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

February 3, 2014

Contents

Abstract

We describe a Matlab package for the PILCO policy search framework for data-efficient reinforcement learning. This package implements the PILCO learning framework in multiple different scenarios with continuous states and actions: pendulum swing-up, cart-pole swing-up, double-pendulum swing-up (with either one or two actuators), cart-double-pendulum swing-up, and unicycling. Results from some of these scenarios have been presented previously in [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 RLGlue, CLSquare[1], PIQLE[2], RL Toolbox[3], LibPG[4], or RLPy[5].

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

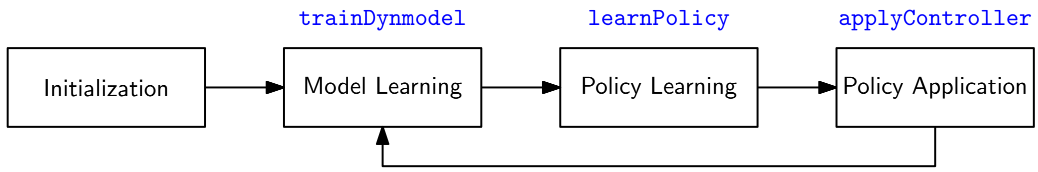


Figure 1.1: Main modules: After an initialization, the first module is responsible for training a GP from available data. The second module is used for policy learning, which consists of policy evaluation and improvement. The third module is responsible for applying the learned policy to the system,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

1.2 Software Design and Implementation

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

1.2.1 Model Learning

The forward model learned is a non-parametric, probabilistic Gaussian process [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 $(\boldsymbol{x}_t, \boldsymbol{u}_t)$, where t is a time index. Training targets are either successor states $\boldsymbol{x}_{t+1}$ or differences $\Delta_t = \boldsymbol{x}_{t+1} - \boldsymbol{x}_t$.
By default, a full GP model is trained by evidence maximization, where we penalize high signal-to-noise ratios in order to maintain numerical stability. There is an option to switch to sparse GPs in case the number of data points exceeds a particular threshold. We implemented the FITC/SPGP sparse GP method proposed by [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 $p(\boldsymbol{x}_1|\pi), \dots, p(\boldsymbol{x}_T|\pi)$ for a given controller π. To do so, we follow the analytic moment-matching approach proposed by [6], and approximate all $p(\boldsymbol{x}_t|\pi)$ by Gaussians $\mathcal{N}\big(\boldsymbol{x}_t \,|\, \boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t\big)$.
Policy Evaluation. Once the long-term predictions $p(\boldsymbol{x}_1|\pi), \dots, p(\boldsymbol{x}_T|\pi)$ are computed, the expected long-term cost

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

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

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

where $\|\boldsymbol{x} - \boldsymbol{x}_{\text{target}}\|^2_{\boldsymbol{W}}$ is the Mahalanobis distance between $\boldsymbol{x}$ and $\boldsymbol{x}_{\text{target}}$, weighted by $\boldsymbol{W}$, σ_c is a scaling factor, and $\boldsymbol{x}_{\text{target}}$ is a target state.
Policy Improvement. To improve the policy, we use gradient-based Quasi-Newton optimization methods, such as BFGS. The required gradients of the expected long-term cost J^π with respect to the policy parameters are computed analytically, which allows for learning low-level policies with hundreds of parameters [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 π computes the corresponding control signal $\boldsymbol{u}_t$ from the current state $\boldsymbol{x}_t$. An ODE solver is used to determine the corresponding successor state $\boldsymbol{x}_{t+1}$. The module returns a trajectory of state-action pairs, from which the training inputs and targets for the GP model can be extracted.

If the controller is applied to a real system, such as a robot, this module is not necessary. Instead, state estimation and control computation need to be performed on the robot directly [4].

1.3 User Interface by Example

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

In line 2, a scenario-specific settings script is executed. Here, we have to define the policy structure (e.g., parametrization, torque limits), the cost function, the dynamics model (e.g., definition of training inputs/targets), some details about the system/plant (e.g., sampling frequency) that are needed for the ODE solver, and some general parameters (e.g., prediction horizon, discount factor).

In lines 4–8, we create an initial set of trajectory rollouts by applying random actions to the system, starting from a state $\boldsymbol{x}_0$ sampled from $p(\boldsymbol{x}_0) = \mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{S}_0,)$. The `rollout` function in line 6 takes care of this. For each trajectory, the training inputs and targets are collected in the matrices `x` and `y`, respectively.

In lines 11–20, the three main modules are executed iteratively. First, the GP dynamics model is learned in the `trainDynModel` script, using the current training data `x,y` (line 13). Second, the policy is learned in the `learnPolicy` script, which updates the policy parameters using analytic policy evaluation and policy gradients. The third module, i.e., the application of the learned controller to the system, is encapsulated in the `applyController` script, which also augments the current data set `x,y` for training the GP model with the new trajectory.

1.4 Quick Start

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

Chapter 2

Software Package Overview

This software package implements the PILCO reinforcement learning framework [3]. The package contains the following directories

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

2.1 Main Modules

The main modules of the PILCO framework and their interplay are visualized in Figure 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 $\pi(\boldsymbol{x}_t)$
      ii. simulate dynamics (or apply control to real robot)
      iii. transition to state $\boldsymbol{x}_{t+1}$
2. `trainDynModel`
3. `learnPolicy`

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

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

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

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

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

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

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

2.1.1 applyController

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

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

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

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

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

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

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

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

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

3. Save the entire workspace (line 36).

2.1.2 trainDynModel

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

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

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

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

3. Train the GP (line 8)

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

2.1.3 learnPolicy

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

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

```

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

Function	Task		
`learnPolicy`			
`minimize`	Find $\theta^* \in \arg\min_\theta J(\theta)$		Policy Search
`value`	Compute $J(\theta) = \sum_{t=1}^{T} \mathrm{E}[c(\mathbf{x}_t)\vert\theta]$	Policy Evaluation	
`propagate` / `controller` / `gp*`	Compute $p(\mathbf{x}_{t+1}\vert\theta)$		
`cost.fcn`	Compute $\mathrm{E}[c(\mathbf{x}_{t+1})\vert\theta]$		
`BFGS`	Update θ	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 $\boldsymbol{\theta}^*$.
2. (optional) Plot overall optimization progress.
3. Long-term prediction of a state trajectory from $p(\boldsymbol{x}_0)$ using the learned policy (line 15) by calling `pred`. This prediction is equivalent to the last predicted trajectory during policy learning, i.e., the predicted state trajectory that belongs to the the learned controller.
4. The predicted state trajectory is used to compute the corresponding distribution over immediate costs (lines 16–17) by calling `calcCost`.
5. (optional) Plot the predicted immediate cost distribution as a function of the time steps (lines 19–25).

2.2 Working with a Real Robot

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

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

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

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

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

applyController_on_robot;   % collect data from robot

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

  % 2. Learn policy
  learnPolicy;

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

We successfully applied this procedure on different hardware platforms [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 $\boldsymbol{x}_* \sim \mathcal{N}(\boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*)$. We assume that the input dimension is D and the

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

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

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

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

- gp1: The underlying model is a sparse GP model. In particular, we use the SPGP/FITC approach by Snelson and Ghahramani [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 $p(\boldsymbol{x})$, $(D \times 1)$
3. s: Covariance matrix of the state distribution $p(\boldsymbol{x})$, $(D \times D)$

3.1.2 Output Arguments

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

- dMdm: $\partial M/\partial m \in \mathbb{R}^{E\times D}$ The derivative of the mean of the prediction with respect to the mean of the input distribution.

- dSdm: $\partial S/\partial m \in \mathbb{R}^{E^2\times D}$ The derivative of the covariance of the prediction with respect to the mean of the input distribution.

- dVdm: $\partial V/\partial m \in \mathbb{R}^{DE\times D}$ The derivative of V with respect to the mean of the input distribution.

- dMds: $\partial M/\partial s \in \mathbb{R}^{E\times D^2}$ The derivative of the mean of the prediction with respect to the covariance of the input distribution.

- dSds: $\partial S/\partial m \in \mathbb{R}^{E^2\times D^2}$ The derivative of the covariance of the prediction with respect to the covariance of the input distribution.

- dVds: $\partial V/\partial m \in \mathbb{R}^{DE\times D^2}$ The derivative of V with respect to the covariance of the input distribution.

As gp0d and gp1d are the functions used to propagate uncertainties through a GP dynamics model, they all compute these derivatives, see Table 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:

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

3.2 Controller

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

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

3.2.1 Interface

Let us explain the interface in more detail

3.2.1.1 Input Arguments

All controllers expect the following inputs

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

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

3.2.1.2 Output Arguments

All controller functions are expected to compute

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

3.3 Cost Functions

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

3.3.1 Interface for Scenario-specific Cost Functions

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

Input Arguments

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

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

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

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

Output Arguments

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

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

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

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

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

Input arguments:

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

Output arguments:

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

Implementation

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

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

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

In lines 11–18, we compute the expected cost $L = \mathbb{E}[c(\boldsymbol{x})]$ and its derivatives with respect to the mean and the covariance matrix of the input distribution. A detailed derivation can be found in [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));
```

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

In lines 19–33, we compute the variance $\mathbb{V}[c(\boldsymbol{x})]$ of the cost and its derivatives with respect to the mean and the covariance matrix of the input distribution. A detailed derivation can be found in [2]. If the variance $\mathbb{V}[c(\boldsymbol{x})] < 10^{-12}$, we set it to 0 for numerical reasons (line 24).

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

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

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

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

3.3.2.2 Quadratic Cost

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

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

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

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

Input arguments

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

Output arguments

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

Implementation

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

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

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

In lines 7–14, we compute the expected cost $L = \mathbb{E}[c(\boldsymbol{x})]$ and its derivatives with respect to the mean and the covariance matrix of the input distribution. A detailed derivation can be found in [2].

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

In lines 15–27, we compute the variance $\mathbb{V}[c(\boldsymbol{x})]$ of the cost and its derivatives with respect to the mean and the covariance matrix of the input distribution. A detailed derivation can be found in [2]. If the variance $\mathbb{V}[c(\boldsymbol{x})] < 10^{-12}$, we set it to 0 for numerical reasons (line 18).

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

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

Chapter 4

How to Create Your Own Scenario

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

4.1 Necessary Files

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

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

4.2 ODE Dynamics

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

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

Input arguments:

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

The input arguments are as follows:

- `t`: The current time.

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

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

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

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

- f: The applied force to the cart.

Output arguments:

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

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

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

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

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

Lines 7–17 compute either $\dot{\boldsymbol{z}}$ (lines 8–14) or the total mechanical energy (line 16). A principled derivation of the system dynamics and the mechanical energy can be found in [2].

4.3 Scenario-specific Settings

4.3.1 Adding Paths

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

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

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

4.3.2 Indices

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

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

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

where x is the position of the cart, $\dot{x}$ the corresponding velocity, and $\dot{\theta}$ and θ are the angular velocity and the angle of the pendulum.

The ODE-solver requires to know what parts of the state are used for the forward dynamics. These indices are captured by `odei` (line 30).

The predictive dimensions of the dynamics GP model are stored in `dyno` (line 32).

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

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

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

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

The `dyni` indices (line 34) describe which variables from the auxiliary state vector in Equation (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 θ, i.e., we no longer need θ in the inputs of the dynamics GP.
The `poli` indices (line 35) describe which state variables from the auxiliary state vector in Equation (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

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

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

4.3.3 General Settings

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

`dt` is the sampling time, i.e., $1/dt$ is the sampling frequency.
`T` is the length of the prediction horizon in seconds.
`H`$= T/dt$ is the length of the prediction horizon in time steps.
`mu0` is the mean of the distribution $p(\boldsymbol{x}_0)$ of the initial state. Here, `mu0`$= \mathbf{0}$ encodes that the cart is in the middle of the track (with zero velocity) with the pendulum hanging down (with zero angular velocity).
`S0` is the covariance matrix of $p(\boldsymbol{x}_0)$.
`N` is the number of times the loop in Figure 1.1 is executed.

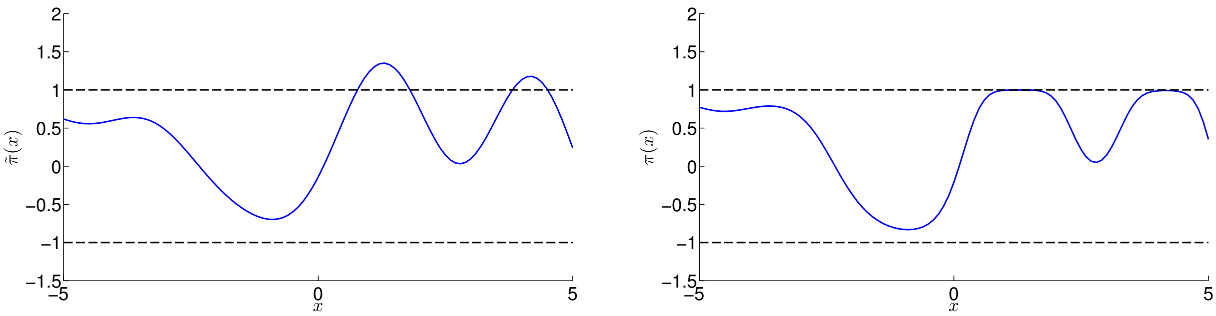


Figure 4.1: Preliminary policy $\tilde{\pi}$ and squashed policy π. The squashing function ensures that the control signals $\boldsymbol{u} = \pi(\boldsymbol{x})$ do not exceed the values $\pm\boldsymbol{u}_{\max}$.

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

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

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

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

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

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

4.3.4 Plant Structure

```
% 3. Plant structure
plant.dynamics = @dynamics_cp;                    % dynamics ode function
plant.noise = diag(ones(1,4)*0.01.^2);            % measurement noise
plant.dt = dt;
plant.ctrl = @zoh;                                % controler 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., $1/dt$ is the sampling frequency.
plant.ctrl is the controller to be applied. Here, @zoh implements a zero-order-hold controller. Other options are @foh (first-order-hold) and @lag (first-order-lag). For more information, have a look at base/simulate.m.
plant.odei–plant.difi copy the indices defined earlier into the plant structure.
plant.prop requires a function handle to the function that computes $p(\boldsymbol{x}_{t+1})$ from $p(\boldsymbol{x}_t)$, i.e., a one-step (probabilistic) prediction. In this software package, we implemented a fairly generic function called propagated, which computes the predictive state distribution $p(\boldsymbol{x}_{t+1})$ and the partial derivatives that are required for gradient-based policy search. For details about the gradients, we refer to [3].

4.3.5 Policy Structure

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

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

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

which is the third-order Fourier series expansion of a trapezoidal wave, normalized to the interval $[-u_{\max}, -u_{\max}]$.
policy.maxU (line 63) defines the maximum force value $u_{\max}$ in Equation (4.3). We assume that $u \in [-u_{\max}, -u_{\max}]$.
In lines 65–66, we augment the original state by $[\sin\theta, \cos\theta]$ by means of gTrig.m, where the indices of the angles θ are stored in plant.angi. We compute a Gaussian approximation to the joint distribution $p(\boldsymbol{x}, \sin\theta, \cos\theta)$. The representation of angles θ by means of $[\sin\theta, \cos\theta]$ avoids discontinuities and automatically takes care of the "wrap-around condition", i.e., $\theta \equiv \theta + 2k\pi, k \in \mathbb{Z}$.
The following lines are used to initialize the policy parameters. The policy parameters are generally stored in policy.p. We distinguish between three types of policy parameters:

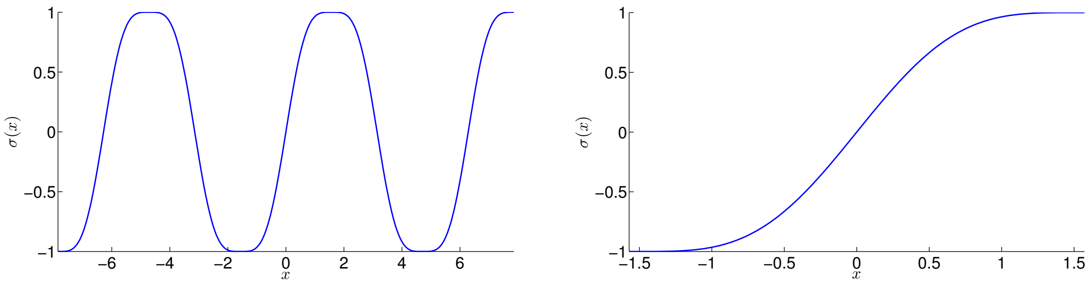


Figure 4.2: Squashing function.

- `policy.p.inputs`: These values play the role of the training inputs of a GP and correspond to the centers $\boldsymbol{c}_i$ of the policy in Equation (4.4). We sample the initial locations of the centers from the initial state distribution $p(\boldsymbol{x}_0)$. We compute this initial state distribution in lines 65–66, where we account for possible angle representations (`plant.angi`) of the state. If `plant.angi` is empty, lines 65–66 do not do anything and `mm`=`mu0` and `ss`=`S0`.

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

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

  – Length-scales: The length-scales weight the dimensions of the state that are passed to the policy (see `poli`). In our case these are: $x, \dot{x}, \dot{\theta}, \sin\theta, \cos\theta$. We initialize the first three length-scales to 1. These values largely depend on the scale of the input data. In our example, the cart position and velocity are measured in m and m/s, respectively, the angular velocity is measured in rad/s. The last two length-scales scale trigonometric values, i.e., $\sin(\theta)$ and $\cos(\theta)$. Since these trigonometric functions map their argument into the interval $[-1, 1]$, we choose a length-scale of 0.7, which is somewhat smaller than unity.

  – Signal variance: We set the signal variance of the controller $\tilde{\pi}$ to 1. Note that we squash $\tilde{\pi}$ through σ, see Equation (4.3). To exploit the full potential of the squashing function σ, it is sufficient to cover the domain $[-\pi/2, \pi/2]$. Therefore, we initialize the signal variance to 1.

  – Noise variance: The noise variance is only important only important as a relative factor to the signal variance. This ratio essentially determines how smooth the policy is. We initialize the noise variance to 0.01^2.

4.3.6 Cost Function Structure

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

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

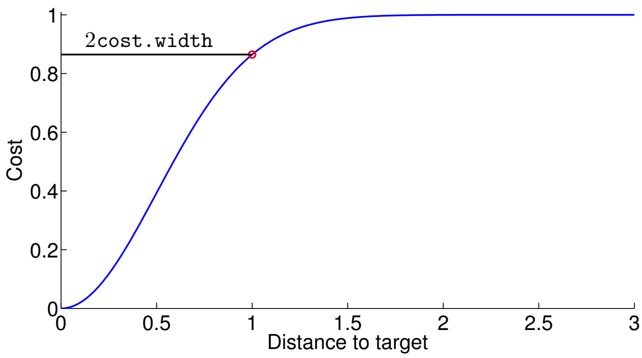


Figure 4.3: In the saturating cost function, see Equation (4.6), `cost.width` determines determines how far away from the target a cost $c < 1$ can be attained.

```
78 cost.expl =  0.0;                          % exploration parameter (UCB)
79 cost.angle = plant.angi;                   % index of angle (for cost function)
80 cost.target = [0 0 0 pi]';                 % target state
```

In line 74, we store the function handle `@loss_cp` in `cost.fcn`. The function `loss_cp` implements a saturating cost function (an unnormalized Gaussian subtracted from 1) with spread σ_c, i.e.,

$$c(\boldsymbol{x}) = 1 - \exp\big(-\tfrac{1}{2\sigma_c^2}\|\boldsymbol{x} - \boldsymbol{x}_{\text{target}}\|^2\big) \quad \in [0, 1] \tag{4.6}$$

where $\boldsymbol{x}_{\text{target}}$ is a target state.
We set the discount factor `cost.gamma` to 1 (line 75) as we look at a finite-horizon problem.
The following parameters are specific to the cost function:

- `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.
- `cost.width` (line 77) is the spread/width σ_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 `cost.width`$= \|\boldsymbol{\mu}_0 - \boldsymbol{x}_{\text{target}}\|/10$.
- `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.
- `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.
- `cost.target` (line 80) defines the target state $\boldsymbol{x}_{\text{target}}$. 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.

```
81 % 6. Dynamics model structure
82 dynmodel.fcn = @gp1d;                  % function for GP predictions
83 dynmodel.train = @train;               % function to train dynamics model
84 dynmodel.induce = zeros(300,0,1);      % shared inducing inputs (sparse GP)
85 trainOpt = [300 500];                  % defines the max. number of line searches
86                                        % when training the GP dynamics models
87                                        % trainOpt(1): full GP,
88                                        % 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[2], $b = 0$ 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`[3], a non-convex gradient-based optimizer. These optional parameters are stored in a structure `opt`.

```
89 % 7. Parameters for policy optimization
90 opt.length = 150;                         % max. number of line searches
91 opt.MFEPLS = 30;                          % max. number of function evaluations
92                                           % per line search
93 opt.verbosity = 1;                        % verbosity: specifies how much
94                                           % information is displayed during
95                                           % 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[4]).

[2]If the size of the training set exceeds a, the full GP automatically switches to its sparse approximation.
[3]http://www.gaussianprocess.org/gpml/code/matlab/util/minimize.m
[4]This is great for debugging.

4.3.9 Plotting Parameters

```
96 % 8. Plotting verbosity
97 plotting.verbosity = 1;            % 0: no plots
98                                    % 1: some plots
99                                    % 2: all plots
```

`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.

```
100 % 9. Some initializations
101 x = []; y = [];
102 fantasy.mean = cell(1,N); fantasy.std = cell(1,N);
103 realCost = cell(1,N); M = cell(N,1); Sigma = cell(N,1);
```

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

$$c(\boldsymbol{x}) = \frac{1}{\#\texttt{cost.cw}} \sum_{i=1}^{\#\texttt{cost.cw}} \left(1 - \exp\left(-\tfrac{1}{2(\sigma_c^{(i)})^2}\|\boldsymbol{x} - \boldsymbol{x}_{\text{target}}\|^2\right)\right) \tag{4.7}$$

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

The mean and the variance of the cost $c(\boldsymbol{x})$ are computed by averaging over the Gaussian state distribution $p(\boldsymbol{x}) = \mathcal{N}(\boldsymbol{x}|\boldsymbol{m}, \boldsymbol{S})$ with mean $\boldsymbol{m}$ and covariance matrix $\boldsymbol{S}$. 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

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

Input arguments:

```
cost            cost structure
  .p            length of pendulum                          [1 x  1 ]
  .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

$$\boldsymbol{j} = [x, \dot{x}, \dot{\theta}, \theta, \sin\theta, \cos\theta]^\top . \tag{4.8}$$

Lines 8–10 initialize the output arguments to 0.
In the following lines, the distance $\boldsymbol{x} - \boldsymbol{x}_{\text{target}}$ 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`.
In line 12, the matrix $\boldsymbol{Q}$ is computed, such that $(\boldsymbol{j} - \boldsymbol{j}_{\text{target}})^\top \boldsymbol{Q}(\boldsymbol{j} - \boldsymbol{j}_{\text{target}})$ is the squared Euclidean distance between the tip of the pendulum in the current state and the target state. For $\boldsymbol{x}_{\text{target}} = [0, *, *, \pi]^\top$, i.e., the pendulum is balanced in the inverted position in the middle of the track, the Euclidean distance is given as

$$\|\boldsymbol{x} - \boldsymbol{x}_{\text{target}}\|^2 = x^2 + 2xl\sin\theta + 2l^2 + 2l^2\cos\theta = (\boldsymbol{j} - \boldsymbol{j}_{\text{target}})^\top \boldsymbol{Q}(\boldsymbol{j} - \boldsymbol{j}_{\text{target}}) , \tag{4.9}$$

$$\boldsymbol{Q} = \begin{bmatrix} 1 & 0 & 0 & 0 & l & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ l & 0 & 0 & 0 & l^2 & 0 \\ 0 & 0 & 0 & 0 & 0 & l^2 \end{bmatrix} . \tag{4.10}$$

Note that at this point only the $\boldsymbol{Q}$-matrix is determined.

```
14 % 3. Trigonometric augmentation
15 if D1-D0 > 0
16   % augment target
17   target = [cost.target(:); gTrig(cost.target(:), 0*s, cost.angle)];
18 
19   % augment state
20   i = 1:D0; k = D0+1:D1;
21   [M(k) S(k,k) C mdm sdm Cdm mds sds Cds] = gTrig(M(i),S(i,i),cost.angle);
22 
23   % compute derivatives (for augmentation)
24   X = reshape(1:D1*D1,[D1 D1]); XT = X';                % vectorized indices
25   I=0*X; I(i,i)=1; ii=X(I==1)'; I=0*X; I(k,k)=1; kk=X(I==1)';
26   I=0*X; I(i,k)=1; ik=X(I==1)'; ki=XT(I==1)';
27 
28   Mdm(k,:)  = mdm*Mdm(i,:) + mds*Sdm(ii,:);                      % chainrule
29   Mds(k,:)  = mdm*Mds(i,:) + mds*Sds(ii,:);
30   Sdm(kk,:) = sdm*Mdm(i,:) + sds*Sdm(ii,:);
31   Sds(kk,:) = sdm*Mds(i,:) + sds*Sds(ii,:);
32   dCdm      = Cdm*Mdm(i,:) + Cds*Sdm(ii,:);
33   dCds      = Cdm*Mds(i,:) + Cds*Sds(ii,:);
34 
35   S(i,k) = S(i,i)*C; S(k,i) = S(i,k)';                        % off-diagonal
36   SS = kron(eye(length(k)),S(i,i)); CC = kron(C',eye(length(i)));
37   Sdm(ik,:) = SS*dCdm + CC*Sdm(ii,:); Sdm(ki,:) = Sdm(ik,:);
38   Sds(ik,:) = SS*dCds + CC*Sds(ii,:); Sds(ki,:) = Sds(ik,:);
39 end
```

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

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

In line 21, the state distribution $p(\boldsymbol{x})$ is probabilistically augmented to $p(\boldsymbol{j})$, where $\boldsymbol{j}$ is given in Equation (4.1). Note that $p(\boldsymbol{j})$ cannot be computed analytically. Instead, we compute the mean and covariance of $p(\boldsymbol{j})$ exactly and approximate $p(\boldsymbol{j})$ 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 $p(\boldsymbol{j})$ and the cross-covariance $\text{cov}[\boldsymbol{x}, \boldsymbol{j}]$ with respect to the mean and covariance of $p(\boldsymbol{x})$ using the chain rule.

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

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

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

```
53      dLds = dLds + b/sqrt(s2) * ( s2dM(:)'*Mds + s2dS(:)'*Sds )/2;
54    end
55  end
56
57  % normalize
58  n = length(cw); L = L/n; dLdm = dLdm/n; dLds = dLds/n; S = S/n;
```

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 $p(\boldsymbol{j})$, 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 $p(\boldsymbol{x})$ by applying the chain rule.
If exploration is desired (line 50), we add $\kappa\sqrt{\mathbb{V}[c(\boldsymbol{x})]}$ to the $\mathbb{E}[c(\boldsymbol{x})]$ 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.

```
% Loop over states in trajectory (= time steps)
for r = 1:size(xx,1)
  if exist('j','var') && ~isempty(M{j})
    draw_cp(latent{j}(r,1), latent{j}(r,4), latent{j}(r,end), cost,  ...
      ['trial # ' num2str(j+J) ', T=' num2str(H*dt) ' sec'], ...
      ['total experience (after this trial): ' num2str(dt*size(x,1)) ...
      ' sec'], M{j}(:,r), Sigma{j}(:,:,r));
  else
     draw_cp(latent{jj}(r,1), latent{jj}(r,4), latent{jj}(r,end), cost,  ...
      ['(random) trial # ' num2str(jj) ', T=' num2str(H*dt) ' sec'], ...
      ['total experience (after this trial): ' num2str(dt*size(x,1)) ...
      ' sec'])
  end
  pause(dt);
end
```

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:

```
x          position of the cart
theta      angle of pendulum
force      force applied to cart
cost       cost structure
  .fcn     function handle (it is assumed to use saturating cost)
  .<>      other fields that are passed to cost
M          (optional) mean of state
S          (optional) covariance of state
text1      (optional) text field 1
text2      (optional) text field 2
```

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
15 end
16
17 mu0Sim(odei,:) = mu0; S0Sim(odei,odei) = S0;
18 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.

```
19 % 3. Controlled learning (N iterations)
20 for j = 1:N
21   trainDynModel;   % train (GP) dynamics model
22   learnPolicy;     % learn policy
23   applyController; % apply controller to system
24   disp(['controlled trial # ' num2str(j)]);
25   if plotting.verbosity > 0;      % visualization of trajectory
26     if ~ishandle(1); figure(1); else set(0,'CurrentFigure',1); end; clf(1);
27     draw_rollout_cp;
28   end
29 end
```

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:

```
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

$$SNR = \frac{\sqrt{\sigma_f^2}}{\sqrt{\sigma_{\text{noise}}^2}},$$

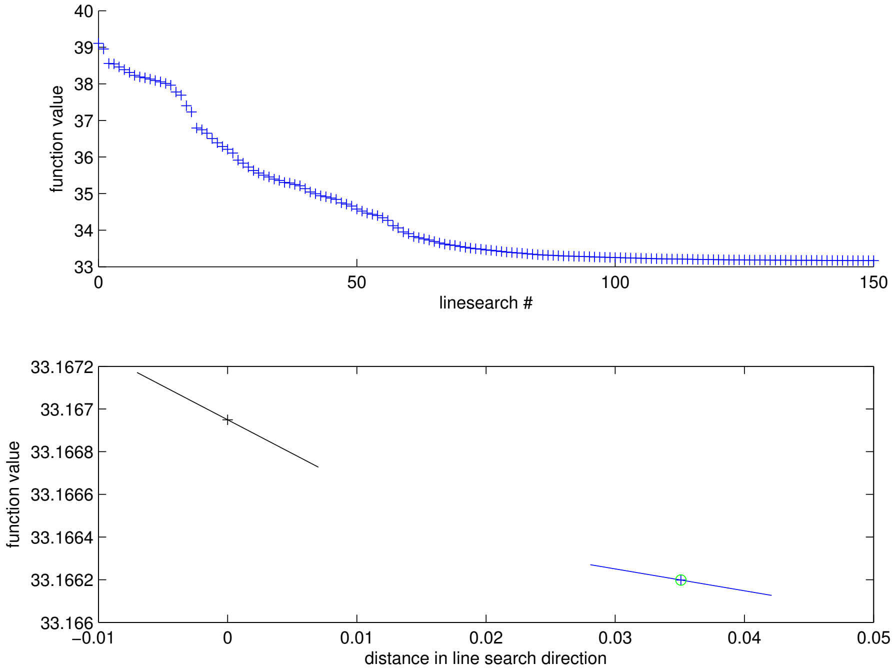


Figure 4.4: Typical display during policy learning: The top subplot shows the overall progress of policy learning as a function of the number of line searches. In this example, the initial policy caused a total expected cost $J(\boldsymbol{\theta})$ of 39.11; the policy after 150 line search searches 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`.

where σ_f^2 is the variance of the underlying function and σ_{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`).

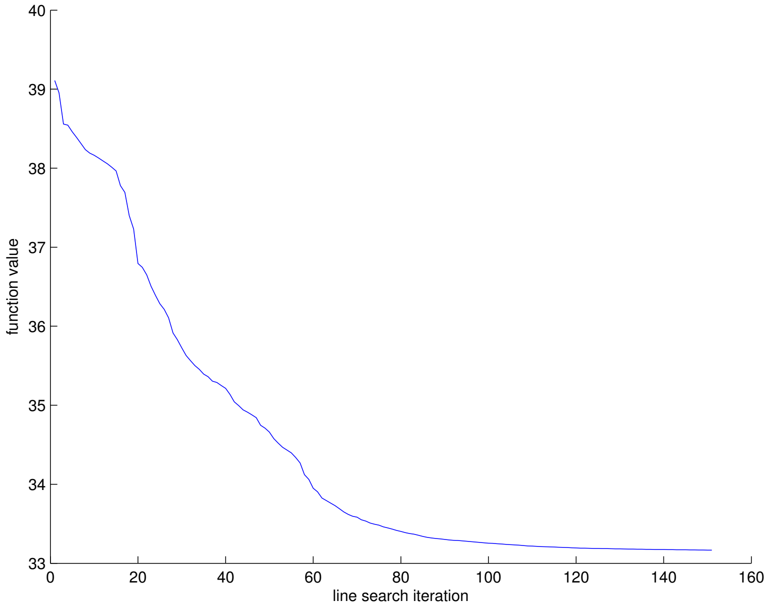


Figure 4.5: Overall progress of a policy search with 150 line searches.

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 `xx` 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 $\boldsymbol{x}_0 \sim p(\boldsymbol{x}_0)$. 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 $u_{\max}$ and $-u_{\max}$ 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.

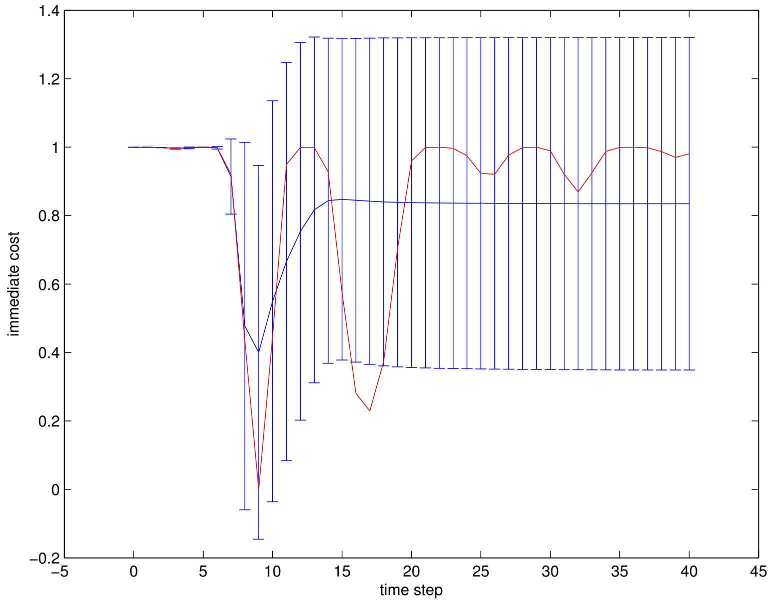
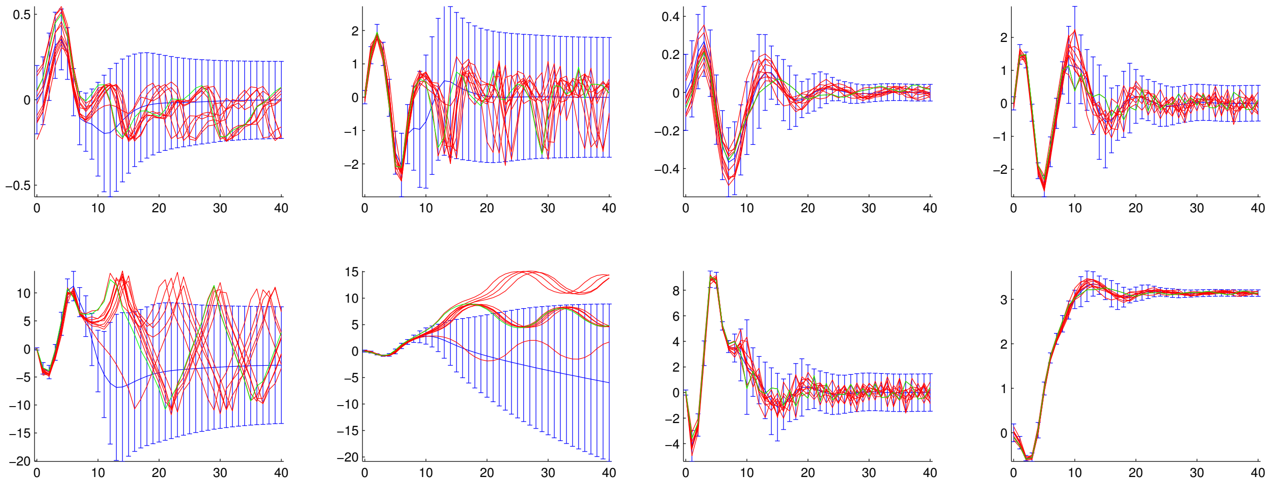


Figure 4.6: Predicted immediate cost (blue error bar) and corresponding incurred cost when applying the policy (red) for each time step of the prediction horizon.

Table 4.1: `plotting.verbosity` Overview.

	Figure 4.5	Figure 4.6	Figure 4.7	Figure 4.8
`plotting.verbosity=0`	✗	✗	✗	✗
`plotting.verbosity=1`	✗	✓	✗	✓
`plotting.verbosity=2`	✓	✓	✓	✓



(a) Early stages of learning. (b) Final stages of learning.

Figure 4.7: Example trajectories. For each predictive dimension (`dyno`), ten state trajectories are displayed, which occurred when the same policy $\pi(\boldsymbol{\theta}^*)$ is applied to the system. Note that the initial state differs as it is sampled from the start state distribution, i.e., $\boldsymbol{x}_0 \sim p(\boldsymbol{x}_0)$. 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.

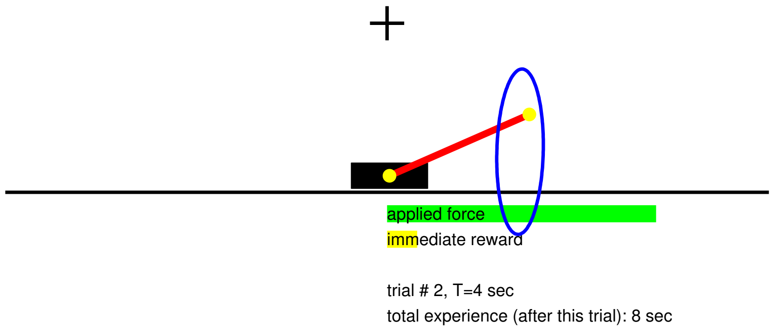


Figure 4.8: Visualization of a trajectory of the cart-pole swing-up during learning. The cart runs on an (infinite) track, the pendulum is freely swinging. The cross denotes the target location for balancing the tip of the pendulum, and the blue ellipse represents the 95% confidence bound of a t-step 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 $u_{\max}$ and $-u_{\max}$ 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 5.1: State and control space dimensions in the implemented scenarios.

Task	State Space	Control Space
Pendulum	$\mathbb{R}^2$	$\mathbb{R}$
Pendubot	$\mathbb{R}^4$	$\mathbb{R}$
Double Pendulum	$\mathbb{R}^4$	$\mathbb{R}^2$
Cart-Pole	$\mathbb{R}^4$	$\mathbb{R}$
Cart-Double Pendulum	$\mathbb{R}^6$	$\mathbb{R}$
Unicycling	$\mathbb{R}^{12}$	$\mathbb{R}^2$

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 φ 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

$$x = \tfrac{1}{2} l \sin\varphi\,,$$
$$y = -\tfrac{1}{2} l \cos\varphi,$$

and the squared velocity of the midpoint of the pendulum is

$$v^2 = \dot{x}^2 + \dot{y}^2 = \tfrac{1}{4} l^2 \dot{\varphi}^2\,.$$

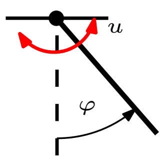


Figure 5.1: Pendulum.

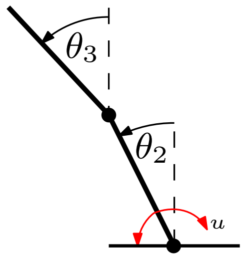


Figure 5.2: Pendubot.

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

$$L = T - V = \tfrac{1}{2}mv^2 + \tfrac{1}{2}I\dot{\varphi}^2 + \tfrac{1}{2}mlg\cos\varphi\,, \tag{5.1}$$

where $g = 9.82\,\mathrm{m/s^2}$ 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,

$$\begin{aligned}\frac{\partial L}{\partial \dot{\varphi}} &= \tfrac{1}{4}ml^2\dot{\varphi} + I\dot{\varphi}\\ \frac{\partial L}{\partial \varphi} &= -\tfrac{1}{2}mlg\sin\varphi\end{aligned}$$

yield

$$\ddot{\varphi}(\tfrac{1}{4}ml^2 + I) + \tfrac{1}{2}mlg\sin\varphi = u - b\dot{\varphi}\,,$$

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

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

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 s 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, θ_2 and θ_3, are measured anti-clockwise from upright. An applied external torque u controls the first joint. Typical values are: $m_2 = 0.5$kg, $m_3 = 0.5$kg $l_2 = 0.6$m, $l_3 = 0.6$m.

The Cartesian coordinates x_2, y_2 and x_3, y_3 of the midpoints of the pendulum elements are

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} -\frac{1}{2}l_2 \sin\theta_2 \\ \frac{1}{2}l_2\cos\theta_2 \end{bmatrix}, \quad \begin{bmatrix} x_3 \\ y_3 \end{bmatrix} = \begin{bmatrix} -l_2\sin\theta_2 - \frac{1}{2}l_3\sin\theta_3 \\ l_2\cos\theta_2 + \frac{1}{2}l_3\cos\theta_3 \end{bmatrix}, \tag{5.2}$$

and the squared velocities of the pendulum midpoints are

$$v_2^2 = \dot{x}_2^2 + \dot{y}_2^2 = \tfrac{1}{4}l_2^2\dot{\theta}_2^2 \tag{5.3}$$

$$v_3^2 = \dot{x}_3^2 + \dot{y}_3^2 = l_2^2\dot{\theta}_2^2 + \tfrac{1}{4}l_3^2\dot{\theta}_3^2 + l_2 l_3\dot{\theta}_2\dot{\theta}_3\cos(\theta_2-\theta_3). \tag{5.4}$$

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

$$L = T - V = \tfrac{1}{2}m_2v_2^2 + \tfrac{1}{2}m_3v_3^2 + \tfrac{1}{2}I_2\dot{\theta}_2^2 + \tfrac{1}{2}I_3\dot{\theta}_3^2 - m_2gy_2 - m_3gy_3\,,$$

where the angular moment of inertia around the pendulum midpoint is $I = \frac{1}{12}ml^2$, 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

$$\begin{aligned} L &= \tfrac{1}{8}m_2l_2^2\dot{\theta}_2^2 + \tfrac{1}{2}m_3\big(l_2^2\dot{\theta}_2^2 + \tfrac{1}{4}l_3^2\dot{\theta}_3^2 + l_2l_3\dot{\theta}_2\dot{\theta}_3\cos(\theta_2-\theta_3)\big) \\ &\quad + \tfrac{1}{2}I_2\dot{\theta}_2^2 + \tfrac{1}{2}I_3\dot{\theta}_3^2 - \tfrac{1}{2}m_2gl_2\cos\theta_2 - m_3g(l_2\cos\theta_2 + \tfrac{1}{2}l_3\cos\theta_3)\,. \end{aligned}$$

The equations of motion are

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = Q_i\,, \tag{5.5}$$

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,

$$\begin{aligned} \frac{\partial L}{\partial \dot{\theta}_2} &= l_2^2\dot{\theta}_2(\tfrac{1}{4}m_2 + m_3) + \tfrac{1}{2}m_3l_2l_3\dot{\theta}_3\cos(\theta_2-\theta_3) + I_2\dot{\theta}_2\,, \\ \frac{\partial L}{\partial \theta_2} &= -\tfrac{1}{2}m_3l_2l_3\dot{\theta}_2\dot{\theta}_3\sin(\theta_2-\theta_3) + (\tfrac{1}{2}m_2 + m_3)gl_2\sin\theta_2\,, \\ \frac{\partial L}{\partial \dot{\theta}_3} &= m_3l_3\big(\tfrac{1}{4}l_3\dot{\theta}_3 + \tfrac{1}{2}l_2\dot{\theta}_2\cos(\theta_2-\theta_3)\big) + I_3\dot{\theta}_3\,, \\ \frac{\partial L}{\partial \theta_3} &= \tfrac{1}{2}m_3l_3\big(l_2\dot{\theta}_2\dot{\theta}_3\sin(\theta_2-\theta_3) + g\sin\theta_3\big) \end{aligned}$$

lead to the equations of motion

$$\begin{aligned} u &= \ddot{\theta}_2\big(l_2^2(\tfrac{1}{4}m_2 + m_3) + I_2\big) + \ddot{\theta}_3\tfrac{1}{2}m_3l_3l_2\cos(\theta_2-\theta_3) \\ &\quad + l_2\big(\tfrac{1}{2}m_3l_3\dot{\theta}_3^2\sin(\theta_2-\theta_3) - g\sin\theta_2(\tfrac{1}{2}m_2+m_3)\big)\,, \\ 0 &= \ddot{\theta}_2\tfrac{1}{2}l_2l_3m_3\cos(\theta_2-\theta_3) + \ddot{\theta}_3(\tfrac{1}{4}m_3l_3^2 + I_3) - \tfrac{1}{2}m_3l_3\big(l_2\dot{\theta}_2^2\sin(\theta_2-\theta_3) + g\sin\theta_3\big)\,. \end{aligned}$$

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

$$\begin{bmatrix} l_2^2(\frac{1}{4}m_2+m_3) + I_2 & \frac{1}{2}m_3l_3l_2\cos(\theta_2-\theta_3) \\ \frac{1}{2}l_2l_3m_3\cos(\theta_2-\theta_3) & \frac{1}{4}m_3l_3^2 + I_3 \end{bmatrix} \begin{bmatrix} \ddot{\theta}_2 \\ \ddot{\theta}_3 \end{bmatrix} = \begin{bmatrix} c_2 \\ c_3 \end{bmatrix}$$

for $\ddot{\theta}_2$ and $\ddot{\theta}_3$, where

$$\begin{bmatrix} c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} -l_2\big(\frac{1}{2}m_3l_3\dot{\theta}_3^2\sin(\theta_2-\theta_3) - g\sin\theta_2(\frac{1}{2}m_2+m_3)\big) + u \\ \frac{1}{2}m_3l_3\big(l_2\dot{\theta}_2^2\sin(\theta_2-\theta_3) + g\sin\theta_3\big) \end{bmatrix}.$$

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

$$
\begin{aligned}
u_1 &= \ddot\theta_2\big(l_2^2(\tfrac{1}{4}m_2+m_3)+I_2\big)+\ddot\theta_3\tfrac{1}{2}m_3l_3l_2\cos(\theta_2-\theta_3)\\
&\quad + l_2\big(\tfrac{1}{2}m_3l_3\dot\theta_3^2\sin(\theta_2-\theta_3)-g\sin\theta_2(\tfrac{1}{2}m_2+m_3)\big)\,,\\
u_2 &= \ddot\theta_2\tfrac{1}{2}l_2l_3m_3\cos(\theta_2-\theta_3)+\ddot\theta_3(\tfrac{1}{4}m_3l_3^2+I_3)-\tfrac{1}{2}m_3l_3\big(l_2\dot\theta_2^2\sin(\theta_2-\theta_3)+g\sin\theta_3\big)\,.
\end{aligned}
$$

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

$$
\begin{bmatrix} l_2^2(\frac{1}{4}m_2+m_3)+I_2 & \frac{1}{2}m_3l_3l_2\cos(\theta_2-\theta_3)\\ \frac{1}{2}l_2l_3m_3\cos(\theta_2-\theta_3) & \frac{1}{4}m_3l_3^2+I_3\end{bmatrix}\begin{bmatrix}\ddot\theta_2\\ \ddot\theta_3\end{bmatrix}=\begin{bmatrix}c_2\\ c_3\end{bmatrix}
$$

for $\ddot\theta_2$ and $\ddot\theta_3$, where

$$
\begin{bmatrix}c_2\\ c_3\end{bmatrix}=\begin{bmatrix}-l_2\big(\frac{1}{2}m_3l_3\dot\theta_3^2\sin(\theta_2-\theta_3)-g\sin\theta_2(\frac{1}{2}m_2+m_3)\big)+u_1\\ \frac{1}{2}m_3l_3\big(l_2\dot\theta_2^2\sin(\theta_2-\theta_3)+g\sin\theta_3\big)+u_2\end{bmatrix}\,.
$$

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 pedulum with mass m_2 and length l, which swings freely in the plane. The pendulum angle θ_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: $m_1 = 0.5\,\mathrm{kg}$, $m_2 = 0.5\,\mathrm{kg}$, $l = 0.6\,\mathrm{m}$ and $b = 0.1\,\mathrm{N/m/s}$.

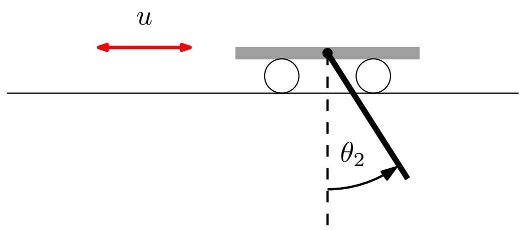


Figure 5.3: Cart-pole system.

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

$$
\begin{aligned}
x_2 &= x_1+\tfrac{1}{2}l\sin\theta_2\,,\\
y_2 &= -\tfrac{1}{2}l\cos\theta_2,
\end{aligned}
$$

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

$$
\begin{aligned}
v_1^2 &= \dot x_1^2\\
v_2^2 &= \dot x_2^2+\dot y_2^2 = \dot x_1^2+\tfrac{1}{4}l^2\dot\theta_2^2+l\dot x_1\dot\theta_2\cos\theta_2\,,
\end{aligned}
$$

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

$$
L = T-V = \tfrac{1}{2}m_1v_1^2+\tfrac{1}{2}m_2v_2^2+\tfrac{1}{2}I\dot\theta_2^2-m_2gy_2\,, \tag{5.6}
$$

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

$$
L = \tfrac{1}{2}(m_1+m_2)\dot x_1^2+\tfrac{1}{6}m_2l^2\dot\theta_2^2+\tfrac{1}{2}m_2l(\dot x_1\dot\theta_2+g)\cos\theta_2\,.
$$

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

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = Q_i\,, \tag{5.7}$$

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,

$$\begin{aligned}
\frac{\partial L}{\partial \dot{x}_1} &= (m_1 + m_2)\dot{x}_1 + \tfrac{1}{2}m_2 l\dot{\theta}_2\cos\theta_2\,,\\
\frac{\partial L}{\partial x_1} &= 0\,,\\
\frac{\partial L}{\partial \dot{\theta}_2} &= \tfrac{1}{3}m_2 l^2\dot{\theta}_2 + \tfrac{1}{2}m_2 l\dot{x}_1\cos\theta_2\,,\\
\frac{\partial L}{\partial \theta_2} &= -\tfrac{1}{2}m_2 l(\dot{x}_1\dot{\theta}_2 + g),
\end{aligned}$$

lead to the equations of motion

$$\begin{aligned}
(m_1 + m_2)\ddot{x}_1 + \tfrac{1}{2}m_2 l\ddot{\theta}_2\cos\theta_2 - \tfrac{1}{2}m_2 l\dot{\theta}_2^2\sin\theta_2 &= u - b\dot{x}_1\,,\\
2l\ddot{\theta}_2 + 3\ddot{x}_1\cos\theta_2 + 3g\sin\theta_2 &= 0\,.
\end{aligned}$$

Collecting the four variables $\boldsymbol{z} = [x_1, \dot{x}_1, \dot{\theta}_2, \theta_2]^\top$ the equations of motion can be conveniently expressed as four coupled ordinary differential equations

$$\frac{d\boldsymbol{z}}{dt} = \begin{bmatrix} z_2 \\ \dfrac{2m_2 l z_3^2\sin z_4 + 3m_2 g\sin z_4\cos z_4 + 4u - 4bz_2}{4(m_1+m_2) - 3m_2\cos^2 z_4} \\ \dfrac{-3m_2 l z_3^2\sin z_4\cos z_4 - 6(m_1+m_2)g\sin z_4 - 6(u - bz_2)\cos z_4}{4l(m_1+m_2) - 3m_2 l\cos^2 z_4} \\ z_3 \end{bmatrix},$$

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, θ_2 and θ_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: $m_1 = 0.5\,\text{kg}$, $m_2 = 0.5\,\text{kg}$, $m_3 = 0.5\,\text{kg}$ $l_2 = 0.6\,\text{m}$, $l_3 = 0.6\,\text{m}$, and $b = 0.1\,\text{Ns/m}$.
The coordinates, x_2, y_2 and x_3, y_3 of the midpoint of the pendulum elements are

$$\begin{aligned}
\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} &= \begin{bmatrix} x_1 - \frac{1}{2}l_2\sin\theta_2 \\ \frac{1}{2}l_2\cos\theta_2 \end{bmatrix}\\
\begin{bmatrix} x_3 \\ y_3 \end{bmatrix} &= \begin{bmatrix} x_1 - l_2\sin\theta_2 - \frac{1}{2}l_3\sin\theta_3 \\ y_3 = l_2\cos\theta_2 + \frac{1}{2}l_3\cos\theta_3 \end{bmatrix}.
\end{aligned}$$

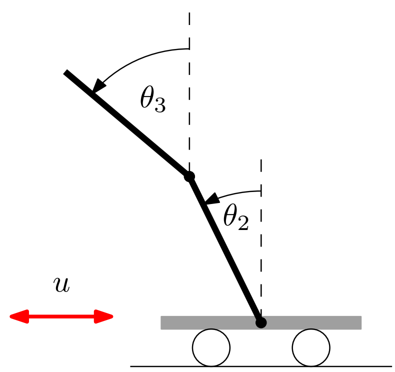


Figure 5.4: Cart-double pendulum.

The squared velocities of the cart and the pendulum midpoints are

$$
\begin{aligned}
v_1^2 &= \dot{x}_1^2\,,\\
v_2^2 &= \dot{x}_2^2 + \dot{y}_2^2 = \dot{x}_1^2 - l_2\dot{x}_1\dot{\theta}_2\cos\theta_2 + \tfrac{1}{4}l_2^2\dot{\theta}_2^2\,,\\
v_3^2 &= \dot{x}_3^2 + \dot{y}_3^2 = \dot{x}_1^2 + l_2^2\dot{\theta}_2^2 + \tfrac{1}{4}l_3^2\dot{\theta}_3^2 - 2l_2\dot{x}_1\dot{\theta}_2\cos\theta_2 - l_3\dot{x}_1\dot{\theta}_3\cos\theta_3 + l_2l_3\dot{\theta}_2\dot{\theta}_3\cos(\theta_2-\theta_3)\,.
\end{aligned}
$$

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

$$
\begin{aligned}
L = T - V &= \tfrac{1}{2}m_1v_1^2 + \tfrac{1}{2}m_2v_2^2 + \tfrac{1}{2}m_3v_3^2 + \tfrac{1}{2}I_2\dot{\theta}_2^2 + \tfrac{1}{2}I_3\dot{\theta}_3^2 - m_2gy_2 - m_3gy_3\\
&= \tfrac{1}{2}(m_1+m_2+m_3)\dot{x}_1^2 - \tfrac{1}{2}m_2l_2\dot{x}\dot{\theta}_2\cos(\theta_2) - \tfrac{1}{2}m_3\big(2l_2\dot{x}\dot{\theta}_2\cos(\theta_2) + l_3\dot{x}_1\dot{\theta}_3\cos(\theta_3)\big)\\
&\quad + \tfrac{1}{8}m_2l_2^2\dot{\theta}_2^2 + \tfrac{1}{2}I_2\dot{\theta}_2^2 + \tfrac{1}{2}m_3(l_2^2\dot{\theta}_2^2 + \tfrac{1}{4}l_3^2\dot{\theta}_3^2 + l_2l_3\dot{\theta}_2\dot{\theta}_3\cos(\theta_2-\theta_3)) + \tfrac{1}{2}I_3\dot{\theta}_3^2\\
&\quad - \tfrac{1}{2}m_2gl_2\cos(\theta_2) - m_3g(l_2\cos(\theta_2) + \tfrac{1}{2}l_3\cos(\theta_3))\,.
\end{aligned}
$$

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

$$
\frac{d}{dt}\frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = Q_i\,, \tag{5.8}
$$

where Q_i are the non-conservative forces. We obtain the partial derivatives

$$
\begin{aligned}
\frac{\partial L}{\partial \dot{x}_1} &= (m_1+m_2+m_3)\dot{x}_1 - (\tfrac{1}{2}m_2+m_3)l_2\dot{\theta}_2\cos\theta_2 - \tfrac{1}{2}m_3l_3\dot{\theta}_3\cos\theta_3\,,\\
\frac{\partial L}{\partial x_1} &= 0\,,\\
\frac{\partial L}{\partial \dot{\theta}_2} &= (m_3l_2^2 + \tfrac{1}{4}m_2l_2^2 + I_2)\dot{\theta}_2 - (\tfrac{1}{2}m_2+m_3)l_2\dot{x}_1\cos\theta_2 + \tfrac{1}{2}m_3l_2l_3\dot{\theta}_3\cos(\theta_2-\theta_3)\,,\\
\frac{\partial L}{\partial \theta_2} &= (\tfrac{1}{2}m_2+m_3)l_2(\dot{x}_1\dot{\theta}_2 + g)\sin\theta_2 - \tfrac{1}{2}m_3l_2l_3\dot{\theta}_2\dot{\theta}_3\sin(\theta_2-\theta_3)\,,\\
\frac{\partial L}{\partial \dot{\theta}_3} &= m_3l_3\big[-\tfrac{1}{2}\dot{x}_1\cos\theta_3 + \tfrac{1}{2}l_2\dot{\theta}_2\cos(\theta_2-\theta_3) + \tfrac{1}{4}l_3\dot{\theta}_3\big] + I_3\dot{\theta}_3\,,\\
\frac{\partial L}{\partial \theta_3} &= \tfrac{1}{2}m_3l_3\big[(\dot{x}_1\dot{\theta}_3 + g)\sin\theta_3 + l_2\dot{\theta}_2\dot{\theta}_3\sin(\theta_2-\theta_3)\big]
\end{aligned}
$$

leading to the equations of motion

$$
\begin{aligned}
(m_1+m_2+m_3)\ddot{x}_1 + \tfrac{1}{2}m_2+m_3)l_2(\dot{\theta}_2^2\sin\theta_2 - \ddot{\theta}_2\cos\theta_2)&\\
+\tfrac{1}{2}m_3l_3(\dot{\theta}_3^2\sin\theta_3 - \ddot{\theta}_3\cos\theta_3) &= u - b\dot{x}_1\\
(m_3l_2^2 + I_2 + \tfrac{1}{4}m_2l_2^2)\ddot{\theta}_2 - (\tfrac{1}{2}m_2+m_3)l_2(\ddot{x}_1\cos\theta_2 + g\sin\theta_2)&\\
+\tfrac{1}{2}m_3l_2l_3[\ddot{\theta}_3\cos(\theta_2-\theta_3) + \dot{\theta}_3^2\sin(\theta_2-\theta_3)] &= 0\\
(\tfrac{1}{4}m_2l_3^2 + I_3)\ddot{\theta}_3 - \tfrac{1}{2}m_3l_3(\ddot{x}_1\cos\theta_3 + g\sin\theta_3)&\\
+\tfrac{1}{2}m_3l_2l_3[\ddot{\theta}_2\cos(\theta_2-\theta_3) - \dot{\theta}_2^2\sin(\theta_2-\theta_3)] &= 0
\end{aligned}
$$

These three linear equations in $(\ddot{x}_1, \ddot{\theta}_2, \ddot{\theta}_3)$ can be rewritten as the linear equation system

$$
\begin{bmatrix}
(m_1+m_2+m_3) & -\frac{1}{2}(m_2+2m_3)l_2\cos\theta_2 & -\frac{1}{2}m_3l_3\cos\theta_3\\
-(\frac{1}{2}m_2+m_3)l_2\cos\theta_2 & m_3l_2^2 + I_2 + \frac{1}{4}m_2l_2^2 & \frac{1}{2}m_3l_2l_3\cos(\theta_2-\theta_3)\\
-\frac{1}{2}m_3l_3\cos\theta_3 & \frac{1}{2}m_3l_2l_3\cos(\theta_2-\theta_3) & \frac{1}{4}m_2l_3^2 + I_3
\end{bmatrix}
\begin{bmatrix}\ddot{x}_1\\ \ddot{\theta}_2\\ \ddot{\theta}_3\end{bmatrix}
=
\begin{bmatrix}c_1\\ c_2\\ c_3\end{bmatrix},
$$

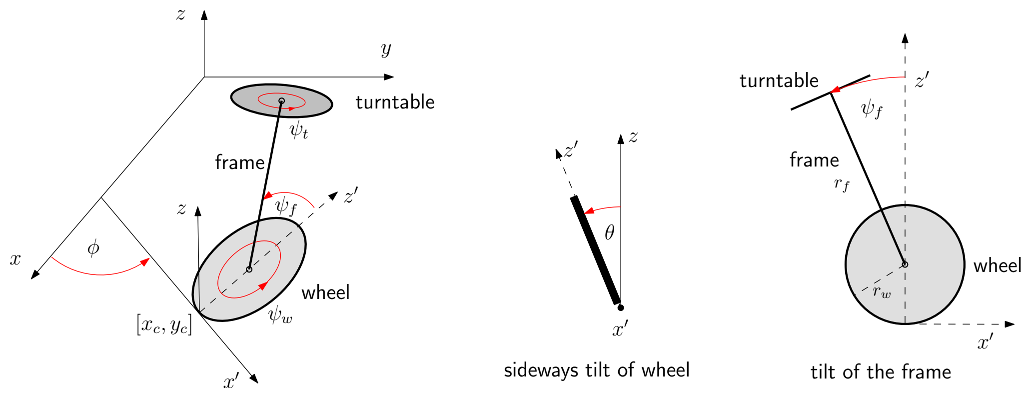


Figure 5.5: Unicycle.

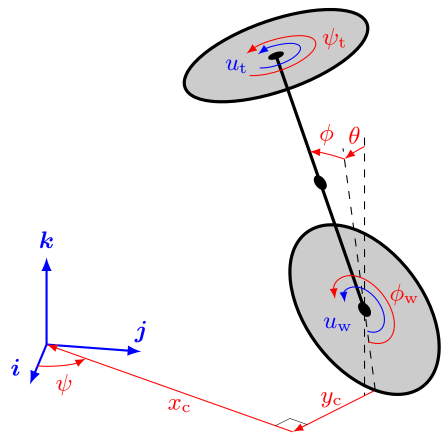


Figure 5.6: The robotic unicycle with state variables: pitch angle ϕ, roll angle θ, yaw angle ψ, wheel angle ϕ_{w}, turntable angle ψ_{t}, the associated angular velocities and the location of the global origin $(x_{\mathrm{c}}, y_{\mathrm{c}})$. The actions are the wheel motor torque u_{w} and the turntable motor torque u_{t}. The global coordinate system is defined by the vectors $\boldsymbol{i}, \boldsymbol{j}$ and $\boldsymbol{k}$.

where

$$\begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} u - b\dot{x}_1 - \frac{1}{2}(m_2 + 2m_3)l_2\dot{\theta}_2^2 \sin\theta_2 - \frac{1}{2}m_3 l_3 \dot{\theta}_3^2 \sin\theta_3 \\ (\frac{1}{2}m_2 + m_3)l_2 g \sin\theta_2 - \frac{1}{2}m_3 l_2 l_3 \dot{\theta}_3^2 \sin(\theta_2 - \theta_3) \\ \frac{1}{2}m_3 l_3 [g \sin\theta_3 + l_2 \dot{\theta}_2^2 \sin(\theta_2 - \theta_3)] \end{bmatrix} .$$

This linear equation system can be solved for $\ddot{x}_1, \ddot{\theta}_2, \ddot{\theta}_3$ and used for numerical simulation.

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.

Constant	Description	Value	Units
m_w	Wheel mass	1.0	kg
r_w	Wheel radius	0.225	m
A_w	Moment of inertia of wheel around $\boldsymbol{i}_\mathrm{w}$	0.0242	$\mathrm{kg\,m^2}$
C_w	Moment of inertia of wheel around $\boldsymbol{k}_\mathrm{w}$	0.0484	$\mathrm{kg\,m^2}$
m_f	Frame mass	23.5	kg
r_f	Frame centre of mass to wheel	0.54	m
A_f	Moment of inertia of frame around $\boldsymbol{i}_\mathrm{f}$	0.4248	$\mathrm{kg\,m^2}$
B_f	Moment of inertia of frame around $\boldsymbol{j}_\mathrm{f}$	0.4608	$\mathrm{kg\,m^2}$
C_f	Moment of inertia of frame around $\boldsymbol{k}_\mathrm{f}$	0.8292	$\mathrm{kg\,m^2}$
m_t	Turntable mass	10.0	kg
r_t	Frame centre of mass to turntable	0.27	m
A_t	Moment of inertia of turntable around $\boldsymbol{i}_\mathrm{t}$	1.3	$\mathrm{kg\,m^2}$
C_t	Moment of inertia of turntable around $\boldsymbol{k}_\mathrm{t}$	0.2	$\mathrm{kg\,m^2}$
g	Gravitational acceleration	9.81	$\mathrm{m\,s^{-2}}$

Table 5.2: Physical constants used for the simulated robotic unicycle. The coordinate systems defined by the $\boldsymbol{i}, \boldsymbol{j}$ and $\boldsymbol{k}$ 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 $\boldsymbol{i}, \boldsymbol{j}$ and $\boldsymbol{k}$. The spatial position of the unicycle is fully defined by the pitch angle ϕ, roll angle θ, yaw angle ψ, wheel angle ϕ_w, turntable angle ψ_t and location of the global origin with respect to the body-centred coordinate system $(x_\mathrm{c}, y_\mathrm{c})$. We chose the state vector to be $\boldsymbol{x} = [\phi, \dot{\phi}, \theta, \dot{\theta}, \psi, \dot{\psi}, \dot{\phi}_\mathrm{w}, \dot{\psi}_\mathrm{t}, x_\mathrm{c}, y_\mathrm{c}]^\top \in \mathbb{R}^{10}$ where we exclude ϕ_w and ψ_t since they clearly have no effect on the dynamics. The action vector $\boldsymbol{u}$ is made up of a wheel motor torque u_w and a turntable motor torque u_t.

Let us start with the coordinates $(x_\mathrm{c}, y_\mathrm{c})$. 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

$$\dot{x}_\mathrm{c} = r_\mathrm{w}\dot{\phi}_\mathrm{w}\cos\psi \tag{5.9}$$

$$\dot{y}_\mathrm{c} = r_\mathrm{w}\dot{\phi}_\mathrm{w}\sin\psi \tag{5.10}$$

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 $\boldsymbol{i}_{\mathrm{w}}, \boldsymbol{j}_{\mathrm{w}}$ and $\boldsymbol{k}_{\mathrm{w}}$ as shown in Figure 5.7(a). We begin by noting that the angular velocity of the wheel coordinate system is $\Omega_{\mathrm{w}} = \dot{\psi}\boldsymbol{k} + \dot{\theta}\boldsymbol{j}_{\mathrm{w}}$. Now noting that the angular velocity of the wheel only differs in the $\boldsymbol{k}_{\mathrm{w}}$ direction and assuming no slip between wheel and floor we have expressions for the velocity and angular velocity of the wheel

$$\begin{aligned}\boldsymbol{v}_{\mathrm{w}} &= -(\boldsymbol{\omega}_{\mathrm{w}} \times r_{\mathrm{w}}\boldsymbol{i}_{\mathrm{w}})\\ \boldsymbol{\omega}_{\mathrm{w}} &= \Omega_{\mathrm{w}} + \dot{\phi}_{\mathrm{w}}\boldsymbol{k}_{\mathrm{w}}\end{aligned}$$

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

$$\begin{aligned}\dot{\boldsymbol{v}}_{\mathrm{w}} &= \frac{\partial \boldsymbol{v}_{\mathrm{w}}}{\partial t} + (\Omega_{\mathrm{w}} \times \boldsymbol{v}_{\mathrm{w}})\\ \dot{\boldsymbol{h}}_{\mathrm{w}} &= A_{\mathrm{w}}\frac{\partial \omega_{\mathrm{w}}[1]}{\partial t}\boldsymbol{i}_{\mathrm{w}} + A_{\mathrm{w}}\frac{\partial \omega_{\mathrm{w}}[2]}{\partial t}\boldsymbol{j}_{\mathrm{w}} + C_{\mathrm{w}}\frac{\partial \omega_{\mathrm{w}}[3]}{\partial t}\boldsymbol{k}_{\mathrm{w}} + (\Omega_{\mathrm{w}} \times \boldsymbol{h}_{\mathrm{w}})\end{aligned}$$

where angular momentum in the wheel frame of reference is $\boldsymbol{h}_{\mathrm{w}} = [A_{\mathrm{w}}; A_{\mathrm{w}}; C_{\mathrm{w}}] \circ \boldsymbol{\omega}_{\mathrm{w}}$. 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 $\boldsymbol{k}_{\mathrm{w}}$ 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

$$m_{\mathrm{w}}\dot{\boldsymbol{v}}_{\mathrm{w}} = R + F_{\mathrm{w}} + W_{\mathrm{w}} \tag{5.11}$$

$$\dot{\boldsymbol{h}}_{\mathrm{w}} = (r_{\mathrm{w}}\boldsymbol{i}_{\mathrm{w}} \times R) + Q_{\mathrm{w}} + T \tag{5.12}$$

5.6.3 Frame FBD

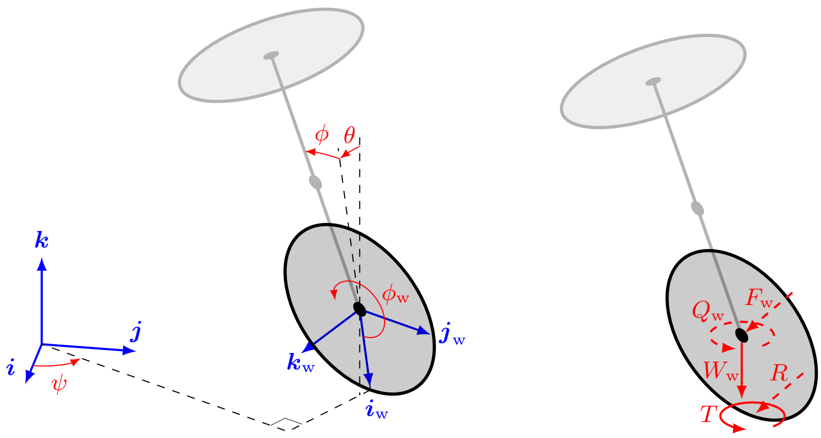
The frame coordinate system is defined by the orthonormal vectors $\boldsymbol{i}_{\mathrm{f}}, \boldsymbol{j}_{\mathrm{f}}$ and $\boldsymbol{k}_{\mathrm{f}} = \boldsymbol{k}_{\mathrm{w}}$ as shown in Figure 5.7(b). In this case, the angular velocity of the frame $\boldsymbol{\omega}_{\mathrm{f}}$ is given by the angular velocity of the wheel plus an additional spin about the wheel axis and the velocity of the frame $\boldsymbol{v}_{\mathrm{f}}$ is given by the velocity of the wheel plus an additional rotation about the wheel centre

$$\begin{aligned}\boldsymbol{v}_{\mathrm{f}} &= \boldsymbol{v}_{\mathrm{w}} - (\boldsymbol{\omega}_{\mathrm{f}} \times r_{\mathrm{f}}\boldsymbol{i}_{\mathrm{f}})\\ \boldsymbol{\omega}_{\mathrm{f}} &= \Omega_{\mathrm{w}} + \dot{\phi}\boldsymbol{k}_{\mathrm{f}}\end{aligned}$$

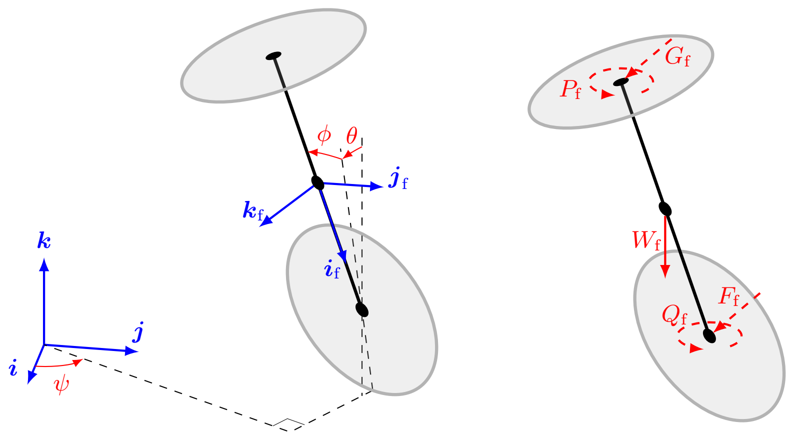
As before, we can now derive the acceleration of the frame $\dot{\boldsymbol{v}}_{\mathrm{f}}$ and the rate of change of angular momentum $\dot{\boldsymbol{h}}_{\mathrm{f}}$ as

$$\begin{aligned}\dot{\boldsymbol{v}}_{\mathrm{f}} &= \frac{\partial \boldsymbol{v}_{\mathrm{f}}}{\partial t} + (\Omega_{\mathrm{f}} \times \boldsymbol{v}_{\mathrm{f}})\\ \dot{\boldsymbol{h}}_{\mathrm{f}} &= A_{\mathrm{f}}\frac{\partial \omega_{\mathrm{f}}[1]}{\partial t}\boldsymbol{i}_{\mathrm{f}} + B_{\mathrm{f}}\frac{\partial \omega_{\mathrm{f}}[2]}{\partial t}\boldsymbol{j}_{\mathrm{f}} + C_{\mathrm{f}}\frac{\partial \omega_{\mathrm{f}}[3]}{\partial t}\boldsymbol{k}_{\mathrm{f}} + (\Omega_{\mathrm{f}} \times \boldsymbol{h}_{\mathrm{f}})\end{aligned}$$

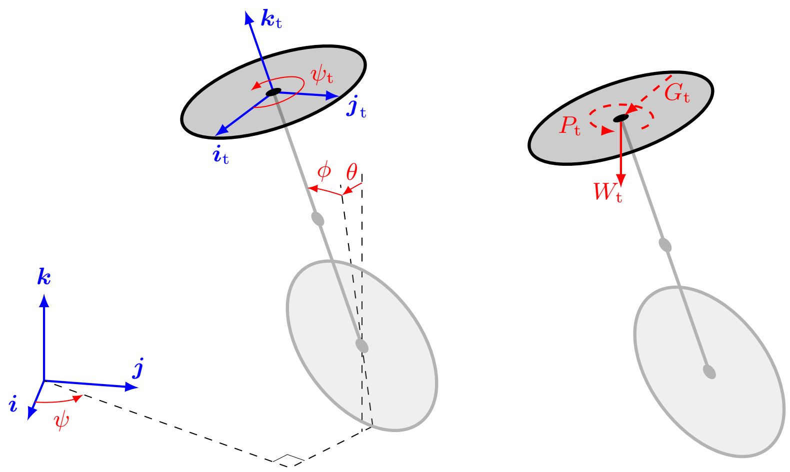
where angular momentum of the frame in this frame of reference is $\boldsymbol{h}_{\mathrm{f}} = [A_{\mathrm{f}}; B_{\mathrm{f}}; C_{\mathrm{f}}] \circ \boldsymbol{\omega}_{\mathrm{f}}$. The forces and moments acting on the frame are shown on the right in Figure 5.7(b). They consist of the known



(a) The wheel as a free body diagram and coordinate system defined by $\boldsymbol{i}_{\mathrm{w}}, \boldsymbol{j}_{\mathrm{w}}$ and $\boldsymbol{k}_{\mathrm{w}}$



(b) The frame as a free body diagram and coordinate system defined by $\boldsymbol{i}_{\mathrm{f}}, \boldsymbol{j}_{\mathrm{f}}$ and $\boldsymbol{k}_{\mathrm{f}}$



(c) The turntable as a free body diagram and coordinate system defined by $\boldsymbol{i}_{\mathrm{t}}, \boldsymbol{j}_{\mathrm{t}}$ and $\boldsymbol{k}_{\mathrm{t}}$

Figure 5.7: Free body diagrams of the wheel, frame and turntable of the unicycle. The model has the angular state variables pitch ϕ, roll θ, yaw ψ, wheel angle ϕ_{w} and turntable angle ψ_{t}. The vectors $\boldsymbol{i}, \boldsymbol{j}$ and $\boldsymbol{k}$ 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 $F_\mathrm{f} = -F_\mathrm{w}$, wheel axle torque $Q_\mathrm{f} = -Q_\mathrm{w}$, turntable axle force G_f & turntable axle torque P_f. But again we know that the dimension of P_f acting along $\boldsymbol{i}_\mathrm{f}$ 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

$$m_\mathrm{f}\dot{\boldsymbol{v}}_\mathrm{f} = F_\mathrm{f} + G_\mathrm{f} + W_\mathrm{f} \tag{5.13}$$

$$\dot{\boldsymbol{h}}_\mathrm{f} = (r_\mathrm{f}\boldsymbol{i}_\mathrm{f} \times F_\mathrm{f}) - (r_\mathrm{t}\boldsymbol{i}_\mathrm{f} \times G_\mathrm{f}) + Q_\mathrm{f} + P_\mathrm{f} \tag{5.14}$$

5.6.4 Turntable FBD

Finally, the turntable coordinate system is defined by the orthonormal vectors $\boldsymbol{i}_\mathrm{t} = \boldsymbol{k}_\mathrm{f}, \boldsymbol{j}_\mathrm{t} = \boldsymbol{j}_\mathrm{f}$ and $\boldsymbol{k}_\mathrm{t} = -\boldsymbol{i}_\mathrm{f}$ as shown in Figure 5.7(c). The velocity of the turntable centre $\boldsymbol{v}_\mathrm{t}$ is equal to the velocity of the wheel plus an additiona lterm caused by rotation about the wheel centre, while the angular velocity $\boldsymbol{\omega}_\mathrm{t}$ differs from $\Omega_\mathrm{t} = \Omega_\mathrm{f}$ only along $\boldsymbol{k}_\mathrm{t}$

$$\begin{aligned}\boldsymbol{v}_\mathrm{t} &= \boldsymbol{v}_\mathrm{w} + (\Omega_\mathrm{t} \times r\boldsymbol{k}_\mathrm{t})\\ \boldsymbol{\omega}_\mathrm{t} &= \Omega_\mathrm{t} + \dot{\psi}_\mathrm{t}\boldsymbol{k}_\mathrm{t}\end{aligned}$$

Again, we derive the acceleration of the frame $\dot{\boldsymbol{v}}_\mathrm{t}$ and the rate of change of angular momentum $\dot{\boldsymbol{h}}_\mathrm{t}$ as

$$\begin{aligned}\dot{\boldsymbol{v}}_\mathrm{t} &= \frac{\partial \boldsymbol{v}_\mathrm{f}}{\partial t} + (\Omega_\mathrm{f} \times \boldsymbol{v}_\mathrm{f})\\ \dot{\boldsymbol{h}}_\mathrm{t} &= A_\mathrm{t}\frac{\partial \omega_\mathrm{t}[1]}{\partial t}\boldsymbol{i}_\mathrm{t} + A_\mathrm{t}\frac{\partial \omega_\mathrm{t}[2]}{\partial t}\boldsymbol{j}_\mathrm{t} + C_\mathrm{t}\frac{\partial \omega_\mathrm{t}[3]}{\partial t}\boldsymbol{k}_\mathrm{t} + (\Omega_\mathrm{t} \times \boldsymbol{h}_\mathrm{t})\end{aligned}$$

where $\boldsymbol{h}_\mathrm{t} = [A_\mathrm{t}; A_\mathrm{t}; C_\mathrm{t}] \circ \boldsymbol{\omega}_\mathrm{t}$. The forces and moments acting on the turntable lead to the last of our equations

$$m_\mathrm{t}\dot{\boldsymbol{v}}_\mathrm{t} = G_\mathrm{t} + W_\mathrm{t} \tag{5.15}$$

$$\dot{\boldsymbol{h}}_\mathrm{t} = P_\mathrm{t} \tag{5.16}$$

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 unkown torques Q and P. The first can be obtained from Equation (5.16) and noting that the component of P_f about $\boldsymbol{k}_\mathrm{t}$ is the reaction to the motor torque u_t

$$\dot{\boldsymbol{h}}_\mathrm{t}^\top \boldsymbol{k}_\mathrm{t} = u_\mathrm{t} \tag{5.17}$$

The second can be obtained by first making use of the relationships $G_\mathrm{f} = -G_\mathrm{t}$ & $P_\mathrm{f} = -P_\mathrm{t}$ and then rearranging Equation (5.13), Equation (5.15) and Equation (5.16) to get

$$\begin{aligned}F_\mathrm{f} &= m_\mathrm{t}\dot{\boldsymbol{v}}_\mathrm{t} + m_\mathrm{f}\dot{\boldsymbol{v}}_\mathrm{f} - W_\mathrm{t} - W_\mathrm{f}\\ G_\mathrm{f} &= W_\mathrm{t} - m_\mathrm{t}\dot{\boldsymbol{v}}_\mathrm{t}\\ P_\mathrm{f} &= -\dot{\boldsymbol{h}}_\mathrm{t}\end{aligned}$$

Substituting these into Equation (5.14) and noting that $Q_\mathrm{w} = -Q_\mathrm{f}$ gives us

$$Q_\mathrm{w} = -\dot{\boldsymbol{h}}_\mathrm{f} - \dot{\boldsymbol{h}}_\mathrm{t} - \Big(r_\mathrm{f}\boldsymbol{i}_\mathrm{f} \times (W_\mathrm{f} + W_\mathrm{t} - m_\mathrm{f}\dot{\boldsymbol{v}}_\mathrm{f} - m_\mathrm{t}\dot{\boldsymbol{v}}_\mathrm{t})\Big) - \Big(r_\mathrm{t}\boldsymbol{i}_\mathrm{f} \times (W_\mathrm{t} - m_\mathrm{t}\dot{\boldsymbol{v}}_\mathrm{t})\Big)$$

Once again, the component of Q_w about the wheel axis is equal to the torque provided by the wheel motor u_w, therefore $Q_\mathrm{w}^\top \boldsymbol{k}_\mathrm{w} = u_\mathrm{w}$ and we have our second expression

$$-\bigg(\dot{\boldsymbol{h}}_\mathrm{f} + \dot{\boldsymbol{h}}_\mathrm{t} + \Big(r_\mathrm{f}\boldsymbol{i}_\mathrm{f} \times (W_\mathrm{f} + W_\mathrm{t} - m_\mathrm{f}\dot{\boldsymbol{v}}_\mathrm{f} - m_\mathrm{t}\dot{\boldsymbol{v}}_\mathrm{t})\Big) + \Big(r_\mathrm{t}\boldsymbol{i}_\mathrm{f} \times (W_\mathrm{t} - m_\mathrm{t}\dot{\boldsymbol{v}}_\mathrm{t})\Big)\bigg)^\top \boldsymbol{k}_\mathrm{w} = u_\mathrm{w} \tag{5.18}$$

Finally, Equation (5.11) can be combined with our expression for $F_\mathrm{f} = -F_\mathrm{w}$ to find the reaction force at the base

$$R = m_\mathrm{w}\dot{\boldsymbol{v}}_\mathrm{w} + m_\mathrm{f}\dot{\boldsymbol{v}}_\mathrm{f} + m_\mathrm{t}\dot{\boldsymbol{v}}_\mathrm{t} - W_\mathrm{w} - W_\mathrm{f} - W_\mathrm{t}$$

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

$$\begin{aligned}\dot{\boldsymbol{h}}_\mathrm{w} = &\Big(r_\mathrm{w}\boldsymbol{i}_\mathrm{w} \times (m_\mathrm{w}\dot{\boldsymbol{v}}_\mathrm{w} + m_\mathrm{f}\dot{\boldsymbol{v}}_\mathrm{f} + m_\mathrm{t}\dot{\boldsymbol{v}}_\mathrm{t} - W_\mathrm{w} - W_\mathrm{f} - W_\mathrm{t})\Big) \\ &- \bigg(\dot{\boldsymbol{h}}_\mathrm{f} + \dot{\boldsymbol{h}}_\mathrm{t} + \Big(r_\mathrm{f}\boldsymbol{i}_\mathrm{f} \times (W_\mathrm{f} + W_\mathrm{t} - m_\mathrm{f}\dot{\boldsymbol{v}}_\mathrm{f} - m_\mathrm{t}\dot{\boldsymbol{v}}_\mathrm{t})\Big) + \Big(r_\mathrm{t}\boldsymbol{i}_\mathrm{f} \times (W_\mathrm{t} - m_\mathrm{t}\dot{\boldsymbol{v}}_\mathrm{t})\Big)\bigg) + T\end{aligned} \tag{5.19}$$

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: $\phi, \dot{\phi}, \theta, \dot{\theta}, \dot{\psi}, \dot{\phi}_\mathrm{w}, \dot{\psi}_\mathrm{t}$, the torque actions: $u_\mathrm{w}, u_\mathrm{t}$ and the accelerations: $\ddot{\phi}, \ddot{\theta}, \ddot{\psi}, \ddot{\phi}_\mathrm{w}, \ddot{\psi}_\mathrm{t}$. 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.

Chapter 6

Testing and Debugging

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

```
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 inv(s)*(covariance of the input and the
              predicted control) wrt the mean of the input distribution
      (viii) 'dCds' - derivative of inv(s)*(covariance of the input and the
              predicted control) wrt the variance of the input distribution
      (ix)   'dCdp' - derivative of inv(s)*(covariance of the input and the
              predicted control) wrt the controller parameters
policy    policy structure
  .fcn    function handle to policy
  .<>     other fields that are passed on to the policy
m         mean of the input distribution
s         covariance of the input distribution
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
```

6.2 Gradient Checks for the Cost Function

`lossT` checks gradients of the (immediate) cost functions, which are specific to each scenario.

6.2.1 Interface

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

Input arguments:

```
deriv     desired derivative. options:
       (i)   'dMdm' - derivative of the mean of the predicted cost
              wrt the mean of the input distribution
       (ii)  'dMds' - derivative of the mean of the predicted cost
              wrt the variance of the input distribution
       (iii) 'dSdm' - derivative of the variance of the predicted cost
              wrt the mean of the input distribution
       (iv)  'dSds' - derivative of the variance of the predicted cost
              wrt the variance of the input distribution
       (v)   'dCdm' - derivative of inv(s)*(covariance of the input and the
              predicted cost) wrt the mean of the input distribution
       (vi)  'dCds' - derivative of inv(s)*(covariance of the input and the
              predicted cost) wrt the variance of the input distribution
cost      cost structure
  .fcn    function handle to cost
  .<>     other fields that are passed on to the cost
m         mean of the input distribution
s         covariance of the input distribution
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
```

6.3 Gradient Checks for the GP Prediction Function

gpT checks gradients of the functions that implement the GP prediction at an uncertain test input, i.e., all gp* functions. These functions can be found in <pilco_root>/gp/.

6.3.1 Interface

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

Input arguments:

```
deriv    desired derivative. options:
     (i)    'dMdm' - derivative of the mean of the GP prediction
             wrt the mean of the input distribution
     (ii)   'dMds' - derivative of the mean of the GP prediction
             wrt the variance of the input distribution
     (iii)  'dMdp' - derivative of the mean of the GP prediction
             wrt the GP parameters
     (iv)   'dSdm' - derivative of the variance of the GP prediction
             wrt the mean of the input distribution
     (v)    'dSds' - derivative of the variance of the GP prediction
             wrt the variance of the input distribution
     (vi)   'dSdp' - derivative of the variance of the GP prediction
             wrt the GP parameters
     (vii)  'dVdm' - derivative of inv(s)*(covariance of the input and the
             GP prediction) wrt the mean of the input distribution
     (viii) 'dVds' - derivative of inv(s)*(covariance of the input and the
             GP prediction) wrt the variance of the input distribution
     (ix)   'dVdp' - derivative of inv(s)*(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:

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

6.4 Gradient Checks for the State Propagation Function

`propagateT` checks gradients of the function `propagated`, which implements state propagation $p(\boldsymbol{x}_t) \to p(\boldsymbol{x}_{t+1})$. This function can be found in `<pilco_root>/base/`.

6.4.1 Interface

```
[dd dy dh] = propagateT(deriv, plant, dynmodel, policy, m, s, delta)
```

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             predicted mean at time t+1                       [E x 1]
Snext             predicted covariance at time t+1                 [E x E]
dMdm              output mean wrt input mean                       [E x D]
dMds              output mean wrt input covariance matrix        [E  x D*D]
dSdm              output covariance matrix wrt input mean       [E*E x  D ]
dSds              output cov wrt input cov                      [E*E x D*D]
dMdp              output mean wrt policy parameters                [E x P]
dSdp              output covariance matrix wrt policy parameters [E*E x  P ]

where P is the number of policy parameters.
```

6.5 Gradient Checks for Policy Evaluation

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

6.5.1 Interface

```
[d dy dh] = valueT(p, delta, m, s, dynmodel, policy, plant, cost, H)
```

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

Last modified: 2014-02-03

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

```
% 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


  % 2. Make many rollouts to test the controller quality
  if plotting.verbosity > 1
    Nroll = 10; lat = cell(1,Nroll); rC = cell(1,Nroll);
    for i=1:Nroll
      [~,~,rC{i},lat{i}] = rollout(gaussian(mu0, S0), policy, HH, plant, cost, 0);
    end

    if ~ishandle(3); figure(3); else set(0,'CurrentFigure',3); end
    hold on; for ii=1:Nroll; plot(rC{ii},'r'); end      % plot the real losses

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

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

  % 3. Plot training predictions and rollouts for comparison
  if ~ishandle(3); figure(3); else set(0,'CurrentFigure',3); end
  hold on; plot(realCost{J+j},'g'); drawnow;         % the trial added to the dataset
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]
```


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
angi = plant.angi;  % angular indices
poli = plant.poli;  % policy indices
dyni = plant.dyni;  % dynamics-model indices

D0 = length(m);                                          % size of the input mean
D1 = D0 + 2*length(angi);          % length after mapping all angles to sin/cos
D2 = D1 + length(policy.maxU);              % length after computing ctrl 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);        % the latent state
q = S(j,i)*C; S(j,k) = q; S(k,j) = q';

mm = zeros(D1,1); ss = zeros(D1); mm(i) = M(i);        % state which policy uses
ss(i,i) = S(i,i) + diag(exp(2*[dynmodel.hyp.on]));     % has observ. noise added
[mm(k), ss(k,k), C] = gTrig(mm(i), ss(i,i), angi);     % gTrig the noisy state
q = ss(j,i)*C; ss(j,k) = q; ss(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,i)*C; S(j,k) = q; S(k,j) = q';

% 3) Compute distribution of the change in state ------------------------------
i = [dyni D1+1:D2]; j = 1:D2; k = D2+1:D3;
[M(k), S(k,k), C] = dynmodel.fcn(dynmodel, M(i), S(i,i));
S(k,k) = S(k,k) + diag(exp(2*[dynmodel.hyp.n]));          % add process noise
q = S(j,i)*C; S(j,k) = q; S(k,j) = q';

```

```
% 4) Select distribution of the next state ----------------------------------
Mnext = M(D2+1:D3); Snext = S(D2+1:D3,D2+1:D3); 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
verb        verbosity level
```

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 ]
```


Last modification: 2013-12-15

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

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

Code

```
clear simulate;                                    % clear the simulate function
if isfield(plant,'augment'), augi = plant.augi;             % sort out indices!
else plant.augment = @(x)[]; augi = []; end
if isfield(plant,'subplant'), subi = plant.subi;
else plant.subplant = @(x,y)[]; subi = []; end
if nargin < 6; verb = 0; 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);      % initialisations
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);
L = zeros(1, H); next = zeros(1,length(simi));

for i = 1:H % ---------------------------------------------------------- run ROLLOUT
  s = x(i,dyno)'; sa = gTrig(s, zeros(length(s)), angi); s = [s; sa];
  x(i,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);            % run dynamics
  next(subi) = plant.subplant(state, u(i,:)); % additional discrete time dynamics
  next(augi) = plant.augment(next);                        % augment next state

  % 3. Test constraints and stop rollout if violated -----------------------
  if isfield(plant,'constraint') && plant.constraint(next)
    H = i-1; if verb; fprintf('state constraints violated...\n'); end; break;
  end

  % 4. Constraints not violated so accept next state
  state = next;                              % update the noise-free, hidden state
  x(i+1,simi) = state(simi) + randn(size(simi))*chol(plant.noise);   % observed
  x(i+1,augi) = plant.augment(x(i+1,:));                  % augment noisy state

  % 5. Compute Cost ---------------------------------------------------------
```

```
  if nargout > 2;  L(i) = cost.fcn(cost,state(dyno)',zeros(length(dyno))); end
end
if verb; fprintf('\nTrial lasted %i steps\n',floor(H));  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

Last modification: 2014-02-03

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(:,dyni) x(:,end-Du+1:end)];     % use dyni and ctrl
dynmodel.targets = y(:,dyno);

dynmodel = dynmodel.train(dynmodel, trainOpt);  % train dynamics GP

% display some hyperparameters
h = dynmodel.hyp;                                % display hyperparameters
disptable([exp([h.n]); exp([h.on]); sqrt(dynmodel.noise); exp([h.s])./sqrt(dynmodel.↩
    noise)],...
    varNames,'process noises: |observ. noises: |total noises: |SNRs: ')

% signal-to-noise ratios (values > 500 can cause numerical problems)
```

7.1.5 value.m

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

```
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
```


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

```
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                                  % for all time steps in horizon
    [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                                % for all time steps in horizon
    [m, S, dmdm0, dSdm0, dmdS0, dSdS0, dmdp, dSdp] = ...
      plant.prop(m, S, plant, dynmodel, policy); % get next state

    % check whether we ran into severe numerical problems
    if any(isnan(S(:))) || any(~isreal(S(:))) || min(eig(S)) > 1e10; break; end

    dmdp = dmdm0*dm0dp + dmdS0*dS0dp + dmdp;
    dSdp = dSdm0*dm0dp + dSdS0*dS0dp + 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, dCdm, dMds, dSds, dCds, dMdp, dSdp, dCdp] ...
         = 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 x 1]
s          covariance of input distribution                    [D x D]
```

Output arguments:

```
M          control mean                                        [E   x   1]
S          control covariance                                  [E   x   E]
C          inv(s)*cov(x,u)                                     [D   x   E]
```

```
dMdm      deriv. of expected control wrt input mean              [E   x   D]
dSdm      deriv. of control covariance wrt input mean            [E*E x   D]
dCdm      deriv. of C wrt input mean                             [D*E x   D]
dMds      deriv. of expected control wrt input covariance        [E   x D*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 expected 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 is the total number of policy parameters
```


Last modified: 2012-07-03

High-Level Steps

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

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

Code

```
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
  q = S(i,i)*Q; S(i,j) = q; S(j,i) = q';  % compute joint covariance S=cov(x,v)
  [M, S, R] = sat(M, S, j, maxU);         % compute squashed control signal u
  C = [eye(D) Q]*R;                       % inv(s)*cov(x,u)
else             % with derivatives
  Mdm = zeros(F,D); Sdm = zeros(F*F,D); Mdm(1:D,1:D) = eye(D);
  Mds = zeros(F,D*D); Sds = kron(Mdm,Mdm);

  X = reshape(1:F*F,[F F]); XT = X';                        % vectorized indices
  I=0*X;I(j,j)=1; jj=X(I==1)'; I=0*X;I(i,j)=1;ij=X(I==1)'; ji=XT(I==1)';

  % 1. Unsquashed controller ----------------------------------------------
```

```
    [M(j), S(j,j), Q, Mdm(j,:), Sdm(jj,:), dQdm, Mds(j,:), ...
      Sds(jj,:), dQds, Mdp, Sdp, dQdp] = con(policy, m, s);
    q = S(i,i)*Q; S(i,j) = q; S(j,i) = q';  % compute joint covariance S=cov(x,v)

    % update the derivatives
    SS = kron(eye(E),S(i,i)); QQ = kron(Q',eye(D));
    Sdm(ij,:) = SS*dQdm;      Sdm(ji,:) = Sdm(ij,:);
    Sds(ij,:) = SS*dQds + QQ; Sds(ji,:) = Sds(ij,:);

    % 2. Apply Saturation ---------------------------------------------------
    [M, S, R, MdM, SdM, RdM, MdS, SdS, RdS] = sat(M, S, j, maxU);

    % apply chain-rule to compute derivatives after concatenation
    dMdm = MdM*Mdm + MdS*Sdm; dMds = MdM*Mds + MdS*Sds;
    dSdm = SdM*Mdm + SdS*Sdm; dSds = SdM*Mds + SdS*Sds;
    dRdm = RdM*Mdm + RdS*Sdm; dRds = RdM*Mds + RdS*Sds;

    dMdp = MdM(:,j)*Mdp + MdS(:,jj)*Sdp;
    dSdp = SdM(:,j)*Mdp + SdS(:,jj)*Sdp;
    dRdp = RdM(:,j)*Mdp + RdS(:,jj)*Sdp;

    C = [eye(D) Q]*R; % inv(s)*cov(x,u)
    % update the derivatives
    RR = kron(R(j,:)',eye(D)); QQ = kron(eye(E),[eye(D) Q]);
    dCdm = QQ*dRdm + RR*dQdm;
    dCds = QQ*dRds + RR*dQds;
    dCdp = QQ*dRdp + RR*dQdp;
end
```

7.2.2 congp.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] ...
         = congp(policy, m, s)
```

Input arguments:

```
policy        policy (struct)
  .p          parameters that are modified during training
    .hyp      GP-log hyperparameters (Ph = (d+2)*D)             [ Ph      ]
    .inputs   policy pseudo inputs                              [ N  x  d ]
    .targets  policy pseudo targets                             [ N  x  D ]
m             mean of state distribution                        [ d  x  1 ]
s             covariance matrix of state distribution           [ d  x  d ]
```

Output arguments:

```
M             mean of the predicted control                     [ D  x  1 ]
S             covariance of predicted control                   [ D  x  D ]
C             inv(s)*covariance between input and control       [ d  x  D ]
dMdm          deriv. of mean control wrt mean of state          [ D  x  d ]
dSdm          deriv. of control variance wrt mean of state      [D*D x  d ]
dCdm          deriv. of covariance wrt mean of state            [d*D x  d ]
dMds          deriv. of mean control wrt variance               [ D  x d*d]
dSds          deriv. of control variance wrt variance           [D*D x d*d]
dCds          deriv. of covariance wrt variance                 [d*D x d*d]
dMdp          deriv. of mean control wrt GP hyper-parameters    [ D  x  P ]
dSdp          deriv. of control variance wrt GP hyper-parameters [D*D x  P ]
dCdp          deriv. of covariance wrt GP hyper-parameters      [d*D x  P ]
```

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

Last modified: 2014-02-03

High-Level Steps

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

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

Code

```
D = length(m);

% 1. Extract policy parameters
policy.hyp = policy.p.hyp;
if isfield(policy.p,'inputs'); policy.inputs = policy.p.inputs; end
policy.targets = policy.p.targets;

% fix policy signal and the noise variance
% (avoids some potential numerical problems)
[policy.hyp.s] = deal(log(1));                    % set signal variance to 1
[policy.hyp.n] = deal(log(0.01));                % set noise standard dev to 0.01
[policy.hyp.m] = deal(zeros(D,1));
[policy.hyp.b] = deal(0);
```

```
policy = gpPreComp(policy);

% 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, policy.inputs, m, s);
else                                        % else compute derivatives too
  [M, S, C, dMdm, dSdm, dCdm, dMds, dSds, dCds, dMdp, dSdp, dCdp] ...
                                   = gp2d(policy, policy.inputs, m, s);

  % 3. Set derivatives of non-free parameters to zero: signal and noise variance
  [n,d] = size(policy.inputs); d2 = d + 2; dimU = size(policy.targets,2);
  sidx = bsxfun(@plus,(d+1:d2)',(0:dimU-1)*d2);
  dMdp(:,sidx(:)) = 0; dSdp(:,sidx(:)) = 0; dCdp(:,sidx(:)) = 0;
  if ~isfield(policy.p,'inputs'); i = [1:d2*dimU d2*dimU+n*d+(1:n*dimU)];
    dMdp = dMdp(:,i); dSdp = dSdp(:,i); dCdp = dCdp(:,i);
  end

end
```

```
function [M, S, V, dMdm, dSdm, dVdm, dMds, dSds, dVds, ...
                                   dMdp, dSdp, dVdp] = conlin(policy, m, s)
% Linear controller with input size D and output size E
%
% policy      policy structure
%   .p         parameters which are modified by training
%     .w        linear weights                                  [ E  x  D ]
%     .b        biases                                          [ E       ]
% m           mean of state distribution                        [ D       ]
% s           covariance matrix of state distribution           [ D  x  D ]
%
% M           mean of the action                                [ E       ]
% S           variance of action                                [ E  x  E ]
% C           inv(s) times input output covariance              [ D  x  E ]
% dMdm        mean output by mean input derivative              [ E  x  D ]
% dSdm        covariance by mean input derivative               [E*E x  D ]
% dCdm        input output covariance by mean input derivative  [D*E x  D ]
% dMds        mean by covariance derivative                     [ E  x D*D]
% dSds        covariance by covariance derivative               [E*E x D*D]
% dCds        C by covariance derivative                        [D*E x D*D]
% dMdp        mean by parameters derivative                     [ E  x  P ]
% dSdp        covariance by parameters derivative               [E*E x  P ]
% dCdp        C by parameters derivative                        [D*E x  P ]
%
% where P = (D+1)*E
%
% Copyright (C) by Carl Edward Rasmussen and Marc Deisenroth, 2012-06-25
% Edited by Joe Hall 2012-07-03

w = policy.p.w; b = policy.p.b; [E D] = size(w);% dim of control and state
M = w*m + b;                                                        % mean
S = w*s*w'; S = (S+S')/2;                                     % covariance
V = w';                                    % inv(s)*input-output covariance

if nargout > 3
  dMdm = w;            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; % symmetrise
  X=reshape(1:E*E,[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*D 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;                        % symmetrise
  dVdp = [zeros(D*E,E) wTdw];
end
```

7.3 GP Directory

```
function [gp, nlml] = train(gp, iter)

% train a GP model with SE covariance function (ARD). First the hypers are
% trained using full GPs. Then, if gp.induce exists, indicating sparse
% approximation, if enough training exmples are present, train the inducing
% inputs. If no inducing inputs are present, then initialize these to be a
% random subset of the training cases.
%
% gp                 struct gaussian process dynamics model
%   hyp      1xE     struct of hyperparameters (ignored on the input)
%     m      Dx1     mean function coefficients
%     b      1x1     mean function bias
%     l      Dx1     ARD log lenghtscale parameters
%     s      1x1     log of signal std dev
%     n      1x1     log of noise std dev
%   inputs   nxD     training inputs
%   targets  nxE     training targets
%   induce   Mxdxe   [optional] inducing inputs
%   mean     1x1     [optional] switch for training mean (1) or keeping it
%                     fixed at its set value (0). Defaults to 1.
% iter       1x2     [optional] number of training iterations
% nlml       1xE     negative log marginal likelihood for each GP (incl curb)
%
%
% Last modification: 2014-02-03

if nargin < 2, iter = [-500 -1000]; end       % default training iterations
if isfield(gp,'trainMean'); mS = gp.trainMean; else mS = 0; end
[N, D] = size(gp.inputs); E = size(gp.targets,2);       % get variable sizes
nlml = zeros(1,E);

curb.snr = 300; curb.ls = 100; curb.std = std(gp.inputs);    % set hyp curb
[gp.hyp(1:E).l] = deal(log(std(gp.inputs)')); t = log(std(gp.targets)); % initialize ↩
    hyp
s = num2cell(t); [gp.hyp.s] = deal(s{:});
s = num2cell(t-log(10)); [gp.hyp.n] = deal(s{:});
if mS; [gp.hyp(1:E).m] = deal(zeros(D,1)); [gp.hyp.b] = deal(0); end

if isfield(gp.hyp,'on'); gp.hyp = rmfield(gp.hyp,'on'); end

for i = 1:E                                    % train each GP separately
  [gp.hyp(i), v] = minimize(gp.hyp(i), @hypCurb, iter(1), gp.inputs, ...
```

```
    gp.targets(:,i), mS, curb);
  nlml(i) = v(end);
end

if isfield(gp,'induce')              % are we using a sparse approximation?
  [M, d, e] = size(gp.induce);
  if M < N;                % only call FITC if we have enough training points

    if d == 0                             % we don't have inducing inputs yet?
      gp.induce = zeros(M,D,e);                               % allocate space
      for i = 1:e
        j = randperm(N); gp.induce(:,:,i) = gp.inputs(j(1:M),:); % random subset
      end
    end

    [gp.induce, nlml2] = minimize(gp.induce, 'fitc', iter(end), gp);
    fprintf('GP NLML, full: %e, sparse: %e, diff: %e\n', ...
      sum(nlml), nlml2(end), nlml2(end)-sum(nlml));
  else
    fprintf('GP NLML: %e\n', sum(nlml));
  end
end

% We must now split the single learnt noise level into observation and
% process noise terms. The AR GP trained here assumes all the noise is
% observation noise. We keep a small amount of process noise for numerical
% stability.
gp.noise = exp(2*[gp.hyp.n]);                      % the total amount of noise
for i=1:E;
  gp.hyp(i).n = gp.hyp(i).s - log(curb.snr);                 % process noise
  gp.hyp(i).on = log(max(gp.noise(i) - exp(2*gp.hyp(i).n),1e-9))/2; % observation noise
end

gp = gpPreComp(gp);         % make the precomputations needed for prediction
```

```
function [f, df] = hypCurb(hyp, x, y, mS, curb)

% wrapper for gpr training, penalising large SNR, extreme length-scales,
% and linear mean weights larger than 1. The penalty thresholds can be set
% in the curb struct input argument. The penalty should keep:
%                            SNR < curb.snr
%        curb.std/curb.ls <  L  < curb.std*curb.ls
%                          -1 <  m  < 1
%
% Carl Edward Rasmussen & Andrew McHutchon, 2013-11-07

if nargin < 4, curb.snr = 500; curb.ls = 100; curb.std = 1; end % set default

p = 30;                                                      % penalty power

[f, df] = gp(hyp, x, y, mS);                                   % first, call gp

f = f + sum(((hyp.l - log(curb.std'))./log(curb.ls)).^p);    % length-scales
df.l = df.l + p*(hyp.l - log(curb.std')).^(p-1)/log(curb.ls)^p;

f = f + sum(((hyp.s - hyp.n)/log(curb.snr)).^p);     % signal to noise ratio
```

```
df.s = df.s + p*(hyp.s - hyp.n).^(p-1)/log(curb.snr)^p;
df.n = df.n - p*sum((hyp.s - hyp.n).^(p-1)/log(curb.snr)^p);

if isfield(hyp,'m') && mS;                    % keep linear weights between -/+1
    f = f + sum(hyp.m.^p);
    df.m = df.m + p*hyp.m.^(p-1);
end
```

7.3.1 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, gp)
```

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.)
gp              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:

```
nlml             negative log-marginal likelihood
dnlml            derivative of negative log-marginal likelihood wrt
                 inducing inputs
```

Adapted from Ed Snelson's SPGP code.

Last modified: 2013-07-05

High-Level Steps

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

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

Code

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

[N, D] = size(gp.inputs); E = length(gp.hyp);
[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(gp.hyp(j).l);                                    % length-scales
  c = gp.hyp(j).s;                                      % log signal std dev
  sig = exp(2.*gp.hyp(j).n);                               % noise variance

  xb = bsxfun(@rdivide,u,b');                  % divide inducing by lengthscales
  x = bsxfun(@rdivide,gp.inputs,b');            % divide inputs by length-scales
  y = gp.targets(:,j);                                    % training targets
  if isfield(gp.hyp,'b'); y = y - gp.hyp(j).b; end                % remove mean
  if isfield(gp.hyp,'m'); y = y - gp.inputs*gp.hyp(j).m; end

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

  try
    L = chol(Kmm)';
  catch
    nlml = Inf; dnlml = zeros(size(params)); return;
  end
  V = L\Kmn;                                            % inv(sqrt(Kmm))*Kmn

  Gamma = 1 + (exp(2*c)-sum(V.^2)')/sig;       % Gamma = diag(Knn-Qnn)/sig + I


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

  nlml = nlml + sum(log(diag(Am))) + (N-M)/2*log(sig) + sum(log(Gamma))/2 ...
    + (y'*y - beta'*beta)/2/sig + 0.5*N*log(2*pi);

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

    At = L*Am; iAt = At\eye(M);                  % chol(sig*B) [thesis, p. 40]
    iA = iAt'*iAt;                                            % inv(sig*B)

    % C = iAt*Kmn2;
    % iK = diag(1./(sig*Gamma)) - C'*C;
    % B = At*At'./sig; % B matrix [thesis, p. 40]

    iAmV = Am\V;                                                % inv(Am)*V
    B1 = At'\(iAmV);
    b1 = At'\beta;                                              % b1 = B1*y
```

```

    iLV = L'\V;                                          % inv(Kmm)*Kmn*inv(sqrt(Gamma))
    iL = L\eye(M);
    iKmm = iL'*iL;

    mu = ((Am'\beta)'*V)';
    bs = y.*(beta'*iAmV)'/sig - sum(iAmV.*iAmV)'/2 - (y.^2+mu.^2)/2/sig + 0.5;
    TT = iLV*(bsxfun(@times,iLV',bs));
    Kmn = bsxfun(@rdivide,Kmn,sqrt(Gamma)');                          % overwrite Kmn

    for i = 1:D                                       % derivatives wrt inducing inputs
      dsq_mm = bsxfun(@minus,xb(:,i),xb(:,i)').*Kmm;
      dsq_mn = bsxfun(@minus,-xb(:,i),-x(:,i)').*Kmn;
      dGamma = -2/sig*dsq_mn.*iLV;

      dfxb(:,i) = -b1.*(dsq_mn*(y-mu)/sig + dsq_mm*b1) + dGamma*bs ...
        + sum((iKmm - iA*sig).*dsq_mm,2) - 2/sig*sum(dsq_mm.*TT,2);
      dsq_mn = dsq_mn.*B1;                                         % overwrite dsq_mn
      dfxb(:,i) = dfxb(:,i) + sum(dsq_mn,2);
      dfxb(:,i) = dfxb(:,i)/b(i);
    end

    dnlml(:,:,j) = dfxb;
  end
end
if 1 == uE; dnlml = sum(dnlml,3); end  % combine derivatives if sharing inducing
```

7.3.2 gp0.m

Summary: Compute joint predictions for multiple GPs with uncertain inputs. If gpmodel.nigp exists, individial 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 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 ]
```

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

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

Code

```
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

% 1) if necessary: re-compute cashed variables
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);

  for i=1:E                                            % compute K and inv(K)
    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
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(1:D,i))); % inverse length-scales
  in = inp*iL;
  B = iL*s*iL+eye(D);

  t = in/B;
  l = exp(-sum(in.*t,2)/2); lb = l.*beta(:,i);
```

```
  tiL = t*iL;
  c = exp(2*X(D+1,i))/sqrt(det(B));

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

% 3) ompute predictive covariance, non-central moments
for i=1:E
  ii = bsxfun(@rdivide,inp,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,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(iK(:,:,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

% 4) centralize moments
S = S - M*M';
```

7.3.3 gp1.m

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 gpmodel.nigp exists, individual noise contributions are added.

```
function [M, S, V] = gp1d(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, 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

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

Code

```
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
pinput = gpmodel.induce;                                   % all pseudo inputs

% 1) If necessary: re-compute cached variables
if numel(X) ~= numel(oldX) || isempty(iK) || isempty(iK2) || ... % if necessary
              sum(any(X ~= oldX)) || numel(iK2) ~=E*np^2 || numel(iK) ~= n*np*E
  oldX = X;                                      % compute K, inv(K), inv(K2)
  iK = zeros(np,n,E); iK2 = zeros(np,np,E); beta = zeros(np,E);

  for i=1:E
    pinp = bsxfun(@rdivide,pinput(:,:,min(i,pE)),exp(X(1:D,i)'));
    inp = bsxfun(@rdivide,input,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)';
    V = L\Kmn;                                     % inv(sqrt(Kmm))*Kmn
    if isfield(gpmodel,'nigp')
      G = exp(2*X(D+1,i))-sum(V.^2)+gpmodel.nigp(:,i)';
    else
      G = exp(2*X(D+1,i))-sum(V.^2);
    end
    G = sqrt(1+G/exp(2*X(D+2,i)));
    V = bsxfun(@rdivide,V,G);
```

```
    Am = chol(exp(2*X(D+2,i))*eye(np) + V*V')';
    At = L*Am;                                  % chol(sig*B) [thesis, p. 40]
    iAt = At\eye(np);
% The following is not an inverse matrix, but we'll treat it as such: multiply
% the targets from right and the cross-covariances left to get predictive mean.
    iK(:,:,i) = ((Am\(bsxfun(@rdivide,V,G)))'*iAt)';
    beta(:,i) = iK(:,:,i)*targets(:,i);
    iB = iAt'*iAt.*exp(2*X(D+2,i));              % inv(B), [Ed's thesis, p. 40]
    iK2(:,:,i) = Kmm\eye(np) - iB; % covariance matrix for predictive variances
  end
end

k = zeros(np,E); M = zeros(E,1); V = zeros(D,E); S = zeros(E);       % allocate
inp = zeros(np,D,E);

% 2) Compute predicted mean and inv(s) times input-output covariance
for i=1:E
  inp(:,:,i) = bsxfun(@minus,pinput(:,:,min(i,pE)),m');

  L = diag(exp(-X(1:D,i)));
  in = inp(:,:,i)*L;
  B = L*s*L+eye(D);

  t = in/B;
  l = exp(-sum(in.*t,2)/2); lb = l.*beta(:,i);
  tL = t*L;
  c = exp(2*X(D+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(D+1,i)-sum(in.*in,2)/2;
end

% 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).*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

% 4) Centralize moments
S = S - M*M';
```

7.3.4 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, individial noise contributions are added.

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

Input arguments:

```
gpmodel    GP model struct
hyp(i)     struct array of GP hyper-parameters                [ 1  x  E ]
    .l     log lengthscales                                   [ D  x  1 ]
    .s     log signal standard deviation                      [ 1  x  1 ]
    .n     log noise standard deviation                       [ 1  x  1 ]
    .m     linear weights for the GP mean                     [ D  x  1 ]
    .b     biases for the GP mean                             [ 1  x  1 ]
  inputs   training inputs                                    [ n  x  D ]
  targets  training targets                                   [ 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 ]
```


Last modified: 2043-03-03

High-Level Steps

1. Compute predicted mean and inv(s) times input-output covariance
2. Compute predictive covariance matrix, non-central moments
3. Centralize moments

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

Code

```
[n, D, pE] = size(x); E = size(gpmodel.beta,2);
h = gpmodel.hyp; beta = gpmodel.beta;
if ~isfield(h,'m'); [h.m] = deal(zeros(D,1)); end
if ~isfield(h,'b'); [h.b] = deal(0); end

M = zeros(E,1); V = zeros(D,E); S = zeros(E);
k = zeros(n,E); a = zeros(D,E); M1 = zeros(E,1);

inp = bsxfun(@minus,x,m');                 % x - m, either n-by-D or n-by-D-by-E


% 2) Compute predicted mean and inv(s) times input-output covariance
for i=1:E     % compute predicted mean and inv(s) times input-output covariance
  il = diag(exp(-h(i).l));            % Lambda^-1/2
  in = inp(:,:,min(i,pE))*il;         % (X - m)*Lambda^-1/2
  B = il*s*il+eye(D);                 % Lambda^-1/2 * S * *Lambda^-1/2 + I

  t = in/B;                           % in.*t = (X-m) (S+L)^-1 (X-m)
  l = exp(-sum(in.*t,2)/2); lb = l.*beta(:,i);
  tL = t*il;
  c = exp(2*h(i).s)/sqrt(det(B));     % = sf2/sqrt(det(S*iL + I))

  M1(i) = sum(lb)*c; M(i) = M1(i) + h(i).m'*m + h(i).b;         % predicted mean
  V(:,i) = tL'*lb*c + h(i).m;              % inv(s) times input-output covariance
  k(:,i) = 2*h(i).s-sum(in.*in,2)/2;
  liBl = il*(B\il); xm = x(:,:,min(i,pE))'*lb*c;
  a(:,i) = diag(exp(2*h(i).l))*liBl*m*M1(i) + s*liBl*xm;
end

% 3) Compute predictive covariance, non-central moments
iL = exp(-2*[h.l]);inpiL = bsxfun(@times,inp,permute(iL,[3,1,2])); % N-by-D-by-E
for i=1:E                       % compute predictive covariance, non-central moments
  for j=1:i
    R = s*diag(iL(:,i)+iL(:,j))+eye(D); t = 1/sqrt(det(R));
    L = exp(bsxfun(@plus,k(:,i),k(:,j)')+maha(inpiL(:,:,i),-inpiL(:,:,j),R\s/2));
    S(i,j) = beta(:,i)'*L*beta(:,j)*t;                   % variance of the mean
    S(i,j) = S(i,j) + h(i).m'*(a(:,j) - m*M1(j)) + h(j).m'*(a(:,i) - m*M1(i));
    S(j,i) = S(i,j);
  end

  S(i,i) = S(i,i) + 1e-6;          % add small jitter for numerical reasons

end

% 4) Centralize moments
S = S - M1*M1' + [h.m]'*s*[h.m];                          % centralize moments
```

Bibliography

[1] James A. Bagnell and Jeff G. Schneider. Autonomous Helicopter Control using Reinforcement Learning Policy Search Methods. In *Proceedings of the International Conference on Robotics and Automation*, 2001.

[2] Marc P. Deisenroth. *Efficient Reinforcement Learning using Gaussian Processes*, volume 9 of *Karlsruhe Series on Intelligent Sensor-Actuator-Systems*. KIT Scientific Publishing, November 2010. ISBN 978-3-86644-569-7.

[3] Marc P. Deisenroth and Carl E. Rasmussen. PILCO: A Model-Based and Data-Efficient Approach to Policy Search. In *Proceedings of the International Conference on Machine Learning*, pages 465–472, New York, NY, USA, June 2011. ACM.

[4] Marc P. Deisenroth, Carl E. Rasmussen, and Dieter Fox. Learning to Control a Low-Cost Manipulator using Data-Efficient Reinforcement Learning. In *Proceedings of the International Conference on Robotics: Science and Systems*, Los Angeles, CA, USA, June 2011.

[5] David Forster. Robotic Unicycle. Report, Department of Engineering, University of Cambridge, UK, 2009.

[6] Joaquin Quiñonero-Candela, Agathe Girard, Jan Larsen, and Carl E. Rasmussen. Propagation of Uncertainty in Bayesian Kernel Models—Application to Multiple-Step Ahead Forecasting. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 2, pages 701–704, April 2003.

[7] Carl E. Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. Adaptive Computation and Machine Learning. The MIT Press, Cambridge, MA, USA, 2006.

[8] Edward Snelson and Zoubin Ghahramani. Sparse Gaussian Processes using Pseudo-inputs. In Y. Weiss, B. Schölkopf, and J. C. Platt, editors, *Advances in Neural Information Processing Systems 18*, pages 1257–1264. The MIT Press, Cambridge, MA, USA, 2006.

[9] Mark W. Spong and Daniel J. Block. The Pendubot: A Mechatronic System for Control Research and Education. In *Proceedings of the Conference on Decision and Control*, pages 555–557, 1995.

[10] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.