangian is the difference between the kinematic energy \( T \) and the potential energy \( V \) and given by

\[
L = T - V = \frac{1}{2}m_2 v_2^2 + \frac{1}{2}m_3 v_2^2 + \frac{1}{2}I_2 \dot{\theta}_2^2 + \frac{1}{2}I_3 \dot{\theta}_3^2 - m_2 gy_2 - m_3 gy_3,
\]

where the angular moment of inertia around the pendulum midpoint is \( J = \frac{1}{12}ml^2 \), and \( g = 9.82 \text{m/s}^2 \) is the acceleration of gravity. Using this moment of inertia, we assume that the pendulum is a thin (but rigid) wire. Plugging in the squared velocities (5.3) and (5.4), we obtain

\[
L = \frac{1}{8}m_2 l_2^2 \dot{\theta}_2^2 + \frac{1}{4}m_3 (l_2^2 \dot{\theta}_2^2 + \frac{1}{4}l_3^2 \dot{\theta}_3^2 + l_2l_3 \dot{\theta}_2 \dot{\theta}_3 \cos(\theta_2 - \theta_3))
\]

\[
+ \frac{1}{2}I_2 \dot{\theta}_2^2 + \frac{1}{2}I_3 \dot{\theta}_3^2 - \frac{1}{2}m_2 gl_2 \cos \theta_2 - m_3 g(l_2 \cos \theta_2 + \frac{1}{2}l_3 \cos \theta_3).
\]

The equations of motion are

\[
\frac{d}{dt} \frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = Q_i,
\]

where\( Q_i \) are the non-conservative forces and \( q_i \) and \( \dot{q}_i \) are the state variables of the system. In our case,

\[
\begin{align*}
\frac{\partial L}{\partial \dot{\theta}_2} &= l_2^2 \dot{\theta}_2 (\frac{1}{4}m_2 + m_3) + \frac{1}{4}m_3 l_2 \dot{\theta}_3 \cos(\theta_2 - \theta_3) + I_2 \dot{\theta}_2, \\
\frac{\partial L}{\partial \dot{\theta}_3} &= -\frac{1}{4}m_3 l_2 l_3 \dot{\theta}_2 \dot{\theta}_3 \sin(\theta_2 - \theta_3) + (\frac{1}{2}m_2 + m_3) gl_2 \sin \theta_2, \\
\frac{\partial L}{\partial \theta_2} &= m_3 l_3 (\frac{1}{4}l_3 \dot{\theta}_3 + \frac{1}{2}l_2 \dot{\theta}_2 \cos(\theta_2 - \theta_3)) + I_2 \dot{\theta}_2, \\
\frac{\partial L}{\partial \theta_3} &= \frac{1}{2}m_3 l_3 (l_2 \dot{\theta}_2 \dot{\theta}_3 \sin(\theta_2 - \theta_3) + g \sin \theta_3)
\end{align*}
\]

lead to the equations of motion

\[
\begin{align*}
\dot{u} &= \ddot{\theta}_2 (l_2^2 (\frac{1}{4}m_2 + m_3) + I_2) + \ddot{\theta}_3 \frac{1}{2}m_3 l_2 l_3 \cos(\theta_2 - \theta_3) \\
&+ l_2 (\frac{1}{2}m_3 l_3 \dot{\theta}_3^2 \sin(\theta_2 - \theta_3) - g \sin \theta_3 (\frac{1}{2}m_2 + m_3)) , \\
0 &= \ddot{\theta}_2 \frac{1}{2}l_2 l_3 m_3 \cos(\theta_2 - \theta_3) + \ddot{\theta}_3 (\frac{1}{4}m_3 l_3^2 + I_3) - \frac{1}{2}m_3 l_3 (l_2 \dot{\theta}_2^2 \sin(\theta_2 - \theta_3) + g \sin \theta_3).
\end{align*}
\]

To simulate the system numerically, we solve the linear equation system

\[
\begin{bmatrix} l_2^2 (\frac{1}{4}m_2 + m_3) + I_2 \\ \frac{1}{2}l_2 l_3 m_3 \cos(\theta_2 - \theta_3) \end{bmatrix} \begin{bmatrix} \ddot{\theta}_2 \\ \ddot{\theta}_3 \end{bmatrix} = \begin{bmatrix} c_2 \\ c_3 \end{bmatrix}
\]

for \( \ddot{\theta}_2 \) and \( \ddot{\theta}_3 \), where

\[
\begin{bmatrix} c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} -l_2 (\frac{1}{2}m_3 l_3 \dot{\theta}_3^2 \sin(\theta_2 - \theta_3) - g \sin \theta_2 (\frac{1}{2}m_2 + m_3)) + u \\ \frac{1}{2}m_3 l_3 (l_2 \dot{\theta}_2^2 \sin(\theta_2 - \theta_3) + g \sin \theta_3) \end{bmatrix}.
\]
5.3 Double Pendulum Swing-up with Two Actuators

The dynamics of the double pendulum with two actuators (one at the shoulder, one at the elbow), are derived exactly as described in Section 5.2, with the single modification that we need to take the second control signal into account in the equations of motion

\[ u_1 = \ddot{\theta}_2 \left( \frac{1}{2} m_2 + m_3 \right) + \ddot{\theta}_3 \frac{1}{2} m_3 l_2 \cos(\theta_2 - \theta_3) + l_2 \left( \frac{1}{2} m_3 l_2 \sin(\theta_2 - \theta_3) - g \sin \theta_2 (\frac{1}{2} m_2 + m_3) \right), \]

\[ u_2 = \ddot{\theta}_2 \frac{1}{2} l_2 l_3 m_3 \cos(\theta_2 - \theta_3) + \ddot{\theta}_3 \left( \frac{1}{2} m_3 l_3^2 + I_3 - \frac{1}{2} m_3 l_3 (l_2 \dot{\theta}_2^2 \sin(\theta_2 - \theta_3) + g \sin \theta_3) \right). \]

To simulate the system numerically, we solve the linear equation system

\[
\begin{bmatrix}
\frac{1}{2} l_2^2 + \frac{1}{4} m_2 + m_3 + I_2 \\
\frac{1}{2} l_2 l_3 m_3 \cos(\theta_2 - \theta_3) \\
\frac{1}{2} m_3 l_2 \cos(\theta_2 - \theta_3) + \frac{1}{2} m_3 l_3^2 + I_3
\end{bmatrix}
\begin{bmatrix}
\ddot{\theta}_2 \\
\ddot{\theta}_3
\end{bmatrix} =
\begin{bmatrix}
\dddot{c}_2 \\
\dddot{c}_3
\end{bmatrix}
\]

for \( \ddot{\theta}_2 \) and \( \ddot{\theta}_3 \), where

\[
\begin{bmatrix}
\dddot{c}_2 \\
\dddot{c}_3
\end{bmatrix} =
\begin{bmatrix}
-\ddot{u}_2 \left( \frac{1}{2} m_3 l_2 \ddot{\theta}_2^2 \sin(\theta_2 - \theta_3) - g \sin \theta_2 (\frac{1}{2} m_2 + m_3) \right) + u_1 \\
\frac{1}{2} m_3 l_3 (l_2 \ddot{\theta}_2^2 \sin(\theta_2 - \theta_3) + g \sin \theta_3) + u_2
\end{bmatrix}.
\]

5.4 Cart-Pole Swing-up

The cart-pole system (inverted pendulum) shown in Figure 5.3 consists of a cart with mass \( m_1 \) and an attached pendulum with mass \( m_2 \) and length \( l \), which swings freely in the plane. The pendulum angle \( \theta_2 \) is measured anti-clockwise from hanging down. The cart can move horizontally with an applied external force \( u \) and a parameter \( b \), which describes the friction between cart and ground. Typical values are: \( m_1 = 0.5 \text{ kg}, m_2 = 0.5 \text{ kg}, l = 0.6 \text{ m} \) and \( b = 0.1 \text{ N/m/s} \).

The position of the cart along the track is denoted by \( x_1 \). The coordinates \( x_2 \) and \( y_2 \) of the midpoint of the pendulum are

\[
x_2 = x_1 + \frac{1}{2} l \sin \theta_2, \\
y_2 = -\frac{1}{2} l \cos \theta_2,
\]

and the squared velocity of the cart and the midpoint of the pendulum are

\[
v_1^2 = \dot{x}_1^2 \\
v_2^2 = \dot{x}_2^2 + \dot{y}_2^2 = \dot{x}_1^2 + \frac{1}{4} l^2 \dot{\theta}_2^2 + l \dot{x}_1 \dot{\theta}_2 \cos \theta_2,
\]

respectively. We derive the equations of motion via the system Lagrangian \( L \), which is the difference between kinetic energy \( T \) and potential energy \( V \) and given by

\[
L = T - V = \frac{1}{2} m_1 v_1^2 + \frac{1}{2} m_2 v_2^2 + \frac{1}{4} l^2 \dot{\theta}_2^2 - m_2 g y_2,
\]

where \( g = 9.82 \text{ m/s}^2 \) is the acceleration of gravity and \( I = \frac{1}{12} m l^2 \) is the moment of inertia of a pendulum around the pendulum midpoint. Plugging this value for \( I \) into the system Lagrangian (5.6), we obtain

\[
L = \frac{1}{2} (m_1 + m_2) \dot{x}_1^2 + \frac{1}{6} m_2 l^2 \dot{\theta}_2^2 + \frac{1}{2} m_2 l (\dot{x}_1 \dot{\theta}_2 + g) \cos \theta_2.
\]
The equations of motion can generally be derived from a set of equations defined through
\[
\frac{d}{dt} \frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = Q_i,
\] (5.7)
where \(Q_i\) are the non-conservative forces and \(q_i\) and \(\dot{q}_i\) are the state variables of the system. In our case,
\[
\frac{\partial L}{\partial \dot{x}_1} = (m_1 + m_2)\ddot{x}_1 + \frac{1}{2} m_2 l_2 \dot{\theta}_2 \cos \theta_2,
\]
\[
\frac{\partial L}{\partial x_1} = 0,
\]
\[
\frac{\partial L}{\partial \dot{\theta}_2} = \frac{1}{3} m_2 l_2 \ddot{\theta}_2 + \frac{1}{2} m_2 l \dot{x}_1 \cos \theta_2,
\]
\[
\frac{\partial L}{\partial \theta_2} = -\frac{1}{2} m_2 l (\dot{x}_1 \dot{\theta}_2 + g),
\]
lead to the equations of motion
\[
(m_1 + m_2)\ddot{x}_1 + \frac{1}{2} m_2 l_2 \ddot{\theta}_2 \cos \theta_2 - \frac{1}{2} m_2 l_2 \dot{\theta}_2^2 \sin \theta_2 = u - b \dot{x}_1,
\]
\[
2\dot{\theta}_2 + 3\ddot{x}_1 \cos \theta_2 + 3g \sin \theta_2 = 0.
\]
Collecting the four variables \(z = [x_1, \dot{x}_1, \dot{\theta}_2, \theta_2]^{T}\) the equations of motion can be conveniently expressed as four coupled ordinary differential equations
\[
\frac{dz}{dt} = \begin{bmatrix}
  z_2 \\
  2m_2 l_2 z_3^2 \sin z_4 + 3m_2 g \sin z_4 \cos z_4 + 4u - 4b z_2 \\
  4(m_1 + m_2) - 3m_2 \cos^2 z_4 \\
  -3m_2 l_2^2 \sin z_4 \cos z_4 - 6(m_1 + m_2) g \sin z_4 - 6(u - b z_2) \cos z_4 \\
  4l(m_1 + m_2) - 3m_2 l \cos^2 z_4 \\
  z_3
\end{bmatrix},
\]
which can be simulated numerically.

## 5.5 Cart-Double Pendulum Swing-up

The cart-double pendulum dynamic system (see Figure 5.4) consists of a cart with mass \(m_1\) and an attached double pendulum with masses \(m_2\) and \(m_3\) and lengths \(l_2\) and \(l_3\) for the two links, respectively. The double pendulum swings freely in the plane. The angles of the pendulum, \(\theta_2\) and \(\theta_3\), are measured anti-clockwise from upright. The cart can move horizontally, with an applied external force \(u\) and the coefficient of friction \(b\). Typical values are: \(m_1 = 0.5\) kg, \(m_2 = 0.5\) kg, \(m_3 = 0.5\) kg \(l_2 = 0.6\) m, \(l_3 = 0.6\) m, and \(b = 0.1\) Ns/m.

The coordinates, \(x_2, y_2\) and \(x_3, y_3\) of the midpoint of the pendulum elements are
\[
\begin{bmatrix}
  x_2 \\
  y_2
\end{bmatrix} = \begin{bmatrix}
  x_1 - \frac{1}{2} l_2 \sin \theta_2 \\
  \frac{1}{2} l_2 \cos \theta_2
\end{bmatrix},
\]
\[
\begin{bmatrix}
  x_3 \\
  y_3
\end{bmatrix} = \begin{bmatrix}
  x_1 - l_2 \sin \theta_2 - \frac{1}{2} l_3 \sin \theta_3 \\
  y_3 = l_2 \cos \theta_2 + \frac{1}{2} l_3 \cos \theta_3
\end{bmatrix}. 
\]
The squared velocities of the cart and the pendulum midpoints are
\[ v_1^2 = \dot{x}_1^2, \]
\[ v_2^2 = \dot{x}_2^2 + \dot{y}_2^2 = \dot{x}_2^2 + l_2\dot{x}_1 \dot{\theta}_2 \cos \theta_2 + \frac{1}{4} l_2^2 \dot{\theta}_2^2, \]
\[ v_3^2 = \dot{x}_3^2 + \dot{y}_3^2 = \dot{x}_3^2 + \dot{I}_3^2 \dot{\theta}_3^2 + \frac{1}{4} l_3^2 \dot{\theta}_3^2 - 2l_2\dot{x}_1 \dot{\theta}_2 \cos \theta_2 - l_3\dot{x}_1 \dot{\theta}_3 \cos \theta_3 + l_2 l_3 \dot{\theta}_2 \dot{\theta}_3 \cos (\theta_2 - \theta_3). \]

The system Lagrangian is the difference between the kinetic energy \( T \) and the potential energy \( V \) and given by
\[
L = T - V = \frac{1}{2} m_1 v_1^2 + \frac{1}{2} m_2 v_2^2 + \frac{1}{2} m_3 v_3^2 + \frac{1}{2} I_2 \dot{\theta}_2^2 + \frac{1}{2} I_3 \dot{\theta}_3^2 - m_2 g y_2 - m_3 g y_3 \\
= \frac{1}{2} (m_1 + m_2 + m_3) \dot{x}_1^2 - \frac{1}{2} m_2 l_2 \dot{x}_2 \cos (\theta_2) - \frac{1}{2} m_3 (2l_2 \dot{x}_2 \cos (\theta_2) + l_3 \dot{x}_1 \dot{\theta}_3 \cos (\theta_3)) \\
+ \frac{1}{2} m_2 l_2^2 \ddot{\theta}_2 + \frac{1}{2} I_2 \ddot{\theta}_2 + \frac{1}{2} m_3 (l_2^2 \ddot{\theta}_2 + \frac{1}{4} l_3^2 \ddot{\theta}_3 + l_2 l_3 \dot{\theta}_2 \dot{\theta}_3 \cos (\theta_2 - \theta_3)) + \frac{1}{2} I_3 \ddot{\theta}_3 \\
- \frac{1}{2} m_2 g l_2 \cos (\theta_2) - m_3 g (l_2 \cos (\theta_2) + \frac{1}{2} l_3 \cos (\theta_3)).
\]

The angular moment of inertia \( I_j, j = 2, 3 \) around the pendulum midpoint is \( I_j = \frac{1}{12} m_j l_j^2 \), and \( g = 9.82 \text{ m/s}^2 \) is the acceleration of gravity. These moments inertia imply the assumption that the pendulums are thin (but rigid) wires.

The equations of motion are
\[
\frac{d}{dt} \frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = Q_i, \quad (5.8)
\]
where \( Q_i \) are the non-conservative forces. We obtain the partial derivatives
\[
\frac{\partial L}{\partial \dot{x}_1} = (m_1 + m_2 + m_3) \ddot{x}_1 - (\frac{1}{2} m_2 + m_3) l_2 \ddot{\theta}_2 \cos \theta_2 - \frac{1}{2} m_3 l_3 \dot{\theta}_3 \cos \theta_3, \\
\frac{\partial L}{\partial \dot{x}_1} = 0, \\
\frac{\partial L}{\partial \dot{\theta}_2} = (m_3 l_2^2 + \frac{1}{4} m_2 l_2^2 + I_2) \ddot{\theta}_2 - (\frac{1}{2} m_2 + m_3) l_2 \dot{x}_1 \cos \theta_2 + \frac{1}{2} m_3 l_3 \dot{\theta}_3 \cos (\theta_2 - \theta_3), \\
\frac{\partial L}{\partial \dot{\theta}_2} = (\frac{1}{2} m_2 + m_3) l_2 (\dot{x}_1 \dot{\theta}_2 + g) \sin \theta_2 - \frac{1}{2} m_3 l_2 \dot{\theta}_3 \dot{\theta}_3 \sin (\theta_2 - \theta_3), \\
\frac{\partial L}{\partial \dot{\theta}_3} = m_3 l_3 [-\frac{1}{2} \dot{x}_1 \cos \theta_3 + \frac{1}{2} l_2 \dot{\theta}_2 \cos (\theta_2 - \theta_3) + \frac{1}{2} l_3 \dot{\theta}_3] + I_3 \dot{\theta}_3, \\
\frac{\partial L}{\partial \dot{\theta}_3} = \frac{1}{2} m_3 l_3 (\dot{x}_1 \dot{\theta}_3 + g) \sin \theta_3 + l_2 \dot{\theta}_2 \dot{\theta}_3 \sin (\theta_2 - \theta_3)]
\]
leading to the equations of motion
\[
(m_1 + m_2 + m_3) \ddot{x}_1 + \frac{1}{2} m_2 + m_3) l_2 (\ddot{\theta}_2 \sin \theta_2 - \ddot{\theta}_2 \cos \theta_2) \\
+ \frac{1}{2} m_3 l_3 (\dot{\theta}_3^2 \sin \theta_3 - \theta_3 \cos \theta_3) = u - b \dot{x}_1 \\
(m_3 l_2^2 + I_2 + \frac{1}{4} m_2 l_2^2) \ddot{\theta}_2 - (\frac{1}{2} m_2 + m_3) l_2 (\dot{x}_1 \cos \theta_2 + g \sin \theta_2) \\
+ \frac{1}{2} m_3 l_3 [\dot{\theta}_3 \cos (\theta_2 - \theta_3) + \dot{\theta}_3^2 \sin (\theta_2 - \theta_3)] = 0 \\
(\frac{1}{2} m_2 l_2^2 + I_3) \ddot{\theta}_3 - \frac{1}{2} m_3 l_3 (\dot{x}_1 \cos \theta_3 + g \sin \theta_3) \\
+ \frac{1}{2} m_3 l_2 \dot{\theta}_2 \cos (\theta_2 - \theta_3) - \frac{1}{2} \theta_2 \dot{\theta}_2 \dot{\theta}_3 \sin (\theta_2 - \theta_3)] = 0
\]

These three linear equations in \((\dot{x}_1, \dot{\theta}_2, \dot{\theta}_3)\) can be rewritten as the linear equation system
\[
\begin{bmatrix}
(m_1 + m_2 + m_3) & -\frac{1}{2} (m_2 + 2m_3) l_2 \cos \theta_2 & -\frac{1}{2} m_3 l_3 \cos \theta_3 \\
-\frac{1}{2} (m_2 + m_3) l_2 \cos \theta_2 & m_3 l_2^2 + I_2 + \frac{1}{4} m_2 l_2^2 & \frac{1}{2} m_3 l_3 \cos (\theta_2 - \theta_3) \\
-\frac{1}{2} m_3 l_3 \cos \theta_3 & \frac{1}{2} m_3 l_3 \cos (\theta_2 - \theta_3) & \frac{1}{4} m_2 l_2^2 + I_3
\end{bmatrix}
\begin{bmatrix}
\dot{x}_1 \\
\dot{\theta}_2 \\
\dot{\theta}_3
\end{bmatrix}
= \begin{bmatrix}
c_1 \\
c_2 \\
c_3
\end{bmatrix},
\]

43
5.6 Unicycling

A sketch of the unicycle system is shown in Figure 5.5. The equations of motion that govern the unicycle were derived in the MSc thesis by Forster [5]. We shall provide a sketch of the full derivation here, in which we follow the steps taken by Forster [5], Section 3.3.

This linear equation system can be solved for $\dot{x}_1, \ddot{\theta}_2, \ddot{\theta}_3$ and used for numerical simulation.
Table 5.2: Physical constants used for the simulated robotic unicycle. The coordinate systems defined by the $i$, $j$ and $k$ vectors are shown in Figure 5.7.

### 5.6.1 Method

The robotic unicycle is shown in Figure 5.6 with global coordinate system defined by the orthonormal vectors $i$, $j$ and $k$. The spatial position of the unicycle is fully defined by the pitch angle $\phi$, roll angle $\theta$, yaw angle $\psi$, wheel angle $\phi_w$, turntable angle $\psi_t$ and location of the global origin with respect to the body-centred coordinate system $(x_c, y_c)$. We chose the state vector to be $x = [\phi, \dot{\phi}, \theta, \dot{\theta}, \psi, \dot{\psi}, \phi_w, \dot{\phi}_w, x_c, y_c]^T \in \mathbb{R}^{10}$ where we exclude $\phi_w$ and $\psi_t$ since they clearly have no effect on the dynamics. The action vector $u$ is made up of a wheel motor torque $u_w$ and a turntable motor torque $u_t$.

Let us start with the coordinates $(x_c, y_c)$. These are centred on the point of contact with the floor and define the location of the global origin. The coordinate $x_c$ lies parallel to the current direction of travel and $y_c$ is orthogonal to it. These coordinates evolve according to

\[
\dot{x}_c = r_w \dot{\phi}_w \cos \psi \\
\dot{y}_c = r_w \dot{\phi}_w \sin \psi
\]

where $r_w$ is the wheel radius. The full unicycle model was obtained by analysing the wheel, frame and turntable as individual Free Body Diagrams (FBDs), as depicted in Figure 5.7. Linear momentum and moment of momentum for each FBD were then resolved to yield six scalar equations for each free body. The internal forces were then eliminated to yield five independent scalar equations which govern the evolution of the angular states. A description of the physical constants of the system along with the values we use in this thesis are given in Table 5.2.
5.6.2 Wheel FBD

The wheel coordinate system is defined by the orthonormal vectors $i_w, j_w$ and $k_w$ as shown in Figure 5.7(a). We begin by noting that the angular velocity of the wheel coordinate system is $\Omega_w = \dot{\psi} k_w + \dot{\theta} j_w$. Now noting that the angular velocity of the wheel only differs in the $k_w$ direction and assuming no slip between wheel and floor we have expressions for the velocity and angular velocity of the wheel

$$v_w = -(\omega_w \times r_w i_w)$$

$$\omega_w = \Omega_w + \dot{\phi}_w k_w$$

From these expressions we can derive the acceleration of the wheel $\dot{v}_w$ and the rate of change of angular momentum $\dot{h}_w$ as

$$\dot{v}_w = \frac{\partial v_w}{\partial t} + (\Omega_w \times v_w)$$

$$\dot{h}_w = A_w \frac{\partial \omega_w[1]}{\partial t} i_w + A_w \frac{\partial \omega_w[2]}{\partial t} j_w + C_w \frac{\partial \omega_w[3]}{\partial t} k_w + (\Omega_w \times h_w)$$

where angular momentum in the wheel frame of reference is $h_w = [A_w; A_w; C_w] \circ \omega_w$. Now we consider the forces acting on the wheel free body. These are given by the unknown quantities: axle force $F_w$, axle torque $Q_w$ & reaction force $R$ and the known quantities: wheel weight $W_w$ & friction torque $T$. These forces and moments are shown in the right-hand plot of Figure 5.7(a). Note that we actually know the component of the axle torque $Q_w$ in the $k_w$ direction as it is given by the reaction of the wheel motor on the wheel itself $u_w$. Resolving the rate of change of linear momentum and the rate of change of angular momentum around the center of mass leads to

$$m_w \dot{v}_w = R + F_w + W_w$$

$$\dot{h}_w = (r_w i_w \times R) + Q_w + T$$

5.6.3 Frame FBD

The frame coordinate system is defined by the orthonormal vectors $i_f, j_f$ and $k_f = k_w$ as shown in Figure 5.7(b). In this case, the angular velocity of the frame $\omega_f$ is given by the angular velocity of the wheel plus an additional spin about the wheel axis and the velocity of the frame $v_f$ is given by the velocity of the wheel plus an additional rotation about the wheel centre

$$v_f = v_w - (\omega_f \times r_f i_f)$$

$$\omega_f = \Omega_w + \dot{\phi}_f k_f$$

As before, we can now derive the acceleration of the frame $\dot{v}_f$ and the rate of change of angular momentum $\dot{h}_f$ as

$$\dot{v}_f = \frac{\partial v_f}{\partial t} + (\Omega_f \times v_f)$$

$$\dot{h}_f = A_f \frac{\partial \omega_f[1]}{\partial t} i_f + B_f \frac{\partial \omega_f[2]}{\partial t} j_f + C_f \frac{\partial \omega_f[3]}{\partial t} k_f + (\Omega_f \times h_f)$$

where angular momentum of the frame in this frame of reference is $h_f = [A_f; B_f; C_f] \circ \omega_f$. The forces and moments acting on the frame are shown on the right in Figure 5.7(b). They consist of the known
Figure 5.7: Free body diagrams of the wheel, frame and turntable of the unicycle. The model has the angular state variables pitch $\phi$, roll $\theta$, yaw $\psi$, wheel angle $\phi_w$ and turntable angle $\psi_t$. The vectors $\hat{i}, \hat{j}$ and $\hat{k}$ define the global fixed frame of reference. The centres of mass for each component are shown by the black dots. Forces and moments are shown on the right, with unknown quantities as dashed lines.
frame weight $W_f$ and the unknown: wheel axle force $F_f = -F_w$, wheel axle torque $Q_f = -Q_w$, turntable axle force $G_f$ & turntable axle torque $P_f$. But again we know that the dimension of $P_f$ acting along $i_f$ is given by the reaction of the frame to the turntable motor $u_t$. So resolving the rate of change of linear momentum and the rate of change of angular momentum around the centre of mass gives us

$$m_f \dot{v}_f = F_f + G_f + W_f$$
$$\dot{h}_f = (r_f i_f \times F_f) - (r_f i_f \times G_f) + Q_f + P_f$$

### 5.6.4 Turntable FBD

Finally, the turntable coordinate system is defined by the orthonormal vectors $i_t = k_f$, $j_t = j_f$ and $k_t = -i_f$ as shown in Figure 5.7(c). The velocity of the turntable centre $v_t$ is equal to the velocity of the wheel plus an additional term caused by rotation about the wheel centre, while the angular velocity $\omega_t$ differs from $\Omega_t = \Omega_f$ only along $k_t$

$$v_t = v_w + (\Omega_t \times r_k)$$
$$\omega_t = \Omega_t + \psi_t k_t$$

Again, we derive the acceleration of the frame $\ddot{v}_t$ and the rate of change of angular momentum $\dot{h}_t$ as

$$\ddot{v}_t = \frac{\partial v_t}{\partial t} + (\Omega_t \times v_t)$$
$$\dot{h}_t = A_1 \frac{\partial \omega_t}{\partial t} i_t + A_1 \frac{\partial \omega_t}{\partial t} j_t + C_1 \frac{\partial \omega_t}{\partial t} k_t + (\Omega_t \times h_t)$$

where $h_t = [A_1; A_1; C_1] \circ \omega_t$. The forces and moments acting on the turntable lead to the last of our equations

$$m_t \dot{v}_t = G_t + W_t$$
$$\dot{h}_t = P_t$$

### 5.6.5 Eliminating Internal Forces

We now have 18 kinematic relationships given by Equations (5.11)–(5.16) which govern the dynamics of the unicycle. These can be reduced to five expressions by eliminating the 13 scalar unknowns found in the unknown internal forces, $F$ and $G$, the unknown reaction force $R$ and the partially unknown torques $Q$ and $P$. The first can be obtained from Equation (5.16) and noting that the component of $P_f$ about $k_1$ is the reaction to the motor torque $u_t$

$$\dot{h}_t^T k_t = u_t$$

The second can be obtained by first making use of the relationships $G_f = -G_t$ & $P_f = -P_t$ and then rearranging Equation (5.13), Equation (5.15) and Equation (5.16) to get

$$F_f = m_f \dot{v}_f + m_f \dot{v}_t - W_t - W_f$$
$$G_f = W_t - m_f \dot{v}_t$$
$$P_f = -\dot{h}_t$$
Substituting these into Equation (5.14) and noting that $Q_w = -Q_f$ gives us

$$Q_w = -\dot{h}_t - \dot{h}_t - (r_i \dot{i}_f \times (W_f + W_t - m_i \dot{v}_f - m_t \dot{v}_t)) - (r_i \dot{i}_f \times (W_t - m_t \dot{v}_t))$$

Once again, the component of $Q_w$ about the wheel axis is equal to the torque provided by the wheel motor $u_w$, therefore $Q_w^\top k_w = u_w$ and we have our second expression

$$-\left(\dot{h}_t + \dot{h}_t + (r_i \dot{i}_f \times (W_f + W_t - m_i \dot{v}_f - m_t \dot{v}_t)) + (r_i \dot{i}_f \times (W_t - m_t \dot{v}_t))\right)^\top k_w = u_w \quad (5.18)$$

Finally, Equation (5.11) can be combined with our expression for $F_f = -F_w$ to find the reaction force at the base

$$R = m_w \dot{v}_w + m_f \dot{v}_f + m_t \dot{v}_t - W_w - W_f - W_t$$

and can be substituted into Equation (5.12) to obtain the following three relationships

$$\dot{h}_w = \left(r_i \dot{i}_w \times (m_w \dot{v}_w + m_f \dot{v}_f + m_t \dot{v}_t - W_w - W_f - W_t)\right)\quad (5.19)$$

$$-\left(\dot{h}_t + \dot{h}_t + (r_i \dot{i}_f \times (W_f + W_t - m_i \dot{v}_f - m_t \dot{v}_t)) + (r_i \dot{i}_f \times (W_t - m_t \dot{v}_t))\right) + T$$

The five expressions contained in Equations (5.17)–(5.19) form the foundation for the model we use. The only unknowns remaining in these relationships are the states: $\phi, \dot{\phi}, \theta, \dot{\theta}, \psi, \dot{\psi}, \phi_w, \dot{\phi}_w, \psi_t$. The equations were then rearranged by Forster, through use of the Symbolic Toolbox in MATLAB, to isolate the five acceleration terms. We use this mapping along with Equations (5.9)–(5.10) to perform simulations of the unicycle.
A crucial ingredient that is required for PILCO’s success is the computation of some gradients. Every important function needs to compute some sort of function value and the derivative of this function value with respect to some of the input values.

We have included some relatively generic\(^1\) gradient test functions in `<pilco_root>/test/`. In all functions, the analytic gradients are compared with finite-difference approximations (see `<pilco_root>/test/checkgrad.m`). Usually, the test functions can be called with the variables in the MATLAB workspace during the execution of PILCO.

### 6.1 Gradient Checks for the Controller Function

`conT` checks gradients of the policy (controller) functions.

#### 6.1.1 Interface

```matlab
function [dd dy dh] = conT(deriv, policy, m, s, delta)
```

**Input arguments:**

- `deriv`: desired derivative. options:
  - (i) ‘dMdm’ - derivative of the mean of the predicted control wrt the mean of the input distribution
  - (ii) ‘dMds’ - derivative of the mean of the predicted control wrt the variance of the input distribution
  - (iii) ‘dMdp’ - derivative of the mean of the predicted control wrt the controller parameters
  - (iv) ‘dSdm’ - derivative of the variance of the predicted control wrt the mean of the input distribution
  - (v) ‘dSds’ - derivative of the variance of the predicted control wrt the variance of the input distribution
  - (vi) ‘dSdp’ - derivative of the variance of the predicted control wrt the controller parameters

\(^1\)The interfaces of the most important functions, e.g., controllers, cost functions, GP predictions, are unified.
(vii) 'dCdm' - derivative of \( \text{inv}(s) \)\((\text{covariance of the input and the predicted control}) \) wrt the mean of the input distribution

(viii) 'dCds' - derivative of \( \text{inv}(s) \)\((\text{covariance of the input and the predicted control}) \) wrt the variance of the input distribution

(ix) 'dCdp' - derivative of \( \text{inv}(s) \)\((\text{covariance of the input and the predicted control}) \) wrt the controller parameters

\[
\begin{align*}
\text{policy} & \quad \text{policy structure} \\
\text{.fcn} & \quad \text{function handle to policy} \\
\text{<>} & \quad \text{other fields that are passed on to the policy}
\end{align*}
\]

\[
\begin{align*}
m & \quad \text{mean of the input distribution} \\
s & \quad \text{covariance of the input distribution} \\
\delta & \quad \text{(optional) finite difference parameter. Default: } 1e^{-4}
\end{align*}
\]

Output arguments:

\[
\begin{align*}
dd & \quad \text{relative error of analytical vs. finite difference gradient} \\
dy & \quad \text{analytical gradient} \\
dh & \quad \text{finite difference gradient}
\end{align*}
\]

### 6.2 Gradient Checks for the Cost Function

\( \text{lossT} \) checks gradients of the (immediate) cost functions, which are specific to each scenario.

#### 6.2.1 Interface

\begin{verbatim}
function [dd dy dh] = lossT(deriv, policy, m, s, delta)
\end{verbatim}

Input arguments:

\[
\begin{align*}
\text{deriv} & \quad \text{desired derivative. options:} \\
(i) & \quad 'dMd\text{m}' - \text{derivative of the mean of the predicted cost} \\
& \quad \text{wrt the mean of the input distribution} \\
(ii) & \quad 'dMd\text{s}' - \text{derivative of the mean of the predicted cost} \\
& \quad \text{wrt the variance of the input distribution} \\
(iii) & \quad 'dSd\text{m}' - \text{derivative of the variance of the predicted cost} \\
& \quad \text{wrt the mean of the input distribution} \\
(iv) & \quad 'dSd\text{s}' - \text{derivative of the variance of the predicted cost} \\
& \quad \text{wrt the variance of the input distribution} \\
(v) & \quad 'dCdm' - \text{derivative of } \text{inv}(s)\text{*(covariance of the input and the predicted cost}) \text{ wrt the mean of the input distribution} \\
(vi) & \quad 'dCds' - \text{derivative of } \text{inv}(s)\text{*(covariance of the input and the predicted cost}) \text{ wrt the variance of the input distribution}
\end{align*}
\]

\[
\begin{align*}
\text{cost} & \quad \text{cost structure} \\
\text{.fcn} & \quad \text{function handle to cost} \\
\text{<>} & \quad \text{other fields that are passed on to the cost} \\
m & \quad \text{mean of the input distribution} \\
s & \quad \text{covariance of the input distribution} \\
\delta & \quad \text{(optional) finite difference parameter. Default: } 1e^{-4}
\end{align*}
\]
Output arguments:

dd relative error of analytical vs. finite difference gradient
dy analytical gradient
dh finite difference gradient

6.3 Gradient Checks for the GP Prediction Function

\( \text{gpT} \) checks gradients of the functions that implement the GP prediction at an uncertain test input, i.e., all \( \text{gp}^* \) functions. These functions can be found in \(<\text{pilco\_root}>/\text{gp}</\text{pilco\_root}>\).

6.3.1 Interface

```matlab
function [dd dy dh] = gpT(deriv, gp, m, s, delta)
```

Input arguments:

- **deriv** desired derivative. options:
  1. 'dMdm' - derivative of the mean of the GP prediction wrt the mean of the input distribution
  2. 'dMds' - derivative of the mean of the GP prediction wrt the variance of the input distribution
  3. 'dMdp' - derivative of the mean of the GP prediction wrt the GP parameters
  4. 'dSdm' - derivative of the variance of the GP prediction wrt the mean of the input distribution
  5. 'dSds' - derivative of the variance of the GP prediction wrt the variance of the input distribution
  6. 'dSdp' - derivative of the variance of the GP prediction wrt the GP parameters
  7. 'dVdm' - derivative of \( \text{inv}(s)^*(\text{covariance of the input and the GP prediction}) \) wrt the mean of the input distribution
  8. 'dVds' - derivative of \( \text{inv}(s)^*(\text{covariance of the input and the GP prediction}) \) wrt the variance of the input distribution
  9. 'dVdp' - derivative of \( \text{inv}(s)^*(\text{covariance of the input and the GP prediction}) \) wrt the GP parameters

- **gp** GP structure
  - .fcn function handle to the GP function used for predictions at uncertain inputs
  - .<> other fields that are passed on to the GP function

- **m** mean of the input distribution

- **s** covariance of the input distribution

- **delta** (optional) finite difference parameter. Default: \( 1e^{-4} \)

Output arguments:
6.4 Gradient Checks for the State Propagation Function

\texttt{propagateT} checks gradients of the function \texttt{propagated}, which implements state propagation \( p(x_t) \rightarrow p(x_{t+1}) \). This function can be found in \texttt{<pilco_root>/base/}.

6.4.1 Interface

\begin{verbatim}
[dd dy dh] = propagateT(deriv, plant, dynmodel, policy, m, s, delta)
\end{verbatim}

Input arguments:

- \texttt{m}: mean of the state distribution at time \( t \) \([D \times 1]\)
- \texttt{s}: covariance of the state distribution at time \( t \) \([D \times D]\)
- \texttt{plant}: plant structure
- \texttt{dynmodel}: dynamics model structure
- \texttt{policy}: policy structure

Output arguments:

- \texttt{Mnext}: predicted mean at time \( t+1 \) \([E \times 1]\)
- \texttt{Snext}: predicted covariance at time \( t+1 \) \([E \times E]\)
- \texttt{dMdm}: output mean wrt input mean \([E \times D]\)
- \texttt{dMds}: output mean wrt input covariance matrix \([E \times D \times D]\)
- \texttt{dSdm}: output covariance matrix wrt input mean \([E\times E \times D]\)
- \texttt{dSds}: output cov wrt input cov \([E\times E \times D \times D]\)
- \texttt{dMdp}: output mean wrt policy parameters \([E \times P]\)
- \texttt{dSdp}: output covariance matrix wrt policy parameters \([E \times E \times P]\)

where \( P \) is the number of policy parameters.

6.5 Gradient Checks for Policy Evaluation

\texttt{valueT} checks the overall gradients \( \partial J(\theta) / \partial \theta \) of the expected long-term cost \( J \) with respect all policy parameters \( \theta \). Policy evaluation, i.e., computing \( J \), and the corresponding gradients are implemented in \texttt{value.m} to be found in \texttt{<pilco_root>/base/}.

6.5.1 Interface

\begin{verbatim}
[d dy dh] = valueT(p, delta, m, s, dynmodel, policy, plant, cost, H)
\end{verbatim}

Input arguments:
p  policy parameters (can be a structure)
    .<>  fields that contain the policy parameters (nothing else)
m  mean of the input distribution
s  covariance of the input distribution
dynmodel GP dynamics model (structure)
policy policy structure
plant plant structure
cost  cost structure
H  prediction horizon
delta (optional) finite difference parameter. Default: 1e-4

Output arguments:

dd  relative error of analytical vs. finite difference gradient
dy  analytical gradient
dh  finite difference gradient
Chapter 7

Code and Auto-generated Documentation of the Main Functions

In the following, we include the auto-generated documentation of the most important functions of the PILCO learning framework. If you change the documentation in the *.m files, please run `<pilco_root>/doc/generate_docs.m` to update the rest of this chapter. HTML files will be generated as well and can be found in `<pilco_root>/doc/html/`.

7.1 Base Directory

7.1.1 applyController.m

Summary: Script to apply the learned controller to a (simulated) system

Copyright (C) 2008-2013 by Marc Deisenroth, Andrew McHutchon, Joe Hall, and Carl Edward Rasmussen.

Last modified: 2013-06-04

High-Level Steps

1. Generate a single trajectory rollout by applying the controller
2. Generate many rollouts for testing the performance of the controller
3. Save the data

Code

```matlab
% 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);
end
ldyno = length(dyno);
for i=1:ldyno
    subplot(cei 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);

7.1.2 propagate.m

Summary: Propagate the state distribution one time step forward.

[Mnext, Snext] = propagate(m, s, plant, dynmodel, policy)

Input arguments:

m          mean of the state distribution at time t          [D x 1]
s          covariance of the state distribution at time t      [D x D]
plant      plant structure
dynmodel   dynamics model structure
policy      policy structure

Output arguments:

Mnext       mean of the successor state at time t+1          [E x 1]
Snext       covariance of the successor state at time t+1      [E x E]

Copyright (C) 2008-2013 by Marc Deisenroth, Andrew McHutchon, Joe Hall, Henrik Ohlsson, and Carl Edward Rasmussen.

Last modified: 2013-01-23
High-Level Steps

1. Augment state distribution with trigonometric functions
2. Compute distribution of the control signal
3. Compute dynamics-GP prediction
4. Compute distribution of the next state

function [Mnext, Snext] = propagate(m, s, plant, dynmodel, policy)

Code

% extract important indices from structures
ang = plant.angi; % angular indices
poli = plant.poli; % policy indices
dyni = plant.dyni; % dynamics-model indices
difi = plant.difi; % state indices where the model was trained on differences

D0 = length(m); % size of the input mean
D1 = D0 + 2*length(ang); % length after mapping all angles to sin/cos
D2 = D1 + length(policy.maxu); % length after computing control signal
D3 = D2 + D0; % length after predicting
M = zeros(D3, 1); M(1:D0) = m; S = zeros(D3); S(1:D0, 1:D0) = s; % init M and S

% 1) Augment state distribution with trigonometric functions
i = 1:D0; j = 1:D0; k = D0+1:D1;
[M(k), S(k,k) C] = gTrig(M(i), S(i,i), angi);
q = [S(j,j)*C; S(j,k) = q; S(k,j) = q';

% 2) Compute distribution of the control signal
i = poli; j = 1:D1; k = D1+1:D2;
[M(k) S(k,k) C] = policy.fcn(policy, mm(i), ss(i,i));
q = [S(j,j)*C; S(j,k) = q; S(k,j) = q';

% 3) Compute dynamics-GP prediction
[n = D0,D2; j = 1:D2;
if isfield(dynmodel, 'sub'), Nf = length(dynmodel.sub); else Nf = 1; end
for n=1:Nf % potentially multiple dynamics models
[j] = sliceModel(dynmodel, n,i,D1,D2,33); j = setdiff(j,k);
[M(k), S(k,k), C] = dyn.fcn(dyn, M(i), S(i,i));
q = [S(j,j)*C; S(j,k) = q; S(k,j) = q';

% 4) Compute distribution of the next state
P = [zeros(D0,D2) eye(D0)]; P(difi,difi) = eye(length(difi));
Mnext = P*M; Snext = P*S*P'; Snext = (Snext+Snext')/2;
7.1.3  rollout.m

Summary: Generate a state trajectory using an ODE solver (and any additional dynamics) from a particular initial state by applying either a particular policy or random actions.

function [x y L latent] = rollout(start, policy, H, plant, cost)

Input arguments:

start vector containing initial states (without controls)  [nX x 1]
policy policy structure
  .fcn policy function
  .p parameter structure (if empty: use random actions)
  .maxU vector of control input saturation values  [nU x 1]
H rollout horizon in steps
plant the dynamical system structure
  .subplant (opt) additional discrete-time dynamics
  .augment (opt) augment state using a known mapping
  .constraint (opt) stop rollout if violated
  .poli indices for states passed to the policy
  .dyno indices for states passed to cost
  .odei indices for states passed to the ode solver
  .subi (opt) indices for states passed to subplant function
  .augi (opt) indices for states passed to augment function

cost cost structure

Output arguments:

x matrix of observed states  [H x nX+nU]
y matrix of corresponding observed successor states  [H x nX ]
L cost incurred at each time step  [ 1 x H ]
latent matrix of latent states  [H+1 x nX ]

Copyright (C) 2008-2013 by Marc Deisenroth, Andrew McHutchon, Joe Hall, and Carl Edward Rasmussen.
High-Level Steps

1. Compute control signal \( u \) from state \( x \): either apply policy or random actions
2. Simulate the true dynamics for one time step using the current pair \((x,u)\)
3. Check whether any constraints are violated (stop if true)
4. Apply random noise to the successor state
5. Compute cost (optional)
6. Repeat until end of horizon

```matlab
function [x y L latent] = rollout(start, policy, H, plant, cost)

if isfield(plant,'augment'), augi = plant.augi; % sort out indices!
else plant.augin = inline('[]'); augi = []; end
if isfield(plant,'subplant'), subi = plant.subi;
else plant.subplant = inline('[]',1); subi = []; end
odei = plant.odei; poli = plant.poli; dyno = plant.dyno; angi = plant.angi;
simi = sort([odei subi]);
nX = length(simi)+length(augi); nU = length(policy.maxU); nA = length(angi);

state(simi) = start; state(augi) = plant.augment(state); % initializations
x = zeros(H+1, nX+2*nA);
x(1,simi) = start' + randn(size(simi))*chol(plant.noise);
x(1,augi) = plant.augment(x(1,:));
u = zeros(H, nU); latent = zeros(H+1, size(state,2)+nU);
y = zeros(H, nX); L = zeros(1, H); next = zeros(1,length(simi));

for i = 1:H % generate trajectory
s = x(i,dyno)'; sa = gTrig(s, zeros(length(s)), angi); s = [s; sa];
x(1:end-2*nA+1:end) = s(end-2*nA+1:end);

% 1. Apply policy ... or random actions ________________________________
if isfield(policy, 'fcn')
    u(i,:) = policy.fcn(policy,s(poli),zeros(length(poli)));
else
    u(i,:) = policy.maxU.*2*rand(1,nU)-1;
end
latent(i,:) = [state u(i,:)]; % latent state

% 2. Simulate dynamics ________________________________________________
next(odei) = simulate(state(odei), u(i,:), plant);
next(subi) = plant.subplant(state, u(i,:));

% 3. Stop rollout if constraints violated ______________________________
if isfield(plant,'constraint') &amp; plant.constraint(next(odei))
    H = 1-1;
    fprintf('state constraints violated...n');
    break;
```

Last modification: 2013-05-21
% 4. Augment state and randomize
state(sim) = next(sim); state(augi) = plant.augment(state);
x(i+1,sim) = state(sim) + randn(size(sim))*chol(plant.noise);
x(i+1,augi) = plant.augment(x(i+1,:));

% 5. Compute Cost
if nargout > 2
    L(i) = cost.fcn(cost, state(dyno)', zeros(length(dyno)));
end
end

y = x(2:H+1,1:nX); x = [x(1:H,:) u(1:H,:)];
latent(H+1, 1:nX) = state; latent = latent(1:H+1,:); L = L(1,1:H);

7.1.4 trainDynModel.m

Summary: Script to learn the dynamics model

Copyright (C) 2008-2013 by Marc Deisenroth, Andrew McHutchon, Joe Hall, and Carl Edward Rasmussen.

Last modification: 2013-05-20

High-Level Steps
1. Extract states and controls from x-matrix
2. Define the training inputs and targets of the GP
3. Train the GP

Code

% 1. Train GP dynamics model
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(:,dymi) x(:,end-Du+1:end)]; % use dyni and ctrl
dynmodel.targets = y(:,dyno);
dynmodel.targets(:,difi) = dynmodel.targets(:,difi) - x(:,dyno(difi));
dynmodel = dynmodel.train(dynmodel, plant, trainOpt); % train dynamics GP

% display some hyperparameters
Xh = dynmodel.hyp;
% noise standard deviations
disp(['Learned noise std: ' num2str(exp(Xh(end,:)))]);
% signal-to-noise ratios (values > 500 can cause numerical problems)
disp(['SNRs: ' num2str(exp(Xh(end-1,:)-Xh(end,:)))]);
7.1.5  value.m

Summary: Compute expected (discounted) cumulative cost for a given (set of) initial state distributions

```matlab
function [J, dJdp] = value(p, m0, S0, dynmodel, policy, plant, cost, H)
```

Input arguments:

- `p` policy parameters chosen by minimize
- `policy` policy structure
- `.fcn` function which implements the policy
- `p` parameters passed to the policy
- `m0` matrix (D by k) of initial state means
- `S0` covariance matrix (D by D) for initial state
- `dynmodel` dynamics model structure
- `plant` plant structure
- `cost` cost function structure
- `.fcn` function handle to the cost
- `gamma` discount factor
- `H` length of prediction horizon

Output arguments:

- `J` expected cumulative (discounted) cost
- `dJdp` (optional) derivative of J wrt the policy parameters

Copyright (C) 2008-2013 by Marc Deisenroth, Andrew McHutchon, Joe Hall, and Carl Edward Rasmussen.

Last modification: 2013-03-21

High-Level Steps

1. Compute distribution of next state
2. Compute corresponding expected immediate cost (discounted)
3. At end of prediction horizon: sum all immediate costs up

```matlab
function [J, dJdp] = value(p, m0, S0, dynmodel, policy, plant, cost, H)
```

Code
policy.p = p;          % overwrite policy.p with new parameters from minimize
p = unwrap(policy.p);  dp = 0*p;
m = m0;  S = S0;  L = zeros(1,H);

if nargout <= 1            % no derivatives required
   for t = 1:H
      [m, S] = plant.prop(m, S, plant, dynmodel, policy);  % get next state
      L(t) = cost.gamma^t.*cost.fcn(cost, m, S);      % expected discounted cost
   end
else                        % otherwise, get derivatives
   dm0dp = zeros([size(m0,1), length(p)]);
   dS0dp = zeros([size(m0,1)*size(m0,1), length(p)]);
   for t = 1:H
      [m, S, dmdm0, dSdm0, dmdS0, dSdS0, dmdp, dSdp] = ...% for all time steps in horizon
      plant.prop(m, S, plant, dynmodel, policy);  % get next state
      dmdp = dmdm0*dmdp + dmdS0*dSdp + dmdp;
      dSdp = dSdm0*dSdp + dSdS0*dSdp + dSdp;
      [L(t), dLdm, dLdS] = cost.fcn(cost, m, S);      % predictive cost
      L(t) = cost.gamma^t*L(t);                       % discount
      dp = dp + cost.gamma^t*( dLdm(:)’*dmdp + dLdS(:)’*dSdp )’;
      dm0dp = dmdp;  dS0dp = dSdp;                    % bookkeeping
   end
end

J = sum(L);  dJdp = rewrap(policy.p, dp);

7.2 Control Directory

7.2.1 concat.m

Summary: Compute a control signal $u$ from a state distribution $x \sim \mathcal{N}(x|m,s)$. Here, the predicted control distribution and its derivatives are computed by concatenating a controller "con" with a saturation function "sat", such as gSat.m.

function [M, S, C, dMdM, dSdM, cDcm, dMdS, dSds, cDds, dMdp, dSdp, cDdp] ... = conCat(con, sat, policy, m, s)

Example call: conCat(@congp, @gSat, policy, m, s)

Input arguments:

con      function handle (controller)
sat     function handle (squashing function)
policy  policy structure
.maxU maximum amplitude of control signal (after squashing)

\( m \) mean of input distribution \([D \times 1]\)

\( s \) covariance of input distribution \([D \times D]\)

Output arguments:

- \( M \) control mean \([E \times 1]\)
- \( S \) control covariance \([E \times E]\)
- \( C \) \( \text{inv}(s) \cdot \text{cov}(x,u) \) \([D \times E]\)
- \( dMdm \) deriv. of expected control wrt input mean \([E \times D]\)
- \( dSdm \) deriv. of control covariance wrt input mean \([E \times E \times D]\)
- \( dCdm \) deriv. of \( C \) wrt input mean \([D \times E \times D]\)
- \( dMds \) deriv. of expected control wrt input covariance \([E \times D \times D]\)
- \( dSds \) deriv. of control covariance wrt input covariance \([E \times E \times D \times D]\)
- \( dCds \) deriv. of \( C \) wrt input covariance \([D \times E \times D \times D]\)
- \( dMdp \) deriv. of expected control wrt policy parameters \([E \times P]\)
- \( dSdp \) deriv. of control covariance wrt policy parameters \([E \times E \times P]\)
- \( dCdp \) deriv. of \( C \) wrt policy parameters \([D \times E \times P]\)

where \( P \) is the total number of policy parameters

Copyright (C) 2008-2013 by Marc Deisenroth, Andrew McHutchon, Joe Hall, and Carl Edward Rasmussen.

Last modified: 2012-07-03

**High-Level Steps**

1. Compute unsquashed control signal
2. Compute squashed control signal

```matlab
function [M, S, C, dMdm, dSdm, dCdm, dMds, dSds, dCds, dMdp, dSdp, dCdp] = conCat(con, sat, policy, m, s)
```

**Code**

```matlab
maxU=policy.maxU; % amplitude limit of control signal
E=length(maxU); % dimension of control signal
D=length(m); % dimension of input

% pre-compute some indices
F=D+E; j=D+1:F; i=1:D;

% initialize M and S
M = zeros(F,1); M(i) = m; S = zeros(F); S(i,i) = s;

if nargout < 4 % without derivatives
    [M(j), S(j,j), Q] = con(policy, m, s); % compute unsquashed control signal v
```
7.2.2 congpm.m

Summary: Implements the mean-of-GP policy (equivalent to a regularized RBF network). Compute mean, variance and input-output covariance of the control u using a mean-of-GP policy function, when the input x is Gaussian. The GP is parameterized using a pseudo training set size N. Optionally, compute partial derivatives wrt the input parameters.

This version sets the signal variance to 1, the noise to 0.01 and their respective lengthscales to zero. This results in only the lengthscales, inputs, and outputs being trained.

function [M, S, C, dMdm, dSdm, dCdm, dMds, dSds, dCds, dMdp, dSdp, dCdp] ... = congpm(policy, m, s)

Input arguments:
policy (struct)

.p parameters that are modified during training

.hyp GP-log hyperparameters \( Ph = (d+2)D \) \( [ Ph ] \)

.inputs policy pseudo inputs \( [ N \times d ] \)

.targets policy pseudo targets \( [ N \times D ] \)

m mean of state distribution \( [ d \times 1 ] \)

s covariance matrix of state distribution \( [ d \times d ] \)

Output arguments:

\[
\begin{align*}
M & \quad \text{mean of the predicted control} \quad [ D \times 1 ] \\
S & \quad \text{covariance of predicted control} \quad [ D \times D ] \\
C & \quad \text{inv}(s) \times \text{covariance between input and control} \quad [ d \times D ] \\
dMdm & \quad \text{deriv. of mean control wrt mean of state} \quad [ D \times d ] \\
dSdm & \quad \text{deriv. of control variance wrt mean of state} \quad \quad [D \times D \times d ] \\
dCdm & \quad \text{deriv. of covariance wrt mean of state} \quad [ d \times d \times d ] \\
dMds & \quad \text{deriv. of mean control wrt variance} \quad [ D \times d \times d ] \\
dSds & \quad \text{deriv. of control variance wrt variance} \quad [D \times D \times d \times d ] \\
dCds & \quad \text{deriv. of covariance wrt variance} \quad [ d \times d \times d \times d ] \\
dMdp & \quad \text{deriv. of mean control wrt GP hyper-parameters} \quad [ D \times P ] \\
dSdp & \quad \text{deriv. of control variance wrt GP hyper-parameters} \quad [D \times D \times P ] \\
dCdp & \quad \text{deriv. of covariance wrt GP hyper-parameters} \quad [ d \times D \times P ] \\
\end{align*}
\]

where \( P = (d+2)D + n(d+D) \) is the total number of policy parameters.

Copyright (C) 2008-2013 by Marc Deisenroth, Andrew McHutchon, Joe Hall, and Carl Edward Rasmussen.

Last modified: 2013-01-24

High-Level Steps

1. Extract policy parameters from policy structure
2. Compute predicted control \( u = \text{inv}(s) \times \text{covariance between input and control} \)
3. Set derivatives of non-free parameters to zero
4. Merge derivatives

\begin{verbatim}
function [M, S, C, dMdm, dSdm, dCdm, dMds, dSds, dCds, dMdp, dSdp, dCdp] = congsp(policy, m, s)
\end{verbatim}

Code

\begin{verbatim}
1 % 1. Extract policy parameters
2 policy.hyp = policy.p.hyp;
3 policy.inputs = policy.p.inputs;
\end{verbatim}
\begin{verbatim}
4 policy.targets = policy.p.targets;

5 6 7 8 9 10
% fix policy signal and the noise variance
% (avoids some potential numerical problems)
policy.hyp(end-1,:) = log(1); % set signal variance to 1
policy.hyp(end,:) = log(0.01); % set noise standard dev to 0.01

11 12 13
% 2. Compute predicted control \( u \), inv(s)*covariance between input and control.
if nargout < 4 % if no derivatives are required
\[ M, S, C \] = gp2(policy, m, s);
else % else compute derivatives too
\[ M, S, C, dMdm, dSdm, dCdm, dMds, dSds, dCdsm, dSds, dCdms, dMs, dSds, dCds, dMdt, \]
\[ dSdt, dCdt, dMdh, dSdh, dCdh \] = gp2d(policy, m, s);

17 18
19 20
\[ d = \text{size}(policy.inputs,2); \]
\[ d2 = \text{size}(policy.hyp,1); \]
\[ \text{dimU} = \text{size}(policy.targets,2); \]
\[ \text{sidx} = \text{bsxfun(@plus,(d+1:d2)',(0:dimU-1)*d2)); \]
\[ \text{dMdh}(:,\text{sidx}(:)) = 0; \]
\[ \text{dSdh}(:,\text{sidx}(:)) = 0; \]
\[ \text{dCdh}(:,\text{sidx}(:)) = 0; \]

23 24
% 4. Merge derivatives
\[ \text{dMdp} = [\text{dMdh} \ \text{dMdi} \ \text{dMdt}]; \]
\[ \text{dSdp} = [\text{dSdh} \ \text{dSdi} \ \text{dSdt}]; \]
\[ \text{dCdp} = [\text{dCdh} \ \text{dCdi} \ \text{dCdt}]; \]
end
\end{verbatim}

### 7.2.3 conlin.m

**Summary:** Affine controller \( u = Wx + b \) with input dimension \( D \) and control dimension \( E \). Compute mean and covariance of the control distribution \( p(u) \) from a Gaussian distributed input \( x \sim N(x|m, s) \). Moreover, the \( s^{-1}\text{cov}(x, u) \) is computed.

function \[ [M, S, V, dMdm, dSdm, dVdm, dMds, dSds, dVds, dMd, dSd, dVd] \ldots \]
\[ = \text{conlin}(policy, m, s) \]

**Input arguments:**

- \textbf{policy} policy structure
  - \textbf{.p} parameters that are modified during training
  - \textbf{.w} linear weights \([ E \times D ]\)
  - \textbf{.b} biases/offset \([ E ]\)
- \textbf{m} mean of state distribution \([ D ]\)
- \textbf{s} covariance matrix of state distribution \([ D \times D ]\)

**Output arguments:**

- \textbf{M} mean of predicted control \([ E ]\)
- \textbf{S} variance of predicted control \([ E \times E ]\)
- \textbf{C} \( \text{inv(s)} \) times input-output covariance \([ D \times E ]\)
- \textbf{dMdm} deriv. of mean control wrt input mean \([ E \times D ]\)
- \textbf{dSdm} deriv. of control covariance wrt input mean \([ E \times E \times D ]\)
- \textbf{dCdm} deriv. of \( C \) wrt input mean \([ D \times E \times D ]\)
- \textbf{dMds} deriv. of mean control wrt input covariance \([ E \times D \times D ]\)
dSds  deriv. of control covariance wrt input covariance  [E*E x D*D]
dCds  deriv. of C wrt input covariance  [D*E x D*D]
dMdp  deriv. of mean control wrt policy parameters  [ E x P ]
dSdp  deriv. of control covariance wrt policy parameters  [E*E x P ]
dCdp  deriv. of C wrt policy parameters  [D*E x P ]

where P = (D+1)*E is the total number of policy parameters

Copyright (C) 2008-2013 by Marc Deisenroth, Andrew McHutchon, Joe Hall, and Carl Edward Rasmussen.

Last modified: 2012-07-03

High-Level Steps

1. Extract policy parameters from policy structure
2. Predict control signal
3. Compute derivatives if required

Code

```matlab
function [M, S, V, dMdm, dSdm, dVdm, dMds, dSds, dVds, dMdp, dSdp, dVdp] = conlin(policy, m, s)

% 1. Extract policy parameters from policy structure
w = policy.p.w;  % weight matrix
b = policy.p.b;   % bias/offset
[E D] = size(w);  % dim of control and state

% 2. Predict control signal
M = w*m + b;      % mean
S = w*s'*w';      % covariance
V = w';           % inv(s)*input−output covariance

% 3. Compute derivatives if required
if nargout > 3
    dMdm = w;         % mean
    dSdm = zeros(E*E, D);  dVdm = zeros(D*E, D);
    dMds = zeros(E,D*D); dSds = kron(w,w);  dVds = zeros(D*E,D*D);
    X = reshape(1:D*D,[D D]); XT=X';
    dSds=dSds+dSds(:,XT(:))/2; % symmetrize
    X = reshape(1:E,E); XT=X';
    dSds=dSds+dSds(XT(;,:))/2;
    wTdw = reshape(permute(reshape(eye(E*D),[E D E D]),[2 1 3 4]),[E*E E*D]);
    dMdp = [eye(E) kron(m',eye(E))];
    dSdp = [zeros(E*E,E) kron(eye(E),w*s*wTdw) + kron(w*s,eye(E))];
    dSdp = (dSdp + dSdp(XT(:,:)))/2; % symmetrize
    dVdp = [zeros(D*E,E) wTdw];
end
```
7.3 GP Directory

7.3.1 train.m

Summary: Train a GP model with SE covariance function (ARD). First, the hyper-parameters are trained using a full GP. Then, if gpmodel.induce exists, indicating sparse approximation, if enough training examples are present, train the inducing inputs (hyper-parameters are taken from the full GP). If no inducing inputs are present, then initialize them to be a random subset of the training inputs.

function [gpmodel nlml] = train(gpmodel, dump, iter)

Input arguments:

- gpmodel: GP structure
- inputs: GP training inputs, \[ N \times D \]
- targets: GP training targets, \[ N \times E \]
- hyp (optional): GP log-hyper-parameters, \[ D+2 \times E \]
- induce (optional): pseudo inputs for sparse GP
- dump: not needed for this code, but required for compatibility reasons
- iter: optimization iterations for training, \[ 1 \times 2 \]
  - [full GP, sparse GP]

Output arguments:

- gpmodel: updated GP structure
- nlml: negative log-marginal likelihood

Copyright (C) 2008-2013 by Marc Deisenroth, Andrew McHutchon, Joe Hall, and Carl Edward Rasmussen.

Last modified: 2013-05-16

High-Level Steps

1. Initialization
2. Train full GP
3. If necessary: train sparse GP (FITC/SPGP)

```
function [gpmodel nlml] = train(gpmodel, dump, iter)
```

Code
% 1) Initialization
if nargin < 3, iter = [-500 -1000]; end % default training iterations

D = size(gpmodel.inputs,2); E = size(gpmodel.targets,2); % get variable sizes
covfunc = {'covSum', {'covSEard', 'covNoise'} }; % specify AWD covariance
curb.snr = 1000; curb.ls = 100; curb.std = std(gpmodel.inputs); % set hyp curb
if ~isfield(gpmodel, 'hyp') % if we don’t pass in hyper-parameters, define them
    gpmodel.hyp = zeros(D+2,E); nlml = zeros(1,E);
    lh = repmat([log(std(gpmodel.inputs)) 0 -1],1,E); % init hyp length scales
    lh(D+1,:) = log(std(gpmodel.targets)); % signal std dev
    lh(D+2,:) = log(std(gpmodel.targets)/10); % noise std dev
else
    lh = gpmodel.hyp; % GP hyper-parameters
end

% 2a) Train full GP (always)
fprintf('Train hyper-parameters of full GP ...
');
for i = 1:E % train each GP separately
    fprintf(' GP %i/%i\n', i, E);
    try % BFGS training
        [gpmodel.hyp(:,:,i), v] = minimize(lh(:,:,i), @hypCurb, iter(1), covfunc, ...
            gpmodel.inputs, gpmodel.targets(:,:,i), curb);
        catch % conjugate gradients (BFGS can be quite aggressive)
            [gpmodel.hyp(:,:,i), v] = minimize(lh(:,:,i), @hypCurb, ...
                struct('length', iter(1), 'method', 'CG', 'verbosity', 1), covfunc, ...
                gpmodel.inputs, gpmodel.targets(:,:,i), curb);
        end
    end
nlml(i) = v(end);
end

% 2b) If necessary: sparse training using FITC/SPGP (Snelson & Ghahramani, 2006)
if isfield(gpmodel, 'induce') % are we using a sparse approximation?
    [N,D] = size(gpmodel.inputs);
    [M uD uE] = size(gpmodel.induce);
    if M >= N; return; end % if too few training examples, we don’t need FITC
    fprintf('Train pseudo-inputs of sparse GP ...
');
    if uD == 0 % we don’t have inducing inputs yet?
        gpmodel.induce = zeros(M,D,uE);
        iter = 3*iter; % train a lot for the first time (it’s still cheap!)
        [cidx, ctrs] = kmeans(gpmodel.inputs, M); % kmeans: initialize pseudo inputs
        for i = 1:uE
            j = randperm(N);
            gpmodel.induce(:,:,i) = gpmodel.inputs(j(1:M),:); % random subset
        end
        gpmodel.induce(:,:,i) = ctrs;
    end
end
% train sparse model
[gpmodel.induce nlml2] = minimize(gpmodel.induce, 'fitc', iter(end), gpmodel);
fprintf(' GP NLML: full: %e, sparse: %e, diff: %e\n', ...
    sum(nlml), nlml2(end), nlml2(end)-sum(nlml));
7.3.2 hypCurb.m

**Summary:** Wrapper for GP training (via gpr.m), penalizing large SNR and extreme length-scales to avoid numerical instabilities

```matlab
function [f df] = hypCurb(lh, covfunc, x, y, curb)
```

**Input arguments:**

- `lh` log-hyper-parameters \([D+2 \times E]\)
- `covfunc` covariance function, e.g.,
  ```matlab
covfunc = \{'covSum', \{'covSEard', 'covNoise'\}\};
```
- `x` training inputs \([n \times D]\)
- `y` training targets \([n \times E]\)
- `curb` (optional) parameters to penalize extreme hyper-parameters
  - `.ls` length-scales
  - `.snr` signal-to-noise ratio (try to keep it below 500)
  - `.std` additional parameter required for length-scale penalty

**Output arguments:**

- `f` penalized negative log-marginal likelihood
- `df` derivative of penalized negative log-marginal likelihood wrt GP log-hyper-parameters

Copyright (C) 2008-2013 by Marc Deisenroth, Andrew McHutchon, Joe Hall, and Carl Edward Rasmussen.

Last modified: 2011-12-19

**High-Level Steps**

1. Compute the negative log-marginal likelihood (plus derivatives)
2. Add penalties and change derivatives accordingly

```matlab
if nargin < 5, curb.snr = 500; curb.ls = 100; curb.std = 1; end  % set default
p = 30;                                           % penalty power
D = size(x,2);
if size(lh,1) == 3*D+2; li = 1:2*D; sfi = 2*D+1:3*D+1;  % 1D and DD terms
elseif size(lh,1) == 2*D+1; li = 1:D; sfi = D+1:2*D;      % Just 1D terms
```

**Code**
else if size(lh, 1) == D+2; li = 1:D; sfi = D+1; \% Just DD terms
else error('Incorrect number of hyperparameters');
end

ll = lh(li); lsf = lh(sfi); lsn = lh(end);

% 1) compute the negative log-marginal likelihood (plus derivatives)
[f df] = gpr(lh, covfunc, x, y); \% first, call gpr

% 2) add penalties and change derivatives accordingly
f = f + sum(((ll - log(curb.std'))./log(curb.ls)).^p); \% length-scales
df(li) = df(li) + p*(ll - log(curb.std'))./(p-1)/log(curb.ls).^p;

f = f + sum(((lsf - lsn)/log(curb.snr)).^p); \% signal to noise ratio
df(sfi) = df(sfi) + p*(lsf - lsn)./(p-1)/log(curb.snr).^p;

df(end) = df(end) - p*sum((lsf - lsn)./(p-1)/log(curb.snr).^p);

7.3.3 fitc.m

Summary: Compute the FITC negative log marginal likelihood and its derivatives with respect to
the inducing inputs (we don’t compute the derivatives with respect to the GP hyper-parameters)

function [nml dnml] = fitc(induce, gpmodel)

Input arguments:

induce matrix of inducing inputs [M x D x uE]
M: number of inducing inputs
E: either 1 (inducing inputs are shared across target dim.)
or E (different inducing inputs for each target dim.)
gpmodel GP structure
.hyp log-hyper-parameters [D+2 x E]
.inputs training inputs [N x D]
.targets training targets [N x E]
.noise (opt) noise

Output arguments:

nml negative log-marginal likelihood
dnml derivative of negative log-marginal likelihood wrt
inducing inputs

Adapted from Ed Snelson’s SPGP code.

Copyright (C) 2008-2013 by Marc Deisenroth, Andrew McHutchon, Joe Hall, and Carl Edward Rasmussen.

Last modified: 2013-05-21
High-Level Steps

1. Compute FITC marginal likelihood
2. Compute corresponding gradients wrt the pseudo inputs

```matlab
function [nlml dnlml] = fitc(induce, gpmodel)

ridge = 1e-06; % jitter to make matrix better conditioned

[N D] = size(gpmodel.inputs); E = size(gpmodel.targets,2);
[M uD uE] = size(induce);
if uD ~= D || (uE==1 && uE ~= E); error('Wrong size of inducing inputs'); end

nlml = 0; dfxb = zeros(M, D); dnlml = zeros(M, D, E); % zero and allocate outputs

for j = 1:E
  if uE > 1; u = induce(:,j); else u = induce; end
  b = exp(gpmodel.hyp(1:D,j)); % length-scales
  c = gpmodel.hyp(D+1,j); % log signal std dev
  sig = exp(2.*gpmodel.hyp(D+2,j)); % noise variance
  xb = bsxfun(@rdivide,u,b'); % divide inducing by lengthscales
  x = bsxfun(@rdivide, gpmodel.inputs, b'); % divide inputs by lengthscales
  y = gpmodel.targets(:,j);

  Kmm = exp(2*c-maha(xb,xb)/2) + ridge*eye(M);
  Kmn = exp(2*c-maha(xb,x)/2);

  % Check whether Kmm is no longer positive definite. If so, return
  try
    L = chol(Kmm)';
    catch
      nlml = Inf; dnlml = zeros(size(params));
    return;
  end
  V = L\Kmn; % inv(sqrt(Kmm)) * Kmn

  if isfield(gpmodel,'noise')
    Gamma = 1 + (exp(2*c)-sum(V.*2')+gpmodel.noise(:,j))/sig;
  else
    Gamma = 1 + (exp(2*c)-sum(V.*2'))/sig; % Gamma = diag(Kmn-Qmn)/sig + I
  end

  V = bsxfun(@rdivide,V,sqrt(Gamma)'); % inv(sqrt(Kmm)) * Kmn * inv(sqrt(Gamma))
  y = y./sqrt(Gamma);
  Am = chol(sig*eye(M) + VsV'); % chol(inv(sqrt(Kmm)) * A * inv(sqrt(Kmm)))
  % VsV' = inv(chol(Kmm)') * K' * inv(chol(Kmm))
  Vy = VsV; % diag(Am)
  beta = Am\Vy;

  nlml = nlml + sum(log(diag(Am))) + (N-M)/2*log(sig) + sum(log(Gamma))/2 ...
```

72
\[
\frac{(y' \cdot y - \beta' \cdot \beta)}{2 \cdot \text{sig}} + 0.5 \cdot H \cdot \log(2 \cdot \pi);
\]

if nargin == 2 % ... and if requested, its partial derivatives

\[
\begin{align*}
\text{At} &= \text{L} \cdot \text{Am} \text{; } \text{iAt} &= \text{At} \setminus \text{eye}(M) \text{; } \% \text{ chol}(\text{sig} \cdot \text{B}) \text{ [Ed's thesis, p. 40]} \\
\text{iA} &= \text{iAt} \cdot \text{iAt} \text{; } \% \text{ inv}(\text{sig} \cdot \text{B}) \\
\text{iAmV} &= \text{Am} \setminus \text{V} \text{; } \% \text{ inv}(\text{Am} \cdot \text{V}) \\
\text{B1} &= \text{At} \setminus (\text{iAmV}) \text{; } \% \text{ b1 = B1} \cdot \text{y} \\
\text{b1} &= \text{At} \setminus \text{\beta} \text{; } \% \text{ b1 = B1} \cdot \text{y} \\
\text{iLV} &= \text{L} \setminus \text{V} \text{; } \% \text{ inv}(\text{Kmn}) \cdot \text{Kmn} \cdot \text{inv}(\sqrt{\text{Gamma}}) \\
\text{iL} &= \text{L} \setminus \text{eye}(M) \text{; } \% \text{ inv}(\text{Am}) \cdot \text{V} \\
\text{iAmV} &= \text{Am} \setminus \text{V} \text{; } \% \text{ inv}(\text{Am}) \cdot \text{V} \\
\text{mu} &= ((\text{Am} \setminus \text{\beta})' \cdot \text{V})' ; \\
\text{bs} &= y \cdot (\text{beta} \cdot \text{AmV})' / \text{sig} - \text{sum}((\text{AmV} \cdot \text{AmV})' / 2 - (y' \cdot \text{mu}' \cdot \text{2}) / 2 / \text{sig} + 0.5 ; \\
\text{TT} &= \text{iLV} \cdot (\text{bsxfun}(\text{@times},\text{iLV},\text{bs})) \text{; } \% \text{ overwrites Kmn} \\
\text{Kmn} &= \text{bsxfun}(\text{@rdivide},\text{Kmn},\text{sqrt}(\text{Gamma})) \text{; } \% \text{ overwrites Kmn} \\
\text{for} \ i = 1: \text{D} % \text{ derivatives wrt inducing inputs} \\
\text{dsq_mm} &= \text{bsxfun}(\text{@minus},\text{xb}(;,: \text{i}),\text{xb}(;,: \text{i})) \cdot \text{Kmm} ; \\
\text{dsq_mn} &= \text{bsxfun}(\text{@minus},\text{xb}(;,: \text{i}),\text{xb}(;,: \text{i})) \cdot \text{Kmn} ; \\
\text{dGamma} &= -2 / \text{sig} \cdot \text{dsq_mn} \cdot \text{iLV} ; \\
\text{dsq_mn} &= \text{bsxfun}(\text{@minus},\text{xb}(;,: \text{i}),\text{xb}(;,: \text{i})) \cdot \text{Kmm} ; \\
\text{dfxb}(;,: \text{i}) &= -\text{b1} \cdot (\text{dsq_mn} \cdot (\text{y} - \text{mu}) / \text{sig} + \text{dsq_mm} \cdot \text{b1}) + \text{dGamma} \cdot \text{bs} \text{ ...} \\
& \quad + \text{sum}((\text{iKmm} - \text{iA} \cdot \text{sig}) \cdot \text{dsq_mm}, 2) - 2 / \text{sig} \cdot \text{sum}((\text{dsq_mm} \cdot \text{TT}, 2) ; \\
\text{dsq_mn} &= \text{dsq_mn} \cdot \text{b1} ; \\
\text{dfxb}(;,: \text{i}) &= \text{dfxb}(;,: \text{i}) + \text{sum}(\text{dsq_mn}, 2) ; \\
\text{dfxb}(;,: \text{i}) &= \text{dfxb}(;,: \text{i}) / \text{b}(\text{i}) ; \\
\end{align*}
\]

end

\text{if} \ 1 = \text{uE} ; \text{dnlml} &= \text{sum}(\text{dnlml}, 3) ; \text{end} % \text{ combine derivatives if sharing inducing}

7.3.4 gp0.m

**Summary:** Compute joint predictions for multiple GPs with uncertain inputs. If gpmodel.nigp exists, individual noise contributions are added. Predictive variances contain uncertainty about the function, but no noise.

function [M, S, V] = gp0(gpmodel, m, s)

**Input arguments:**

- **gpmodel** - GP model struct
- **hyp** - log-hyper-parameters \[D+2 \times E\]
- **inputs** - training inputs \[n \times D\]
- **targets** - training targets \[n \times E\]
- **nigp** - (optional) individual noise variance terms \[n \times E\]
- **m** - mean of the test distribution \[D \times 1\]
- **s** - covariance matrix of the test distribution \[D \times D\]
Output arguments:

\[ M \quad \text{mean of pred. distribution} \quad [E \times 1] \]
\[ S \quad \text{covariance of the pred. distribution} \quad [E \times E] \]
\[ V \quad \text{inv}(s) \text{ times covariance between input and output} \quad [D \times E] \]

Copyright (C) 2008-2013 by Marc Deisenroth, Andrew McHutchon, Joe Hall, and Carl Edward Rasmussen.

Last modified: 2013-05-24

High-Level Steps

1. If necessary, compute kernel matrix and cache it
2. Compute predicted mean and inv(s) times input-output covariance
3. Compute predictive covariance matrix, non-central moments
4. Centralize moments

```matlab
function [M, S, V] = gp0(gpmodel, m, s)

persistent K iK beta oldX oldn;
[n, D] = size(gpmodel.inputs); % number of examples and dimension of inputs
[n, E] = size(gpmodel.targets); % number of examples and number of outputs
X = gpmodel.hyp; % short hand for hyperparameters

if numel(X) ~= numel(oldX) || isempty(iK) || sum(any(X ~= oldX)) || n ~= oldn
    oldX = X; oldn = n;
    iK = zeros(n,n,E); K = zeros(n,n,E); beta = zeros(n,E);
end
for i=1:E
    inp = bsxfun(@rdivide,gpmodel.inputs,exp(X(1:D,i)'));
    K(:,:,i) = exp(2*X(D+1,i)-maha(inp.inp)/2);
    if isfield(gpmodel,'nigp')
        L = chol(K(:,:,i) + exp(2*X(D+2,i))*eye(n) + diag(gpmodel.nigp(:,:,i)))';
    else
        L = chol(K(:,:,i) + exp(2*X(D+2,i))*eye(n))';
    end
    iK(:,:,i) = L'\(L\eye(n));
    beta(:,:,i) = L'\(L\gpmodel.targets(:,:,i));
end
k = zeros(n,E); M = zeros(E,1); V = zeros(D,E); S = zeros(E);
inp = bsxfun(@minus,gpmodel.inputs,m');
```

% centralize inputs
% 2) compute predicted mean and \( \text{inv}(s) \) times input-output covariance

\[
\text{for } i = 1:E
\]
\[
iL = \text{diag}(\exp(-X(1:D,i))); \quad \% \text{inverse length-scales}
\]
\[
in = \text{inp} * iL;
\]
\[
S = iL * \text{inv}(\text{eye}(D));
\]
\[
t = \text{in} / B;
\]
\[
l = \exp(-\sum(\text{in} .* t,2)/2); \quad \text{lb} = l * \text{beta}(:,i);
\]
\[
tiL = t * iL;
\]
\[
c = \exp(2*X(D+1,i))/\text{det}(B); \quad \% \text{inv}(s) \text{ times input-output covariance}
\]
\[
M(i) = \text{sum(lb)} * c; \quad \% \text{predicted mean}
\]
\[
V(:,i) = tiL' * \text{lb} * c;
\]
\[
k(:,i) = 2*X(D+1,i)-\sum(\text{in} .* \text{in},2)/2;
\]
\[
\text{end}
\]
\[
% 3) compute predictive covariance, non-central moments
\]
\[
\text{for } i = 1:E
\]
\[
i = \text{bsxfun}(@rdivide,\text{inp},\exp(2*X(1:D,i)'));
\]
\[
\text{for } j = 1:i
\]
\[
R = s * \text{diag}(\exp(-2*X(1:D,i))+\exp(-2*X(1:D,j)))+\text{eye}(D);
\]
\[
t = 1/\text{sqrt}(\text{det}(R));
\]
\[
ij = \text{bsxfun}(@rdivide,\text{inp},\exp(2*X(1:D,j)'));
\]
\[
L = \exp(\text{bsxfun}(@plus,k(:,i),k(:,j)'))+\text{maha}(ii,-ij,R\text{\#}2)/2);
\]
\[
\text{if } i==j
\]
\[
S(i,i) = tiL' * \text{beta}(:,i)' * L * \text{beta}(:,i) - \text{sum}(\text{sum}(iK(:,:,i) .* L));
\]
\[
\text{else}
\]
\[
S(i,j) = \text{beta}(:,i)' * L * \text{beta}(:,j) * t;
\]
\[
S(j,i) = S(i,j);
\]
\[
\text{end}
\]
\[
\text{end}
\]
\[
% 4) centralize moments
\]
\[
S = S - M*M';
\]

7.3.5  \textit{gp1.m}

\textbf{Summary:} Compute joint predictions for the FITC sparse approximation to multiple GPs with uncertain inputs. Predictive variances contain uncertainty about the function, but no noise. If \texttt{gpmodel.nigp} exists, individual noise contributions are added.

\texttt{function [M, S, V] = gp1d(gpmodel, m, s)}

\textbf{Input arguments:}

\begin{itemize}
  \item \texttt{gpmodel} \quad \text{GP model struct}
  \item \texttt{hyp} \quad \text{log-hyper-parameters} \quad [D+2 \times E]
  \item \texttt{inputs} \quad \text{training inputs} \quad [n \times D]
  \item \texttt{targets} \quad \text{training targets} \quad [n \times E]
  \item \texttt{nigp} \quad \text{(optional) individual noise variance terms} \quad [n \times E]
\end{itemize}
m mean of the test distribution [D x 1]
s covariance matrix of the test distribution [D x D]

Output arguments:
M mean of pred. distribution [E x 1]
S covariance of the pred. distribution [E x E]
V inv(s) times covariance between input and output [D x E]

Copyright (C) 2008-2013 by Marc Deisenroth, Andrew McHutchon, Joe Hall, and Carl Edward Rasmussen.
Last modified: 2013-03-05

High-Level Steps
1. If necessary, compute kernel matrix and cache it
2. Compute predicted mean and inv(s) times input-output covariance
3. Compute predictive covariance matrix, non-central moments
4. Centralize moments

```matlab
function [M, S, V] = gp1(gpmodel, m, s)

if ~isfield(gpmodel, 'induce') || numel(gpmodel.induce)==0,
    [M, S, V] = gp0(gpmodel, m, s); return; end

persistent iK iK2 beta oldX;
ridge = 1e-6; % jitter to make matrix better conditioned
[n, D] = size(gpmodel.inputs); % number of examples and dimension of inputs
E = size(gpmodel.targets, 2); % number of examples and number of outputs
X = gpmodel.hyp; input = gpmodel.inputs; targets = gpmodel.targets;

[np pD pE] = size(gpmodel.induce); % number of pseudo inputs per dimension
input = gpmodel.induce; % all pseudo inputs

% 1) If necessary: re-compute cached variables
if numel(X) ~= numel(oldX) || isempty(iK) || isempty(iK2) || ... % if necessary
    oldX = X;
    iK = zeros(np,n,E); iK2 = zeros(np,np,E); beta = zeros(np,E);

    for i=1:E
        pinp = bsxfun(@rdivide, input(:,i,:), exp(X(1:D,i)'));
        inp = bsxfun(@rdivide, pinp, exp(X(1:D,i)'));
        Kmm = exp(2*X(D+1,i)-maha(pinp,pinp)/2) + ridge*eye(np); % add small ridge
        Kmn = exp(2*X(D+1,i)-maha(pinp,inp)/2);
        L = chol(Kmm);
```
\begin{verbatim}
V = L\Kmn;  \quad \% \text{inv} (\sqrt{\text{Kmn}}) * \text{Kmn}
if isfield(gpmmodel, 'nigp')
    G = \exp(2*X(D+1,i)) - \text{sum}(V.^2) + gpmmodel.nigp(:,i)';
else
    G = \exp(2*X(D+1,i)) - \text{sum}(V.^2);
end
G = \text{sqrt}( \text{1+G/exp(2*X(D+2,i)))};
V = bsxfun(@rdivide, V,G);
Am = \text{chol} (\exp(2*X(D+2,i)) * eye(np) + V*V')';

At = L*Am;  \quad \% \text{chol}(\text{sig*B}) \quad \text{[thesis, p. 40]}
iAt = At\eye(np);

k = \text{zeros}(np,E); M = \text{zeros}(E,1); V = \text{zeros}(D,E); S = \text{zeros}(E);  \quad \% \text{allocate}
inp = \text{zeros}(np,D,E);  

\text{// 2) Compute predicted mean and inv(s) times input–output covariance}
for i=1:E
    inp(:,i) = bsxfun(@minus, inp(:,min(i,pE)), m');
L = diag(exp(-X(1:D,i)));
in = inp(:,i)*L;
S = L*s*L+eye(D);

t = inp/B;
L = \exp(-\text{sum(in.*t,2)/2}); lb = 1.*beta(:,i);
tL = t*L;
\epsilon = \exp(2*X(D+1,i))/\text{sqrt(det}(B));

M(i) = \text{sum}(lb)*c;  \quad \% \text{predicted mean}
V(:,i) = tL's*lb*c;  \quad \% \text{inv(s) times input–output covariance}
k(:,i) = 2*X(D+1,i) - \text{sum(in.*in,2)/2};
end

\text{// 3) Compute predictive covariance matrix, non–central moments}
for i=1:E
    ii = bsxfun(@rdivide, inp(:,i), exp(2*X(1:D,i)'));
for j=1:i
    R = s*diag(exp(-2*X(1:D,i))+exp(-2*X(1:D,j)))+eye(D); t = 1./sqrt(det(R));
    ij = bsxfun(@rdivide, inp(:,j), exp(2*X(1:D,j)'));
    L = exp(bsxfun(@plus, k(:,i), k(:,j)')) + maha(ii, -ij, R/s/2);
    if i=j
        S(i,i) = t*(beta(:,i)'*L*beta(:,i) - sum(sum(iK2(:,i,i).*L)));
    else
        S(i,j) = beta(:,i)'*L*beta(:,j)*t; S(j,i) = S(i,j);
    end
end
S(i,i) = S(i,i) + \exp(2*X(D+1,i));
end
\end{verbatim}
7.3.6 gp2.m

Summary: Compute joint predictions and derivatives for multiple GPs with uncertain inputs. Does not consider the uncertainty about the underlying function (in prediction), hence, only the GP mean function is considered. Therefore, this representation is equivalent to a regularized RBF network. If gpmodel.nigp exists, individual noise contributions are added.

function [M, S, V] = gp2(gpmodel, m, s)

Input arguments:

- gpmodel: GP model struct
- hyp: log-hyper-parameters [D+2 x E]
- inputs: training inputs [n x D]
- targets: training targets [n x E]
- nigp: (optional) individual noise variance terms [n x E]
- m: mean of the test distribution [D x 1]
- s: covariance matrix of the test distribution [D x D]

Output arguments:

- M: mean of pred. distribution [E x 1]
- S: covariance of the pred. distribution [E x E]
- V: inv(s) times covariance between input and output [D x E]

Copyright (C) 2008-2013 by Marc Deisenroth, Andrew McHutchon, Joe Hall, and Carl Edward Rasmussen.

Last modified: 2013-03-05

High-Level Steps

1. If necessary, re-compute cached variables
2. Compute predicted mean and inv(s) times input-output covariance
3. Compute predictive covariance matrix, non-central moments
4. Centralize moments
Code

```
persistent iK oldX oldIn oldOut beta oldn;
D = size(gpmodel.inputs,2);  \% number of examples and dimension of inputs
[n, E] = size(gpmodel.targets); \% number of examples and number of outputs

input = gpmodel.inputs; target = gpmodel.targets; X = gpmodel.hyp;

\% 1) If necessary: re-compute cached variables
if numel(X) ~= numel(oldX) || isempty(iK) \% n ~= oldn || ...
    oldX = X; oldIn = input; oldOut = target; oldn = n;
end

K(:,i,:) = exp(2*X(i+1,i)-maha(inp.inp)/2);
if isfield(gpmodel,'nigp')
    L = chol(K(:,i,:)+exp(2*X(i+2,i))*eye(n)+diag(gpmodel.nigp(:,i))');
else
    L = chol(K(:,i,:)+exp(2*X(i+2,i))*eye(n))';
end
iK(:,i,:) = L' \backslash (L' \backslash eye(n))';
beta(:,i,:) = L' \backslash (L' \backslash gpmodel.targets(:,i));
end

k = zeros(n,E); M = zeros(E,1); V = zeros(D,E); S = zeros(E);
inp = bsxfun(@minus,gpmodel.inputs,m'); \% centralize inputs

\% 2) Compute predicted mean and inv(s) times input–output covariance
for i=1:E
    IL = diag(exp(-X(i,:)))); \% inverse length–scales
    in = inp*IL;
    S = il*s*il+eye(D);
    t = in/B;
    l = exp(-sum(in.*t,2)/2);  \% lb = l.*beta(:,i);
    tl = t*il;
    c = exp(2*X(i+1,i))/sqrt(det(B));
    M(i,:) = sum(lb)*c; \% predicted mean
    V(:,i,:) = tl'*lb*c; \% inv(s) times input–output covariance
    k(:,i) = 2*X(i+1,i)-sum(in.*in,2)/2;
end

\% 3) Compute predictive covariance, non–central moments
for i=1:E
    ii = bsxfun(@divide,inp,exp(2*X(i,:))');
    for j=1:1
        R = s*diag(exp(-2*X(i,:)')+exp(-2*X(1:j,:)))+eye(D);
        t = 1/sqrt(det(R));
        ij = bsxfun(@divide,inp,exp(2*X(i,j)));
        L = exp(bsxfun(@plus,k(:,i),k(:,j)+maha(ii,-ij,R\s/2));
    end
```

79
\[ S(i,j) = t * \text{beta}(i,:) * L * \text{beta}(j,:) \; ; \; S(j,i) = S(i,j) ; \]

\[ S(i,i) = S(i,i) + 1e^{-6}; \quad \% \text{add small jitter for numerical reasons} \]

\% 4) Centralize moments
\[ S = S - M \times M' ; \]
Bibliography


