OLCAR - Exercise 3 - Path Integral Policy Improvement
Handout: 12.05.2015
Due: 28.05.2015

Introduction

In exercise 1, you designed an ILQC controller that was able to control a quadrotor for performing agile maneuvers such as flying to a defined point and/or passing through some predefined via-points. ILQC is a model-based optimal control approach which combines both the trajectory and the controller design into a single problem. In exercise 1, you assumed complete knowledge about the system model, without any uncertainty in the system parameters. In general, model-based algorithms like ILQC show superior performance in simulation, but their performance in real physical applications is limited by the accuracy of the system model. In fact, our commonly used system models are usually intentionally designed in a simplistic way - in order to keep complexity and the dimensions of the state-space low and in order to maintain useful mathematical properties. These limitations typically restrict the performance of model-based methods on real robots.

Another approach to simultaneously finding reference trajectories and controllers is to use model-free machine learning techniques, like the *Path Integral Policy Improvement* methods that were presented in the lecture. However, major drawbacks of existing learning algorithms are that their success highly depends on the quality of the initial guess (since they are all local methods), and they just partially exploit the domain knowledge of the system designer, i.e. a known system model.

In this exercise, you will combine the benefits of model-based an model-free approaches. You will adapt a model-based optimal controller (that you derived in exercise 1 using ILQC) using a PI2 learning algorithm.

This in fact is a real scenario when we are dealing with an actual robot. Although often we have a model of the robot dynamics, the model is based on some assumptions and simplifications. Therefore, the designed controller based on this model performs noticeably different on the real robot. In this case, we implement a learning algorithm which adapts the model-based controller on the fly.

Software Structure

In this exericse, we use the same quadrotor platform as in exercise 1. Details about the system description, state, inputs and dynamics can be found in the exercise 1 handout. The MATLAB code structure, as given in `main_ex3.m`, is given below. Generally the steps executed in the main function are similar to those in exercise 1:

- `Task = Task_Design()` defines a task including start and goal state, etc., for both the initializing ILQC controller and the Path Integral Learning problem. For the latter, it additionally sets the number of gaussian basis functions, the number of rollouts per iteration, the maximum number of iterations, the exploration noise and how many (of the best) rollouts per iteration are kept and reused in the next iteration. For making your implementation work, you should not have to change this function.

- `load('Quadcopter_Model.mat','Model')` defines the dynamics of the nominal quadrotor model which is used for calculating an initial ILQC controller.
- `Task.cost = Cost_Design(Model.param.mQ,Task)` generates the cost functions for both the ILQC and the path integral learning. Note that the cost functions are identical for both problems parts - in other words, both algorithm optimize the same cost functions.
- `LQR_Design(Model,Task)` designs an LQR controller as initial guess for the ILQC controller, similar to exercise 1, using the nominal model. `ILQC_Design(Model,Task,Initial_Controller,@Quad_Simulator)` subsequently designs an ILQC controller for the given task and the nominal model.
- In `BaseFcnTrans(ILQC_Controller,Task.n_gaussian)`, the ILQC controller is transformed into basis-function representation, which later serves as initialization to the path integral learning algorithm. ILQC provides a time-indexed state feedback plus feedforward. Representing the initial ILQC controller in terms of a parameter matrix θ which acts on the basis functions drastically reduces the number of control parameters.
- `load('Quadcopter_Model_perturbed.mat','Model')` defines the dynamics of the 'real' quadrotor for generating samples for the Path Integral learning steps.
- `PIs_Learning(Model_perturbed,Task,ReducedController)` implements the PI2 learning. In every iteration, it carries out the steps described in Algorithm 10 from chapter 3 in the script. It calls the function `PI2_Update(Task,bsim_out,bR)`, which calculates updates for the parameter-matrix θ in every iteration.
- `Sample_Rollout(Model,Task,Controller)`: generates a sample rollout of the specified model and Task subject to a certain controller. The learned controller is shown in a visualization similar to exercise 1.
- Lastly, a plot is generated which shows the cost improvement over the number of PI2 learning iterations.

Exercise 3 Problem Statement

The key idea: connecting ILQC with PI2 In exercise 1, you completed the function `ILQC_Design(Model,Task,Initial_Controller,@Quad_Simulator)`, which calculates the full ILQC controller for the specified problem. In the first part, use your *own* ILQC controller implementation and the nominal quadrotor model to generate a suitable initial guess for the learning algorithm. As backup, we provide a protected version of a working ILQC implementation, however, we strongly encourage you to use your own design.

PI2 Update function implementation Now, we put aside the assumption of complete model knowledge. The 'real' robotic system is now simulated by a model that is perturbed in the parameters. In the learning part, those true system parameters are unknown to the algorithm, however, it can draw samples from the simulator imitating the 'real' robot.

In this exercise, you are asked to implement a part of the General PI2 algorithm, which is Algorithm 10 in the lecture notes. The protected function called

```
[LearnedController,...,  ...] = PIs_Learning(Model_perturbed,Task,ReducedController)
```

provides the framework for the algorithm, including the initialization, the exploration-noise annealing and drawing samples from the perturbed system. However, the function

```
delta_theta = PI2_Update(Task,batch_sim_out,batch_cost)
```

must be completed. It gets called by `PIs_Learning` and spans the missing part of the lecture notes' Algorithm 10 (shown in Algorithm 1 of this exercise sheet).

Your task is to fill in the missing parts of the function `PI2_Update`. Its input arguments are the following:

- `Task` is a struct which contains all task specific parameters, similar to Exercise 1. Additionally it contains PI2 specific parameters like `num_rollouts`, `n_basefct`, the exploration noise `std_noise`, etc.
- `batch_sim_out` assume we are given a problem with state-space dimension $(n \times 1)$, input-space dimensions $(p \times 1)$, g gaussian basis functions and r rollouts per iteration. Then, `batch_sim_out` is a struct array of dimensions $(r \times 1)$ which contains all rollouts' simulation data in batch format. It has the fields
  - .t vector of discrete simulation times for every rollout
  - .x state trajectory for every rollout
  - .u input trajectory for every rollout
  - .Controller The controller structure. It contains the fields

Algorithm 1 Excerpt of General PI2 Algorithm

```
for the $i$th control input do
    for each time, s do
        Calculate the Return from starting time $s$ for the $k$th rollout:
            $R(\tau^k(s)) = \Phi(\mathbf{x}(t_f)) + \int_s^{t_f} \left(q(t,\mathbf{x}) + \frac{1}{2}\mathbf{u}^T\mathbf{R}\mathbf{u}\right) dt$
        Calculate $\alpha$ from starting time $s$ for the $k$th rollout:
            $\alpha^k(s) = {\exp(-\frac{1}{\lambda}R(\tau^k(s)))}/{\sum_{k=1}^K \exp(-\frac{1}{\lambda}R(\tau^k(s)))}$
        Calculate the time varying parameter increment $\Delta\boldsymbol{\theta}_i(s)$:
            $\Delta\boldsymbol{\theta}_i(s) = \sum_{k=1}^K \alpha^k(s) \frac{\boldsymbol{\Upsilon}(s)\boldsymbol{\Upsilon}^T(s)}{\boldsymbol{\Upsilon}^T(s)\boldsymbol{\Upsilon}(s)} \boldsymbol{\epsilon}_i^k(s)$
    end for
    for the $j$th column of $\Delta\boldsymbol{\theta}_i$ matrix, $\Delta\boldsymbol{\theta}_{i,j}$ do
        Time-averaging the parameter vector
            $\Delta\boldsymbol{\theta}_{i,j} = \left(\int_{t_0}^{t_f} \Delta\boldsymbol{\theta}_{i,j}(s) \circ \boldsymbol{\Upsilon}(s) ds\right) \cdot \Big/ \int_{t_0}^{t_f} \boldsymbol{\Upsilon}(s) ds$
    end for
    Update parameter vector for control input $i$, $\boldsymbol{\theta}_i$:
        $\boldsymbol{\theta}_i \leftarrow \boldsymbol{\theta}_i + \omega\Delta\boldsymbol{\theta}_i$
end for
```

    * `@BaseFnc(t, x)` a function which returns a matrix of dimensions $((n+1)\cdot g \times \text{length}(t))$, indicating the basis function activation for every element of t.
    * `.theta` the parameter matrix of dimensions $((g \cdot (n+1)) \times p)$

  - `.eps` contains the scaled noise for parameter perturbation and is of dimensions $((g \cdot (n+1)) \times p \times \text{length}(t))$. The noise gets reduced automatically as the number of rollouts increases.

- `batch_cost` a matrix of dimensions $(r \times \text{lenght}(t))$ which holds the cost at every timestep for every rollout.

Furthermore, the function already contains code for calculating the exponentiated cost, as well as the two additional functions `vec2mat(vec)` and `mat2vec(mat)`. The latter two functions are provided because the quadrotor simulator requires the parameter matrix θ to be given in a different shape than in Algorithm 10 in the lecture notes. Therefore, conversions are required at the beginning and the end of the function you are to implement. For example, calling something like
`temp = sim_out.Controller.BaseFnc(sim_out.t,sim_out.x)` will give a vector-like parameter representation, which first needs to be converted by calling `vec2mat(temp)`. Similarly, at the end of your implementation, your `delta_theta` needs to be converted back to the vector-like representation by calling `mat2vec(delta_theta)`.

Additional Questions

Answer the following questions in .pdf format with max. 5 sentences per question and upload the document to the course website.

1. How much cost improvement did you obtain using PI2 learning? *(answer in 1 sentence and attach one of your cost-plots)*
2. How does the exploration noise (`Task.std_noise`) affect the learning curve? What happens if you decrease/increase it?
3. The tuning parameter `Task.num_reuse` specifies how many (of the best) rollouts are saved, carried over and reused in the next learning iteration. Why does it make sense to keep some of the best rollouts for the next update?
4. How does the quality of your initial guess affect the PI2 learning? For example, what happens if you limit your ILQC iterations to only 1?
5. While executing your program, you might have noticed that the cost is not always strictly decreasing during learning. What is your explanation for this behaviour?

Deliverables

On the course website `http://www.adrl.ethz.ch/doku.php/adrl:education:lecture:fs2015` a skeleton of the matlab software can be downloaded. Please complete the missing parts and upload your `{PI2_Update.m}` file through the appropriate form on the website. Additionally, upload a .pdf with the answers to the posed questions (no more than 5 sentences per question, max. 1 A4 page in total). All this material must be uploaded by **28.05.2015, 11:59 pm**.

The doodle poll for the interview sign up is available at `https://ethz.doodle.com/bsi7gvkycvrmht6t`