# Guide for MC-PILCO Software Package (version 1.0)

**Fabio Amadio**[1,*], **Alberto Dalla Libera**[1,+], **and Diego Romeres**[2,]

[1] *Department of Information Engineering, University of Padova* (Via Gradenigo 6/B, 35131 Padova, Italy)
[*] amadiofa@dei.unipd.it
[+] dallaliber@dei.unipd.it
[2] *Mitsubishi Electric Research Laboratories* (201 Broadway, 02139, Cambridge, Massachusetts)
romeres@merl.com

## Monte Carlo-Probabilistic Inference for Learning and COntrol (MC-PILCO)[1]

This package implements a Model-based Reinforcement Learning algorithm called Monte Carlo Probabilistic Inference for Learning COntrol (MC-PILCO), for modeling and control of dynamical system. The algorithm relies on Gaussian Processes (GPs) to model the system dynamics and on a Monte Carlo approach to estimate the policy gradient during optimization.

The algorithm is also extended to work for Partially Measurable Systems and takes the name of MC-PILCO-4PMS.

Please see *Amadio et al.*[1] for a detailed description of the algorithms. The code is implemented in Python 3 and reproduces the simulation examples in *Amadio et al.*[1], namely, the ablation studies and solution of a cart-pole system (available both as a Python simulated system and as an environment in the physic engine MuJoCo) and a trajectory controller for a UR5 (simulated in MuJoCo). Statistical results can be reproduced via Monte Carlo simulations.

The user has the possibility to add his own python system or MuJoCo Environment and solve it with MC-PILCO or MC-PILCO-4PMS.

## Installation

- Download the code from Github https://github.com/merlresearch/MC-PILCO
- Create a Python 3 environment with the following packages: PyTorch[2] (version 1.4 or superior), NumPy[3], Matplotlib[4], Pickle[5], Argparse[6].
- To test the code on MuJoCo[7] environments, also MuJoCo_py[8] and Gym[9] libraries are needed.
- Or you can create a conda environment with the above dependencies by executing:

```
conda env create -file environment.yaml
```

## Library Structure

```
/mc_pilco
├── /gpr_lib/ ..... Gaussian Process Regression library
├── /model_learning
│   └── Model_learning.py ..... Classes for model learning and one-step-ahead predictions
├── /policy_learning
│   ├── Cost_function.py ..... Collection of cost function classes
│   ├── MC_PILCO.py ..... Main class implementing MC-PILCO algorithm
│   ├── MC_PILCO_mujoco_envs.py ..... Main class implementing MC-PILCO in MuJoCo
│   └── Policy.py ..... Collection of policy classes
├── /results_tmp/ ..... Data logging folder
├── /simulation_class
│   ├── model.py ..... Classes for interaction with ODE-simulated systems
│   ├── model_mujoco.py ..... Classes for interaction with MuJoCo environments
│   └── ode_systems.py ..... Collection of functions describing ODEs of simulated systems
├── /envs
│   ├── /assets
│   │   ├── cartpole_swingup.xml ..... Definition of the cart-pole system for MuJoCo
│   │   ├── UR5.xml ..... Definition of the UR5-robot system for MuJoCo
│   │   ├── /mesh/
│   │   └── /textures/
│   ├── cartpole_swingup.py ..... Definition of the cart-pole gym environment
│   ├── target_q_trajectory.csv ..... Target joint trajectory for the UR5 experiment
│   └── ur5.py ..... Definition of the UR5-robot gym environment
├── log_plot_cartpole.py ..... Plot results of cart-pole experiment
├── log_plot_cartpole_mujoco.py ..... Plot results of MuJoCo cart-pole experiment
├── log_plot_ur5.py ..... Plot results of MuJoCo UR5-robot experiment
├── repeat_test.py ..... Repeat a test several times
├── apply_mcpilco_policy.py ..... Apply MC-PILCO policy on cart-pole
├── apply_mcpilco_policy_on_model.py ..... Apply MC-PILCO policy on model
├── apply_mcpilco4pms_policy.py ..... Apply MC-PILCO4PMS policy on cart-pole
├── apply_mcpilco4pms_policy_on_model.py ..... Apply MC-PILCO4PMS policy on model
├── test_mcpilco_cartpole.py ..... Test MC-PILCO on cart-pole
├── test_mcpilco_cartpole_mujoco.py ..... Test MC-PILCO on MuJoCo cart-pole
├── test_mcpilco_cartpole_multi_init.py ..... Bimodal initial condition experiment
├── test_mcpilco4pms_cartpole.py ..... Test MC-PILCO4PMS on cart-pole
├── test_mcpilco_cartpole_rbf_ker.py ..... Test MC-PILCO on cart-pole using RBF kernels
└── test_mcpilco_ur5_mujoco.py ..... Test MC-PILCO on MuJoCo UR5-robot
```

## Scenario 1: Cart-pole swing-up using GP model with SE+P$^{(2)}$ kernel

**`test_mcpilco_cartpole.py`**

*This first example is meant to describe the structure and the main ingredients of a possible launch file to learn how to control a dynamical system with the algorithm MC-PILCO.*

The example in the file `test_mcpilco_cartpole.py` describes how to apply the algorithm MC-PILCO to a cart-pole system simulated with Ordinary Differential Equations. The differential equations are written as Python functions, see `simulation_class\ode_systems.py`. The cart-pole dynamics are modeled as GPs equipped with a squared exponential (SE) plus a second order polynomial kernel P$^{(2)}$, denoted as SE+P$^{(2)}$ in *Amadio et al.*[?]. The user can try the experiment by running `python test_mcpilco_cartpole.py`. We will refer to the cart position with $p$, to the pole angle with $\theta$ and to the applied force with $u$. This scenario is simulated through class `simulation_class.model.Model()` and cart-pole ODE dynamics function is defined inside `simulation_class.ode_systems.py`.

*(A)* Import libraries, classes and methods.

```
import torch
import numpy as np
import model_learning.Model_learning as ML
import policy_learning.Policy as Policy
import policy_learning.MC_PILCO as MC_PILCO
import policy_learning.Cost_function as Cost_function
import simulation_class.ode_systems as f_ode
import gpr_lib.Likelihood.Gaussian_likelihood as Likelihood
import gpr_lib.Utils.Parameters_covariance_functions as cov_func
import matplotlib.pyplot as plt
import pickle as pkl
import argparse
```

*(B)* Load the parameter `seed` from command line to set the random seed of PyTorch and NumPy (by default `seed` is set to 1). Also some other PyTorch settings are defined. It is necessary to choose if the computations must be performed weather on CPU or GPUs, by selecting accordingly the PyTorch device. CPU is selected by default, but the user can choose to use GPUs by uncommenting the associated line. It is also possible to select the number of threads used for intraop parallelism with parameter `num_threads` (by default `num_threads` is set to 1). Finally, here are also defined two flags that the user can change to modify the settings of the test. `fl_SOD_GP` set to `True` enables the use of Subset Of Data (SOD) reduction techniques for GPs, `fl_reinforce_init_dist` can be set either to `Gaussian` or `Uniform` to define the initial distribution for particle states.

```
# Load random seed from command line
p = argparse.ArgumentParser('test cartpole')
p.add_argument('-seed', type=int, default=1, help='seed')
locals().update(vars(p.parse_known_args()[0]))
# Set the seed
torch.manual_seed(seed)
np.random.seed(seed)
# Default data type
dtype=torch.float64
# Set the device
device=torch.device('cpu')
# device=torch.device('cuda:0')
# Set number of computational threads
num_threads = 1
torch.set_num_threads(num_threads)
# Flag to select if to use or not a SOD approximation in the GPs
fl_SOD_GP = False
# Flag to select initial distribution of the particles ['Gaussian' or 'Uniform']
fl_reinforce_init_dist = 'Gaussian'
```

*(C)* Set environment parameters.

- `num_trials`: Number of trials to be performed by MC-PILCO .
- `T_sampling`: Sampling time $T_s$ (in seconds).
- `T_control`: Length of the episode (in seconds).

- `state_dim`: Dimension of the state. In the cart-pole case it is 4, being $\mathbf{x} = \left[p, \dot{p}, \theta, \dot{\theta}\right]$.
- `input_dim`: Dimension of the input. In the cart-pole case it is 1, the force applied to the cart.
- `num_gp`: Number of GPs. By default it is `state_dim/2` because MC-PILCO uses the *speed-integration* model.
- `gp_input_dim`: Dimension of the GP input. In the cart-pole case it is 6, being $\tilde{\mathbf{x}} = \left[p, \dot{p}, \dot{\theta}, sin(\theta), cos(\theta), u\right]$.
- `ode_fun`: Name of the considered ODE function, defined in `simulation_class/model.py`.
- `u_max`: Maximum (absolute) input value.
- `std_list`: List containing standard deviations of the measurement noise for each state component.

```
print('---- Set environment parameters ----')
num_trials = 5
T_sampling = 0.05
T_exploration = 3.
T_control = 3.
state_dim = 4
input_dim = 1
num_gp = int(state_dim/2)
gp_input_dim = 6
ode_fun = f_ode.cartpole
u_max = 10.
std_noise = 10**(-2)
std_list = [std_noise, std_noise, std_noise, std_noise]
```

*(D)* Initialize GP models.

Notice that multiple model classes have been defined in `model_learning/Model_learning.py`, corresponding to different GPR and kernel structures. These classes are convenient wrappers around the Gaussian Process Regression library `gpr_lib`. The user can write his/her own model class. The model class used in the experiment is instantiated in the object `f_model_learning`. In this case we considered the class `Speed_Model_learning_RBF_MPK_angle_state()`, that combines a *squared-exponential* and *polynomial* kernel, adopting a *speed-integration* model structure. The dictionary `model_learnin_par` contains all the main information and parameters to learn the model. Some of the main keys in this dictionary are:

- `angle_indeces`: Indeces of the state components that are angles.
- `not_angle_indeces`: Indeces of the state components that are not angles.
- `T_sampling`: Sampling time $T_s$, needed for the *speed-integration* model structure.
- `vel_indeces`: Indeces of the state components that are velocities.
- `not_vel_indeces`: Indeces of the state components that are not velocities.
- `init_dict_list`: List of dictionaries that characterize the kernels combined in the model.

For this model class two dictionaries are defined and listed within `init_dict_list`, one for each kernel `init_dict_RBF` for the *squared-exponential* kernel and `init_dict_MPK` for the *polynomial* kernel; the class will then sum the kernels up. The key names within each dictionary should be self-explanatory, some examples are:

- `active_dims`, to define the kernel in only a subset of the variables defined in the GP input vector.
- `poly_deg`, defines the degree of the polynomial kernel in `init_dict_MPK` (in this experiment it is set to 2).
- `approximation_mode`, defines the reduction techniques adopted, for example if its value is set to `SOD`, the Subset Of Data (SOD) reduction technique is implemented.
- `approximation_dict`, contains the parameters for SOD. `SOD_threshold_mode` can be set to `absolute` or `relative`. In `absolute` mode, the threshold values for each GP are fixed and must be passed inside a list in `SOD_threshold`. In `relative` mode, the thresholds are variable and their values are equal to the noise standard deviations of each GP scaled by the factor passed inside `SOD_threshold`. `flg_SOD_permutation` enables (or not) the permutation of data.

```
print('\n---- Set model learning parameters ----')
f_model_learning = ML.Speed_Model_learning_RBF_MPK_angle_state
print(f_model_learning)
model_learning_par = {}
model_learning_par['num_gp'] = num_gp
model_learning_par['angle_indeces'] = [2]
model_learning_par['not_angle_indeces'] = [0,1,3]
model_learning_par['T_sampling'] = T_sampling
model_learning_par['vel_indeces'] = [1,3]
model_learning_par['not_vel_indeces'] = [0,2]
model_learning_par['device'] = device
model_learning_par['dtype'] = dtype
if fl_SOD_GP:
    model_learning_par['approximation_mode'] = 'SOD'
    model_learning_par['approximation_dict'] = {'SOD_threshold_mode':'relative', 'SOD_threshold':.5, '
    flg_SOD_permutation':False} # Set SoD threshold
# RBF initial par
init_dict_RBF = {}
init_dict_RBF['active_dims'] = np.arange(0,gp_input_dim)
init_dict_RBF['lengthscales_init'] = np.ones(init_dict_RBF['active_dims'].size)
init_dict_RBF['flg_train_lengthscales'] = True
init_dict_RBF['lambda_init'] = np.ones(1)
init_dict_RBF['flg_train_lambda'] = False
init_dict_RBF['sigma_n_init'] = 1*np.ones(1)
init_dict_RBF['sigma_n_num'] = None
init_dict_RBF['flg_train_sigma_n'] = True
init_dict_RBF['dtype'] = dtype
init_dict_RBF['device'] = device
# MPK initial par
init_dict_MPK = {}
init_dict_MPK['active_dims'] = np.arange(0,gp_input_dim)
init_dict_MPK['poly_deg'] = 2
init_dict_MPK['Sigma_pos_par_init_list'] = [np.ones(gp_input_dim+1)]+[np.ones((deg+1)*(gp_input_dim))
    for deg in range(1,init_dict_MPK['poly_deg'])]
init_dict_MPK['flg_train_Sigma_pos_par_list'] = [True]*init_dict_MPK['poly_deg']
init_dict_MPK['dtype'] = dtype
init_dict_MPK['device'] = device
model_learning_par['init_dict_list'] = [[init_dict_RBF, init_dict_MPK]]*state_dim
```

*(E)* Set the exploration policy

(all the implemented policies are defined in classes inside `policy_learning/Policy.py`).

The class `Random_exploration()` applies a random control action (sampled from a uniform distribution between -`u_max` and `u_max`) at each time step. Parameters are included inside the dictionary `rand_exploration_par`.

```
print('\n---- Set exploration policy ----')
f_rand_exploration_policy = Policy.Random_exploration
rand_exploration_policy_par = {}
rand_exploration_policy_par['state_dim'] = state_dim
rand_exploration_policy_par['input_dim'] = input_dim
rand_exploration_policy_par['u_max'] = u_max
rand_exploration_policy_par['dtype'] = dtype
rand_exploration_policy_par['device'] = device
```

*(F)* Set the control policy

(all the implemented policies are defined in classes inside `policy_learning/Policy.py`).

A *squashed-RBF-network* policy that takes as input an extended state $\left[p, \dot{p}, \dot{\theta}, sin(\theta), cos(\theta)\right]$ is used in this example. This policy is defined by the `Sum_of_gaussians_with_angles()` class. Parameters are included inside the dictionary `control_policy_par`. Important parameters to be set are:

- `angle_indeces`: Indeces of the state components that are angles.
- `non_angle_indeces`: Indeces of the state that components are not angles.
- `num_basis`: Number of basis functions.
- `centers_init`: Initial centers of the Gaussian basis functions.
- `lengthscales_init`: Initial lengthscales of the Gaussian basis functions.

- `weight_init`: Initial weights of the sum of basis functions.
- `flg_drop`: Enables dropout during policy optimization.
- `flg_squash`: Enables "squashing" inputs between -`u_max` and `u_max`.

```
print('\n---- Set control policy ----')
num_basis = 200
f_control_policy = Policy.Sum_of_gaussians_with_angles
control_policy_par = {}
control_policy_par['state_dim'] = state_dim
control_policy_par['input_dim'] = input_dim
control_policy_par['angle_indices'] = np.array([2])
control_policy_par['non_angle_indices'] = np.array([0,1,3])
control_policy_par['u_max'] = u_max
control_policy_par['num_basis'] = num_basis
control_policy_par['dtype'] = dtype
control_policy_par['device'] = device
angle_centers = np.pi*2*(np.random.rand(num_basis,1)-0.5)
cos_centers = np.cos(angle_centers)
sin_centers = np.sin(angle_centers)
not_angle_centers = np.pi*2*(np.random.rand(num_basis,3)-0.5)
control_policy_par['centers_init'] = np.concatenate([not_angle_centers,cos_centers,sin_centers],1)
control_policy_par['lengthscales_init'] = 1*np.ones(state_dim+1)
control_policy_par['weight_init'] = u_max*(np.random.rand(input_dim,num_basis)-0.5)
control_policy_par['flg_squash'] = True
control_policy_par['flg_drop'] = True
policy_reinit_dict = {}
policy_reinit_dict['lenghtscales_par'] = control_policy_par['lengthscales_init']
policy_reinit_dict['centers_par'] = np.array([np.pi, np.pi, np.pi, 1., 1.])
policy_reinit_dict['weight_par'] = u_max
```

*(G)* Set the cost function
(all the implemented cost functions are defined inside `policy_learning/Cost_function.py`).

The class `Cart_pole_cost()` implements the cost function $1-\exp\left(-\frac{(|\theta|-\pi)^2}{l_\theta^2}-\frac{p^2}{l_p^2}\right)$. Parameters `pos_index` and `angle_index` specify, respectively, the index of the cart position and pole angle in the state vector. In `target_state` it is defined the upward unstable equilibrium angle (in this case $\pi$) and the target position of the cart, respectively in the first and the second element of the vector. In `lengthscales` the user can choose the lengthscales of the cost $[l_\theta, l_p]$.

```
print('\n---- Set cost function ----')
f_cost_function = Cost_function.Cart_pole_cost
cost_function_par = {}
cost_function_par['pos_index'] = 0
cost_function_par['angle_index'] = 2
cost_function_par['target_state'] = torch.tensor([np.pi, 0.], dtype=dtype, device=device)
cost_function_par['lengthscales'] = torch.tensor([3.,1.], dtype=dtype, device=device)
```

(H) Set the policy learning object.
All the functions and associated parameters defined in points (D)-(G) are now combined into a dictionary called `MC_PILCO_init_dict` which defines a "Policy learning" object named `PL_obj` through the class `MC_PILCO.MC_PILCO()`.

```
print('\n---- Init policy learning object ----')
MC_PILCO_init_dict = {}
MC_PILCO_init_dict['T_sampling'] = T_sampling
MC_PILCO_init_dict['state_dim'] = state_dim
MC_PILCO_init_dict['input_dim'] = input_dim
MC_PILCO_init_dict['f_sim'] = ode_fun
MC_PILCO_init_dict['std_meas_noise'] = np.array(std_list)
MC_PILCO_init_dict['f_model_learning'] = f_model_learning
MC_PILCO_init_dict['model_learning_par'] = model_learning_par
MC_PILCO_init_dict['f_rand_exploration_policy'] = f_rand_exploration_policy
MC_PILCO_init_dict['rand_exploration_policy_par'] = rand_exploration_policy_par
MC_PILCO_init_dict['f_control_policy'] = f_control_policy
MC_PILCO_init_dict['control_policy_par'] = control_policy_par
MC_PILCO_init_dict['f_cost_function'] = f_cost_function
MC_PILCO_init_dict['cost_function_par'] = cost_function_par
MC_PILCO_init_dict['log_path'] = 'results_tmp/'+str(seed)
```

```
MC_PILCO_init_dict['dtype'] = dtype
MC_PILCO_init_dict['device'] = device
PL_obj = MC_PILCO.MC_PILCO(**MC_PILCO_init_dict)
```

*(I)* Model optimization settings.
The dictionary `model_optimization_dict` defines the model learning settings. The most important options to be set are:

- `N_epoch`: Number of optimization steps.
- `f_optimizer`: This is the optimizer defined as a string together with its parameters. In this example, `'lambda p : torch.optim.Adam(p, lr=0.01)'` indicates that GP hyperparameters are optimized using Adam with learning rate 0.01
- `criterion`: The metric with which the hyperparameters are optimized. In this example we use `Likelihood.Marginal_log_li` i.e., the maximization of the marginal likelihood.

```
print('\n---- Set MC-PILCO options ----')
# Model optimization options
model_optimization_opt_dict = {}
model_optimization_opt_dict['f_optimizer'] = 'lambda p : torch.optim.Adam(p, lr=0.01)'
model_optimization_opt_dict['criterion'] = Likelihood.Marginal_log_likelihood
model_optimization_opt_dict['N_epoch'] = 1501
model_optimization_opt_dict['N_epoch_print'] = 500
model_optimization_opt_list = [model_optimization_opt_dict]*num_gp
```

*(J)* Policy optimization settings.
The dictionary `policy_optimization_dict` defines the policy optimization settings. The most important options to be set are:

- `num_particles`: Number of simulated particles.
- `opt_setp_lists`: List of maximum policy optimization steps for each trial.
- `lr_lists`: List with the starting learning rates for each trial.
- `p_dropout_lists`: List with the starting dropout probabilities for each trial.
- `p_drop_reduction`: Dropout probability reduction rate.
- `alpha_diff_cost`: Monitoring signal parameter $\alpha_s$.
- `min_diff_cost`: Monitoring signal parameter $\sigma_s$.
- `num_min_diff_cost`: Monitoring signal parameter $n_s$.
- `min_step`: Number of initial steps without updating learning rate and dropout.
- `lr_min`: Minimum allowed learning rate.

```
print('\n---- Set MC-PILCO options ----')
# Policy optimization options
policy_optimization_dict = {}
policy_optimization_dict['num_particles'] = 400
policy_optimization_dict['opt_steps_list'] = [2000, 4000, 4000, 4000, 4000]
policy_optimization_dict['lr_list'] = [0.01, 0.01, 0.01, 0.01, 0.01]
policy_optimization_dict['f_optimizer'] = 'lambda p, lr : torch.optim.Adam(p, lr)'
policy_optimization_dict['num_step_print'] = 100
policy_optimization_dict['p_dropout_list'] = [.25, .25, .25, .25, .25]
policy_optimization_dict['p_drop_reduction'] = 0.25/2
policy_optimization_dict['alpha_diff_cost']=0.99
policy_optimization_dict['min_diff_cost']= 0.08
policy_optimization_dict['num_min_diff_cost']=200
policy_optimization_dict['min_step']=200
policy_optimization_dict['lr_min']=0.0025
policy_optimization_dict['policy_reinit_dict'] = policy_reinit_dict
```

*(K)* Initialization of the policy optimization method and definition of initial state distribution.
The parameters defined in points (I)-(J) are now combined into a dictionary called `reinforce_param_dict`. Inside this dictionary, the following options are set:

- `T_exploration`: Duration in seconds of the initial exploratory policy.
- `T_control`: Duration in seconds of the learned policy at each trial.
- `num_trials`: Number of trials.
- Set the distribution of the initial states in the particles evolution.
  If the initial conditions are a normal distribution the mean and covariance matrix are defined in `initial_state` and `initial_state_variance`, respectively. The covariance matrix is then computed as diag(`initial_state_variance`).
  If the initial conditions are a uniform distribution, the upper and lower bounds are defined with the parameters `init_up_bound` and `init_low_bound`.

```
# Options for method reinforce
reinforce_param_dict = {}
reinforce_param_dict['initial_state'] = np.array([0., 0., 0., 0.])
reinforce_param_dict['initial_state_var'] = np.array([0.0001, 0.0001, 0.0001, 0.0001])
if fl_reinforce_init_dist == 'Gaussian': # Set initial state Gaussian distribution
    reinforce_param_dict['initial_state'] = np.array([0., 0., 0., 0.])
    reinforce_param_dict['initial_state_var'] = np.array([0.0001, 0.0001, 0.0001, 0.0001])
elif fl_reinforce_init_dist == 'Uniform': # Set initial state uniform distribution
    reinforce_param_dict['flg_init_uniform'] = True
    reinforce_param_dict['init_up_bound'] = np.array([0.03, 0.03, 0.03, 0.03])
    reinforce_param_dict['init_low_bound'] = -np.array([0.03, 0.03, 0.03, 0.03])
reinforce_param_dict['T_exploration'] = T_exploration
reinforce_param_dict['T_control'] = T_control
reinforce_param_dict['num_trials'] = num_trials
reinforce_param_dict['model_optimization_opt_list'] = model_optimization_opt_list
reinforce_param_dict['policy_optimization_dict'] = policy_optimization_dict
```

*(L)* Save experiment configuration.

```
print('\n---- Save test configuration ----')
config_log_dict = {}
config_log_dict['MC_PILCO_init_dict'] = MC_PILCO_init_dict
config_log_dict['reinforce_param_dict'] = reinforce_param_dict
pkl.dump(config_log_dict,open('results_tmp/'+str(seed)+'/config_log.pkl','wb'))
```

*(M)* Run the policy learning method.

```
# Start the learning algorithm
PL_obj.reinforce(**reinforce_param_dict)
```

The reader may look at the file presented in Scenario 1 as a general structure to be adapted according to its application.

## Scenario 2: Cart-pole swing-up using GP model with SE kernel

**`test_mcpilco_cartpole_rbf_ker.py`**

The same cart-pole experiment described in Scenario 1, can be tested using SE kernel for GPs by running `python test_mcpilco_cartpole_rbf_ker.py`.

The only difference w.r.t. Scenario 1 is the definition of the model learning parameters at point *(D)*.

- `f_model_learning` is now defined with the class `Speed_Model_learning_RBF_angle_state()`
- `init_dict_list`: is defined with only one dictionary that defines the SE kernel.

```
print('\n---- Set model learning parameters ----')
f_model_learning = ML.Speed_Model_learning_RBF_angle_state
print(f_model_learning)
model_learning_par = {}
model_learning_par['num_gp'] = num_gp
model_learning_par['angle_indeces'] = [2]
```

```
model_learning_par['not_angle_indeces'] = [0,1,3]
model_learning_par['T_sampling'] = T_sampling
model_learning_par['vel_indeces'] = [1,3]
model_learning_par['not_vel_indeces'] = [0,2]
model_learning_par['device'] = device
model_learning_par['dtype'] = dtype
init_dict = {}
# RBF initial par
init_dict['active_dims'] = np.arange(0,gp_input_dim)
init_dict['lengthscales_init'] = np.ones(init_dict['active_dims'].size)
init_dict['flg_train_lengthscales'] = True
init_dict['lambda_init'] = np.ones(1)
init_dict['flg_train_lambda'] = False
init_dict['sigma_n_init'] = 1*np.ones(1)
init_dict['sigma_n_num'] = None
init_dict['flg_train_sigma_n'] = True
init_dict['dtype'] = dtype
init_dict['device'] = device
model_learning_par['init_dict_list'] = [init_dict]*num_gp
```

## Scenario 3: Cart-pole swing-up with bimodal initial state distribution

**`test_mcpilco_cartpole_multi_init.py`**

The same cart-pole experiment described in scenario 1 can be tested when considering the initial distribution of the cart positions generate by a bimodal gaussian distribution by running `python test_mcpilco_cartpole_multi_init.py`.

Starting from the file of Scenario 1, it is possible to define the initial state distribution as a mixture of equally probable Gaussians by changing parameters `initial_state` and `initial_state_var` defined in point *(K)*. In the presented scenario the initial distribution is given by two Gaussians, one centered in $[-1,0,0,0]$ and the other in $[1,0,0,0]$, both with covariance matrix $\text{diag}([10^{-4},10^{-4},10^{-4},10^{-4}])$.

```
reinforce_param_dict['initial_state'] = np.array([[-1., 0., 0., 0.],[1., 0., 0., 0.]])
reinforce_param_dict['initial_state_var'] = np.array([[0.0001, 0.0001, 0.0001, 0.0001],[0.0001, 0.0001,
    0.0001, 0.0001]])
```

## Scenario 4: Cart-pole swing-up with non-measurable velocities (MC-PILCO4PMS)

**`test_mcpilco4pms_cartpole.py`**

The user can try the algorithm MC-PILCO4PMS in the simulation experiment of the cart-pole scenario with non-measurable components of the state by running `python test_mcpilco4pms_cartpole.py`. For further details on MC-PILCO4PMS algorithm we refer the reader to *Amadio et al.*[?]. In this scenario it is assumed to have the possibility to measure only cart position and pole angle, without having direct measurements of velocities.

In order to consider this situation and apply MC-PILCO4PMS, define the policy learning object `PL_obj` with the class `MC_PILCO.MC_PILCO4PMS()` at point *(H)* of Scenario 1. Inside the parameter dictionary `MC_PILCO_init_dict`, it is necessary to define the indices of the state components that correspond to both the positions, `pos_indeces`, and the velocities, `vel_indeces`. Furthermore, in this scenario we need to simulate an online filter that estimates the velocities from the position measurements, in this case represented by numerical differentiation followed by low-pass filtering. The cut-off frequency of the filter is given inside the `filtering_dict` as `{'fc':0.5}`. This scenario is simulated through class `simulation_class.model.PMS_Model()`. The user can test different online state estimators by modifying the method `PMS_Model.rollout()`.

```
MC_PILCO_init_dict['pos_indeces'] = [0,2]
MC_PILCO_init_dict['vel_indeces'] = [1,3]
MC_PILCO_init_dict['filtering_dict'] = {'fc':0.5} #define the cutoff freq
PL_obj = MC_PILCO.MC_PILCO4PMS(**MC_PILCO_init_dict)
```

Additionally, In this experiment we also considered a different exploration policy, `Sum_of_sinusoids()`, that implements a sum of sine waves with random amplitudes and frequencies.

```
print('\n---- Set exploration policy ----')
f_rand_exploration_policy = Policy.Sum_of_sinusoids
```

```
rand_exploration_policy_par = {}
rand_exploration_policy_par['state_dim'] = state_dim
rand_exploration_policy_par['input_dim'] = input_dim
rand_exploration_policy_par['u_max'] = u_max
rand_exploration_policy_par['dtype'] = dtype
rand_exploration_policy_par['device'] = device
rand_exploration_policy_par['num_sin'] = 10
rand_exploration_policy_par['omega_min'] = 0.1*(2*np.pi)
rand_exploration_policy_par['omega_max'] = 2*(2*np.pi)
rand_exploration_policy_par['amplitude_min'] = u_max/10
rand_exploration_policy_par['amplitude_max'] = u_max/10
```

## Scenario 5: MuJoCo cart-pole swing-up

**`test_mcpilco_cartpole_mujoco.py`**

The algorithm MC-PILCO can be applied also to MuJoCo environments. In this Scenario we show how to solve a MuJoCo-simulated cart-pole system by running `python test_mcpilco_cartpole_mujoco.py`. The structure of the file is similar to Scenario 1, with minor modifications needed to work with MuJoCo. The MuJoCo environment defines the state vector as $\mathbf{x} = \left[p, \theta, \dot{p}, \dot{\theta}\right]$, and the upward equilibrium pole angle is assumed to be in 0, so some parameters throughout the file have been change accordingly.

In point *(A)*, it is necessary to register the desired MuJoCo environment name, and define the MuJoCo simulation time step `sim_timestep` (please note that it is different from the sampling time). `sim_timestep` must be the same of the one defined in the `xml` file defining the MuJoCo system (inside `envs/assets/`).

```
from gym.envs.registration import register
register(
    id = 'CartpoleSwingupEnv-v0',
    entry_point = 'envs.cartpole_swingup:CartpoleSwingupEnv',
)
...
sim_timestep = 0.01 # Simulator timestep, it must be the same of envs/assets/cartpole_swingup.xml
env_name = 'CartpoleSwingupEnv-v0'
```

Then, the policy learning object `PL_obj` at point *(H)*, is defined through the class `MC_PILCO_mujoco_envs.MC_PILCO_Mujoco()`, which takes care of the interaction with MuJoCo environments. It is necessary to provide the additional parameter `sim_timestep`, and the environment name through `f_sim` class parameter.

```
MC_PILCO_init_dict['f_sim'] = env_name
...
MC_PILCO_init_dict['sim_timestep'] = sim_timestep
PL_obj = MC_PILCO_mujoco_envs.MC_PILCO_Mujoco(**MC_PILCO_init_dict)
```

## Scenario 6: MuJoCo UR5 robot arm

**`test_mcpilco_ur5_mujoco.py`**

The example in the file `test_mcpilco_ur5_mujoco.py` describes how to apply the algorithm MC-PILCO to a UR5 robotic arm simulated in MuJoCo, with the objective of learning a joint-space controller able to follow a given reference trajectory. The user can try the experiment by running `python test_mcpilco_ur5_mujoco.py`.

*(A)* Import the need libraries, classes and methods.

```
import torch
import numpy as np
import model_learning.Model_learning as ML
import policy_learning.Policy as Policy
import policy_learning.Cost_function as Cost_function
import policy_learning.MC_PILCO_mujoco_envs as MC_PILCO_mujoco_envs
import gpr_lib.Likelihood.Gaussian_likelihood as Likelihood
import gpr_lib.Utils.Parameters_covariance_functions as cov_func
import matplotlib.pyplot as plt
```

```
import pickle as pkl
import argparsefrom gym.envs.registration import register
```

*(B)* Register the MuJoCo environment name. Load the parameter `seed` from command line to set the random seed of PyTorch and NumPy. Define PyTorch settings, as described at point *(B)* of Scenario 1.

```
# Register gym environment
register(
    id = 'UR5_Env-v0',
    entry_point = 'envs.ur5:UR5_Env',
)

# Load random seed from command line
p = argparse.ArgumentParser('test ur5 mujoco')
p.add_argument('-seed', type=int, default=1, help='seed')
locals().update(vars(p.parse_known_args()[0]))

# Set the seed
torch.manual_seed(seed)
np.random.seed(seed)

# Default data type
dtype=torch.float64

# Set the device
device=torch.device('cpu')
# device=torch.device('cuda:0')

# Set number of computational threads
num_threads = 1
torch.set_num_threads(num_threads)
```

*(C)* Set environment parameters. The state of the system is formed by joint angles $\mathbf{q} \in \mathbb{R}^6$ and velocities $\dot{\mathbf{q}} \in \mathbb{R}^6$, namely $\mathbf{x} = [\mathbf{q}, \dot{\mathbf{q}}] \in \mathbb{R}^{12}$. The control actions are the torques applied to each joint $\mathbf{u} \in \mathbb{R}^6$.

```
print('---- Set environment parameters ----')
num_trials = 2
T_sampling = 0.02
T_control = 4.
T_exploration = T_control
state_dim = 12
input_dim = 6
num_gp = 6
gp_input_dim = state_dim + int(state_dim/2) + input_dim
env_name = 'UR5_Env-v0'
sim_timestep = 0.001 # Simulator timestep, it must be the same of envs/assets/UR5.xml
u_max = [1., 1., 1., 1., 1., 1.]
std_noise = 10**(-3)
std_list = std_noise*np.ones(state_dim)
```

*(D)* Initialize GP models. The model class used is `Speed_Model_learning_RBF_MPK_angle_state()`, that combines a *squared-exponential* and *polynomial* kernel, adopting a *speed-integration* model structure. The degree of the polynomial kernel `poly_deg` is set to 1 in this experiment. SOD reduction technique is used to reduce computational time, the threshold values are passed in `absolute` mode.

```
print('\n---- Set model learning parameters ----')
f_model_learning = ML.Speed_Model_learning_RBF_MPK_angle_state
model_learning_par = {}
model_learning_par['num_gp'] = num_gp
model_learning_par['angle_indeces'] = [0,1,2,3,4,5]
model_learning_par['not_angle_indeces'] = [6,7,8,9,10,11]
model_learning_par['T_sampling'] = T_sampling
model_learning_par['vel_indeces'] = [6,7,8,9,10,11]
model_learning_par['not_vel_indeces'] = [0,1,2,3,4,5]
model_learning_par['device'] = device
model_learning_par['dtype'] = dtype
model_learning_par['approximation_mode'] = 'SOD'
model_learning_par['approximation_dict'] = {'SOD_threshold_mode':'absolute', 'SOD_threshold':[.001]*
    num_gp, 'flg_SOD_permutation':False} # Set SoD threshold
# RBF initial par
```

```
init_dict_RBF = {}
init_dict_RBF['active_dims'] = np.arange(0,gp_input_dim)
init_dict_RBF['lengthscales_init'] = np.ones(init_dict_RBF['active_dims'].size)
init_dict_RBF['flg_train_lengthscales'] = True
init_dict_RBF['lambda_init'] = np.ones(1)
init_dict_RBF['flg_train_lambda'] = False
init_dict_RBF['sigma_n_init'] = 1*np.ones(1)
init_dict_RBF['flg_train_sigma_n'] = True
init_dict_RBF['dtype'] = dtype
init_dict_RBF['device'] = device
# MPK initial par
init_dict_MPK = {}
init_dict_MPK['active_dims'] = np.arange(0,gp_input_dim)
init_dict_MPK['poly_deg'] = 1
init_dict_MPK['Sigma_pos_par_init_list'] = [np.ones(gp_input_dim+1)]+[np.ones((deg+1)*(gp_input_dim))
    for deg in range(1,init_dict_MPK['poly_deg'])]
init_dict_MPK['flg_train_Sigma_pos_par_list'] = [True]*init_dict_MPK['poly_deg']
init_dict_MPK['dtype'] = dtype
init_dict_MPK['device'] = device
model_learning_par['init_dict_list'] = [[init_dict_RBF, init_dict_MPK]]*state_dim
```

*(E)* Set the exploration policy and the load the target trajectory $\{\mathbf{q}_t^r, \dot{\mathbf{q}}_t^r\}_{t=0}^H$. The exploration policy is a poorly-tuned PD controller, parameters `sqrt_Kp_gains` and `sqrt_Kd_gains` define the squared values for the proportional and derivative gains, respectively.

```
print('\n---- Load target trajectory ----')
target_traj = np.genfromtxt('envs/target_q_trajectory.csv', delimiter=',')

print('\n---- Set exploration policy ----')
f_rand_exploration_policy = Policy.PD_controller
rand_exploration_policy_par = {}
rand_exploration_policy_par['state_dim'] = state_dim
rand_exploration_policy_par['input_dim'] = input_dim
rand_exploration_policy_par['u_max'] = u_max
rand_exploration_policy_par['dtype'] = dtype
rand_exploration_policy_par['device'] = device
rand_exploration_policy_par['sqrt_Kp_gains'] = [1., 1., 1., 1., 1, 1.]
rand_exploration_policy_par['sqrt_Kd_gains'] = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1]
rand_exploration_policy_par['target_traj'] = torch.tensor(target_traj,dtype=dtype,device=device)
```

*(F)* Set the control policy `Sum_of_gaussians_with_target_trajectory()`. This policy is a *squashed-RBF-network* (defined by the same parameters described in section *(F)* of Scenario 1 that receives in input, besides the state, also the tracking errors $\mathbf{q}_t^r - \mathbf{q}_t$ and theirs derivatives $\dot{\mathbf{q}}_t^r - \dot{\mathbf{q}}_t$.

```
print('\n---- Set control policy ----')
num_basis = 400
f_control_policy = Policy.Sum_of_gaussians_with_target_trajectory
control_policy_par = {}
control_policy_par['state_dim'] = state_dim*2
control_policy_par['input_dim'] = input_dim
control_policy_par['u_max'] = u_max
control_policy_par['num_basis'] = num_basis
control_policy_par['dtype'] = dtype
control_policy_par['device'] = device
control_policy_par['centers_init'] = np.concatenate([np.pi/2*2*(np.random.rand(num_basis,state_dim)-0.5)
    ,0.1*2*(np.random.rand(num_basis,state_dim)-0.5)],1)
control_policy_par['lengthscales_init'] = np.pi*np.ones(state_dim*2)
control_policy_par['weight_init'] = 1.*2*(np.random.rand(input_dim,num_basis)-0.5)
control_policy_par['target_traj'] = torch.tensor(target_traj,dtype=dtype,device=device)
control_policy_par['flg_squash'] = True
control_policy_par['flg_drop'] = True
policy_reinit_dict = {}
policy_reinit_dict['lenghtscales_par'] = 10.*np.ones(state_dim*2)
policy_reinit_dict['centers_par'] = np.concatenate([np.pi/2*np.ones([num_basis,state_dim]),0.1*np.ones([
    num_basis,state_dim])],1)
policy_reinit_dict['weight_par'] = 1.
```

*(G)* Set the cost `Expected_saturated_distance_from_trajectory()`, that is defined as: $1 - \exp\left(-\left(\frac{||\mathbf{q}_t^r - \mathbf{q}_t||}{l_q}\right)^2 - \left(\frac{||\dot{\mathbf{q}}_t^r - \dot{\mathbf{q}}_t||}{l_{\dot{q}}}\right)^2\right)$. $l_q$ and $l_{\dot{q}}$ are defined by the parameter `lengthscales`.

```
print('\n---- Set cost function ----')
cost_function_par = {}
f_cost_function = Cost_function.Expected_saturated_distance_from_trajectory
cost_function_par['target_traj'] = torch.tensor(target_traj,dtype=dtype,device=device)
cost_function_par['lengthscales'] = torch.tensor([0.5,0.5,0.5,0.5,0.5,0.5,1.,1.,1.,1.,1.,1.], dtype=
    dtype,device=device)
cost_function_par['used_indeces'] = list(range(0,12))
```

(H) Previously defined parameters are used for the initialization of the policy learning object `PL_obj`. Please note that in this case we must use `MC_PILCO_mujoco_envs.MC_PILCO_Mujoco()`, the class in charge of handling interaction with MuJoCo environments.

```
print('\n---- Init policy learning object ----')
MC_PILCO_init_dict = {}
MC_PILCO_init_dict['T_sampling'] = T_sampling
MC_PILCO_init_dict['state_dim'] = state_dim
MC_PILCO_init_dict['input_dim'] = input_dim
MC_PILCO_init_dict['f_sim'] = env_name
MC_PILCO_init_dict['std_meas_noise'] = std_list
MC_PILCO_init_dict['f_model_learning'] = f_model_learning
MC_PILCO_init_dict['model_learning_par'] = model_learning_par
MC_PILCO_init_dict['f_rand_exploration_policy'] = f_rand_exploration_policy
MC_PILCO_init_dict['rand_exploration_policy_par'] = rand_exploration_policy_par
MC_PILCO_init_dict['f_control_policy'] = f_control_policy
MC_PILCO_init_dict['control_policy_par'] = control_policy_par
MC_PILCO_init_dict['f_cost_function'] = f_cost_function
MC_PILCO_init_dict['cost_function_par'] = cost_function_par
MC_PILCO_init_dict['log_path'] = 'results_tmp/'+str(seed)
MC_PILCO_init_dict['dtype'] = dtype
MC_PILCO_init_dict['device'] = device
MC_PILCO_init_dict['sim_timestep'] = sim_timestep
PL_obj = MC_PILCO_mujoco_envs.MC_PILCO_Mujoco(**MC_PILCO_init_dict)
```

*(I)* Specify model optimization settings.

```
print('\n---- Set MC-PILCO options ----')
# Model optimization options
model_optimization_opt_dict = {}
model_optimization_opt_dict['f_optimizer'] = 'lambda p : torch.optim.Adam(p, lr=0.01)'
model_optimization_opt_dict['criterion'] = Likelihood.Marginal_log_likelihood
model_optimization_opt_dict['N_epoch'] = 2001
model_optimization_opt_dict['N_epoch_print'] = 500
model_optimization_opt_list = [model_optimization_opt_dict]*num_gp
```

*(J)* Specify policy optimization settings.

```
# Policy optimization options
policy_optimization_dict = {}
policy_optimization_dict['num_particles'] = 200
policy_optimization_dict['opt_steps_list'] = [5000,5000]
policy_optimization_dict['lr_list'] = [0.01, 0.01]
policy_optimization_dict['f_optimizer'] = 'lambda p, lr : torch.optim.Adam(p, lr)'
policy_optimization_dict['num_step_print'] = 200
policy_optimization_dict['p_dropout_list'] = [0.25, 0.25]
policy_optimization_dict['p_drop_reduction'] = 0.25/2
policy_optimization_dict['alpha_diff_cost']=0.99
policy_optimization_dict['min_diff_cost']= 0.04
policy_optimization_dict['num_min_diff_cost'] = 400
policy_optimization_dict['min_step'] = 400
policy_optimization_dict['lr_reduction_ratio']=0.5
policy_optimization_dict['lr_min'] = 0.0025
policy_optimization_dict['policy_reinit_dict'] = policy_reinit_dict
```

*(K)* Initialization of the policy optimization method and definition of initial state distribution.

```
# Options for method reinforce
reinforce_param_dict = {}
reinforce_param_dict['initial_state'] = target_traj[0,:]
reinforce_param_dict['initial_state_var'] = np.array([10**-6, 10**-6, 10**-6, 10**-6, 10**-6, 10**-6,
    10**-6, 10**-6, 10**-6, 10**-6, 10**-6, 10**-6])
reinforce_param_dict['T_exploration'] = T_exploration
reinforce_param_dict['T_control'] = T_control
```

```
reinforce_param_dict['num_trials'] = num_trials
reinforce_param_dict['random_initial_state'] = False
reinforce_param_dict['model_optimization_opt_list'] = model_optimization_opt_list
reinforce_param_dict['policy_optimization_dict'] = policy_optimization_dict
```

*(L)* Save experiment configuration.

```
print('\n---- Save test configuration ----')
config_log_dict = {}
config_log_dict['MC_PILCO_init_dict'] = MC_PILCO_init_dict
config_log_dict['reinforce_param_dict'] = reinforce_param_dict
pkl.dump(config_log_dict,open('results_tmp/'+str(seed)+'/config_log.pkl','wb'))
```

*(M)* Run the policy learning method.

```
# Start the learning algorithm
PL_obj.reinforce(**reinforce_param_dict)
```

## Plot results

All the test files save log files inside `results_tmp`, in a folder named by the adopted random seed (to be created). By running the `log_plot_` files associated to each experiment, it is possible to plot the obtained results (specify the random seed desired in the command line). For example by running `python log_plot_cartpole.py -seed 1`.

## Run repeated tests (with different random seeds)

With `repeat_test.py` it is possible to repeat several times a file to test the same scenario with different random seeds (as it was done in the ablation studies presented in the article). It is necessary to specify the random seeds used and create a folder for each one inside `results_tmp`.

## Load policy/model from log

Method `reinforce()` saves all data and parameters of the policy and the model in log files stored inside folder `results_tmp`, by default. It is possible to load any learned policy and model from log files by using methods `load_policy_from_log()` and `load_model_from_log()`, respectively. Both methods take as input a `folder` parameter, that specifies the path where the log file is located, and a `num_trial` parameter, that indicates the particular number of trial from which the user wants to load the policy/model. Examples of their use are provided in the files:

**`apply_mcpilco_policy.py` / `apply_mcpilco4pms_policy.py`**
The policy learned at trial `num_trial` with MC-PILCO/MC-PILCO4PMS is loaded (from the folder indicated by `seed`) and it is applied `num_trial` times to the cart-pole system.

**`apply_mcpilco_policy_on_model.py` / `apply_mcpilco4pms_policy_on_model.py`**
The policy and the model learned at trial `num_trial` with MC-PILCO/MC-PILCO4PMS are loaded (from the folder indicated by `seed`) and the policy is applied on `num_particles` particles simulated with the obtained GP model.

## Re-start training after loading

After loading model and policy from log files, as described at previous point, it is possible to re-start the learning procedure using `reinforce()`, by setting `loaded_model = True` in the method parameters. In this case, initial exploration is not performed and the optimization starts from the loaded policy parameters and models.

## References


1. Amadio, F. *et al.* Model-based policy search using monte carlo gradient estimation with real systems application. *IEEE Transactions on Robotics* **38**, 3879–3898 (2022).
2. PyTorch. *https://pytorch.org/*.
3. NumPy. *https://numpy.org/*.
4. Matplotlib. *https://matplotlib.org/*.
5. Pickle. *https://docs.python.org/3/library/pickle.html*.

**6.** Argparse. *https://docs.python.org/3/library/argparse.html*.

**7.** MuJoCo. *mujoco.org*.

**8.** MuJoCo-Py. *https://github.com/openai/mujoco-py*.

**9.** Gym. *http://gym.openai.com/*.