Torque-limited Simple Pendulum: Reference

Felix Wiebe, Jonathan Babel,

Contributors:

Shivesh Kumar, Shubham Vyas, Daniel Harnack, Melya Boukheddimi, Mihaela Popescu, Frank Kirchner

March 2022

TABLE OF CONTENTS

CHAPTER

ONE

INSTALLATION GUIDE

In order to execute the python code within the repository you will need to have *Python (>=3.6, <4)* along with the package installer *pip3* on your system installed.

- **python (>=3.6, <4)**
- **pip3**

If you aren't running a suitable python version currently on your system, we recommend you to install the required python version inside of an virtual environment for python (**pyenv**) and to install all python packages necessary to use this repo afterwards inside a newly created virtual environment (**virtualenv**). The installation procedure for Ubuntu (18.04.5 and 20.04.2.0 LTS) are described in the next section. You can find instructions for MacOS and other Linux distributions, as well as information about common build problems here:

- **pyenv:** https://github.com/pyenv/pyenv
- **virtualenv:** https://github.com/pyenv/pyenv-virtualenv

1.1 Installing this Python Package

If you want to install this package with your system python version, you can do that by going to the directory software/python and typing:

```
pip install .
```

Note: This has to be repeated if you make changes to the code (besides the scripts).

1.2 Instructions for Ubuntu (18.04.5 and 20.04.2.0 LTS)

The instructions provide assistance in the setup procedure, but with regards to the software LICENSE they are provided without warranty of any kind. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, arising from, out of or in connection with the software or the use or other dealings in the software.

1. Clone this repo from GitHub, in case you haven't done it yet:

```
git clone git@github.com:dfki-ric-underactuated-lab/torque_limited_simple_
↪pendulum.git
```

2. Check your Python version with:

```
python3 --version
```

If you are already using suitable Python 3.6 version jump directly to step *Creating a Virtual Environment* otherwise continue here and first install a virtual environment for python.

1.3 Pyenv: Virtual environment for Python

The following instructions are our recommendations for a sane build environment.

1. Make sure to have installed python's binary dependencies and build tools as per:

```
sudo apt-get update
sudo apt-get install make build-essential libssl-dev zlib1g-dev
libbz2-dev libreadline-dev libsqlite3-dev wget curl llvm
libncursesw5-dev xz-utils tk-dev libxml2-dev libxmlsec1-dev libffi-dev␣
↪liblzma-dev
```

2. Once prerequisites have been installed correctly, install Pyenv with:

```
curl https://pyenv.run | bash
```

3. Configure your shell's environment for Pyenv

> **Note:** The below instructions are designed for common shell setups. If you have an uncommon setup and they don't work for you, use the linked guidance to figure out what you need to do in your specific case: https://github.com/pyenv/pyenv#advanced-configuration

Before editing your *.bashrc* and *.profile* files it is a good idea to <ins>make a copy</ins> of both files in case something goes wrong. Add pyenv to your *.bashrc* file from the terminal:

```
echo 'eval "$(pyenv init -)"' >> ~/.bashrc
```

Add these lines at the beginning of your *.profile* file (not from the terminal):

```
export PYENV_ROOT="$HOME/.pyenv"
export PATH="$PYENV_ROOT/bin:$PATH"
```

and this line at the very end of your *.profile* file (not from the terminal):

```
eval "$(pyenv init --path)"
```

4. Source *.profile* and *.bashrc*, then restart your shell so the path changes take effect:

```
source ~/.profile
source ~/.bashrc

exec $SHELL
```

5. Run *pyenv init* - in your shell, then copy and also execute the output to enable shims:

```
pyenv init -
```

Restart your login session for the changes to take effect. If you're in a GUI session, you need to fully log out and log back in. You can now begin using pyenv.

Note: Consider uprading to the latest version of Pyenv via:

pyenv update

1.4 Installing Python into Pyenv

You can diplay a list of available Python versions with:

```
pyenv install -l | grep -ow [0-9].[0-9].[0-9]
```

Install your desired Python version using Pyenv (We suggest 3.6.9 for ubuntu 18.04 and 3.8.10 for ubuntu 20.04):

```
pyenv install 3.x.x
```

Double check your work:

```
pyenv versions
```

To use Python 3.x only for this specific project change directory to the cloned git repo and type:

```
pyenv local 3.x.x
```

1.5 Creating a Virtual Environment with Pyenv

In order to clutter your system as little as possible all further packages will be installed inside a virtual environment, which can be easily removed at any time. The recommended way to configure your own custom Python environment is via *Virtualenv*.

1. Clone virtualenv from https://github.com/pyenv/pyenv-virtualenv into the pyenv-plugin directory:

```
git clone https://github.com/pyenv/pyenv-virtualenv.git $(pyenv root)/
↪plugins/pyenv-virtualenv
```

2. Add pyenv virtualenv-init to your shell to enable auto-activation of virtualenvs:

```
echo 'eval "$(pyenv virtualenv-init -)"' >> ~/.bashrc
```

3. Restart your shell to enable pyenv-virtualenv:

```
exec "$SHELL"
```

4. To create a new virtual environment, e.g. named simple-pendulum with Python 3.6.9 run:

```
pyenv virtualenv 3.6.9 simple-pendulum
```

5. Activate the new virtual environment with the command:

```
pyenv activate simple-pendulum
```

The name of the current virtual environment *(venv)* appears to the left of the prompt, indicating that you are now working inside a virtual environment. When finished working in the virtual environment, you can deactivate it by running the following:

```
pyenv deactivate
```

In case that you don't need the virtual environment anymore, you can deactivate it and remove it together with all previously installed packages:

```
pyenv uninstall simple-pendulum
```

1.6 Installing pip3

Update the package list inside your recently created virtual environment:

```
sudo apt update
```

and install pip3 via:

```
sudo apt install python3-pip
```

If you like, you can update pip and verify your the installation by:

```
pip install --upgrade pip
pip3 --version
```

Robotics Innovation Center
Robert-Hooke-Str. 1
28359 Bremen
Deutschland
robotik@dfki.de

1.7 Install Requirements for this Repository

Navigate inside your cloned git repo to */torque_limited_simple_pendulum/software/python* and make sure your virtual environment is active *pyenv activate simple-pendulum*. Now you can install version specific packages for all required packages from the *requirements.txt* file via:

```
python3 -m pip install -r requirements.txt
```

Note:

1. You can generate your own requirements.txt file with this command: *pip freeze > requirements.txt*
2. You can skip this step and directly install this package with the setup.py file as described in the next line. This will install the requirement packages as well. The setup.py will not install specific versions of the requirements, that have been tested by us, but instead, it will install the latest version.

This package then can be installed from the software/python directory by typing:

```
pip install .
```

This was the final installation step. Your system is now prepared to run all code snippets from this repo. Have fun exploring all kind of different simple pendulum controllers!

1.8 OPTIONAL: Gepetto Viewer

The optimal control library Crocoddyl has an interface to the gepetto-viewer for visualization. For installing the gepetto viewer we refer to their github repository.

CHAPTER

TWO

USAGE INSTRUCTIONS

2.1 Installing Python Driver for T-Motor AK80-6 Actuator

1. Clone the python motor driver from: https://github.com/dfki-ric-underactuated-lab/mini-cheetah-tmotor-python-can
2. Modify the *.bashrc* file to add the driver to your python path. Make sure you restart your terminal after this step.:

```
# mini-cheetah driver
export PYTHONPATH=~/path/from/home/to/underactuated-robotics/python-motor-
↪driver:${PYTHONPATH}
```

Make sure you setup your can interface first. The easiest way to do this is to run *sh setup_caninterface.sh* from the *mini-cheetah-motor/python-motor-driver* folder. To run an offline computed swingup trajectory, use: *python3 swingup_control.py*. The script assumes can id as *'can0'* and motor id as *0x01*. If these parameters differ, please modify them within the script. Alternatively, the motor driver can also be installed via pip from https://pypi.org/project/mini-cheetah-motor-driver-socketcan/.:

```
pip install mini-cheetah-motor-driver-socketcan
```

2.2 Setting up the CAN interface

1. Run this command and make sure that *can0* (or any other can interface depending on the system)shows up as an interface after connecting the USB cable to your laptop: *ip link show*
2. Configure the *can0* interface to have a 1 Mbaud communication frequency: *sudo ip link set can0 type can bitrate 1000000*
3. To bring up the *can0* interface, run: *sudo ip link set up can0*

> **Note:** Alternatively, one could run the shell script *setup_caninterface.sh* which will do the job for you.

4. To change motor parameters such as CAN ID or to calibrate the encoder, a serial connection is used. The serial terminal GUI used on linux for this purpose is *cutecom*

2.3 Testing Communication

To enable one motor at *0x01*, set zero position and disable the motor, run: *python3 can_motorlib_test.py can0*

Use in Scripts: Add the following import to your python script: *from canmotorlib import CanMotorController* after making sure this folder is available in the import path/PYTHONPATH.

Example Motor Initialization:

```
`motor = CanMotorController(can_socket='can0', motor_id=0x01, socket_timeout=0.
↪5)`
```

Available Functions:

- *enable_motor()*
- *disable_motor()*
- *set_zero_position()*
- *send_deg_command(position_in_degrees, velocity_in_degrees, Kp, Kd, tau_ff):*
- *send_rad_command(position_in_radians, velocity_in_radians, Kp, Kd, tau_ff):*

All functions return current position, velocity, torque in SI units except for *send_deg_command*.

Performance Profiler: Sends and received 1000 zero commands to measure the communication frequency with 1/2 motors. Be careful as the motor torque will be set to zero.

2.4 Using different Controllers for the Swing-Up

All implemented controllers can be called from the *main.py* file. The desired controller is selected via a required flag, e.g. if you want to execute the gravity compensation experiment the corresponding command would then be:

```
python main.py -gravity
```

Make sure you execute the command from the directory */software/python/examples_real_system* otherwise you have to specify the path to *main.py* as well. If you want to autosave all data from your experiment use the optional flag *-save* together with the flag required for the controller.

To get an overview of all possible arguments display help documentation via:

```
python main.py -h
```

Robotics Innovation Center
Robert-Hooke-Str. 1
28359 Bremen
Deutschland
robotik@dfki.de

2.5 Saving Results

The results folder serves as the directory, where all results generated from the python code shall be stored. The distinct seperation between python script files and generated output files helps to keep the python package clear and tidy. We provide some example output data from the very start, so that you may see what results each script produces even before you run the code. The tools to create result files from the respective experiment data are located within the python package under */utilities*, *plot.py*, *plot_policy.py* and *process_data.py*.

In general particular functions get called from *main.py* or another script to produce the desired output, which improves resuability of the utilities and keeps the code concise. The results of each experiment are saved in a new folder, which is automatically assigned a timestamp and an appropriate name.

CHAPTER

THREE

HOW TO TEST THE CODE

For verifying the functionality of the python code, first install pytest via:

```
pip install -U pytest
```

Note: If you install pytest inside a virtual environment, you have to deactivate and then reactivate the environment for pytest to be used in the virtual environment. Every trajectory optimization algorithm and every controller in this software package has a unit_test.py file which verfies the functionality of the corresponding piece of software. The pendulum plant has a unit_test.py script as well.

pytest automatically identifies all python scripts with names `\*_test.py` or `test_*.py` in the current directory and all subdirectories. Therefore, if you want to perform all unit tests in this software package, simply go to the python root directory and type:

```
python -m pytest
```

If you want perform a specific unit test (e.g. for the lqr controller) use:

```
pytest software/python/controller/lqr/unit_test.py
```

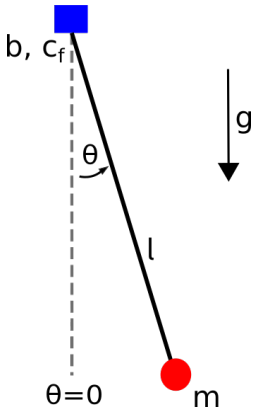
CHAPTER

FOUR

THE PHYSICS OF A SIMPLE PENDULUM

4.1 Equation of Motion

$$I\ddot{\theta} + b\dot{\theta} + c_f \text{sign}(\dot{\theta}) + mgl\sin(\theta) = \tau$$



where

- θ, $\dot{\theta}$, $\ddot{\theta}$ are the angular displacement, angular velocity and angular acceleration of the pendulum. $\theta = 0$ means the pendulum is at its stable fixpoint (i.e. hanging down).
- I is the inertia of the pendulum. For a point mass: $I = ml^2$
- m mass of the pendulum
- l length of the pendulum
- b damping friction coefficient
- c_f coulomb friction coefficient
- g gravity (positive direction points down)
- τ torque applied by the motor

The pendulum has two fixpoints, one of them being stable (the pendulum hanging down) and the other being unstable (the pendulum pointing upwards). A challenge from the control point of view is to swing the pendulum up to the unstable fixpoint and stabilize the pendulum in that state.

Robotics Innovation Center
Robert-Hooke-Str. 1
28359 Bremen
Deutschland
robotik@dfki.de

4.2 Energy of the Pendulum

- Kinetic Energy (K)

$$K = \frac{1}{2} m l^2 \dot{\theta}^2$$

- Potential Energy (U)

$$U = -mgl\cos(\theta)$$

- Total Energy (E)

$$E = K + U$$

4.3 `PendulumPlant`

The PendulumPlant class contains the kinematics and dynamics functions of the simple, torque limited pendulum.

The pendulum plant can be initialized as follows:

```
pendulum = PendulumPlant(mass=1.0,
                         length=0.5,
                         damping=0.1,
                         gravity=9.81,
                         coulomb_fric=0.02,
                         inertia=None,
                         torque_limit=2.0)
```

where the input parameters correspond to the parameters in the equaiton of motion (1). The input inertia=None is the default and the intertia is set to the inertia of a point mass at the end of the pendulum stick $I = ml^2$. Additionally, a torque_limit can be passed to the class. Torques greater than the torque_limit or smaller than -torque_limit will be cut off.

The plant can now be used to calculate the forward kinematics with:

```
[[x,y]] = pendulum.forward_kinematics(pos)
```

where pos is the angle θ of interest. This function returns the (x,y) coordinates of the tip of the pendulum inside a list. The return is a list of all link coordinates of the system (as the pendulum has only one, this returns [[x,y]]).

Similarily, inverse kinematics can be computed with:

```
pos = pendulum.inverse_kinematics(ee_pos)
```

where ee_pos is a list of the end_effector coordinates [x,y]. pendulum.inverse_kinematics returns the angle of the system as a float.

Forward dynamics can be calculated with:

```
accn = pendulum.forward_dynamics(state, tau)
```

where state is the state of the pendulum $[\theta, \dot{theta}]$ and tau the motor torque as a float. The function returns the angular acceleration.

For inverse kinematics:

tau = pendulum.inverse_kinematics(state, accn)

where again state is the state of the pendulum $[\theta, \dot{theta}]$ and accn the acceleration. The function return the motor torque τ that would be neccessary to produce the desired acceleration at the specified state.

Finally, the function:

```
res = pendulum.rhs(t, state, tau)
```

returns the integrand of the equaitons of motion, i.e. the object that can be calculated with a time step to obtain the forward evolution of the system. The API of the function is written to match the API requested inside the simulator class. *t* is the time which is not used in the pendulum dynamics (the dynamics do not change with time). state again is the pendulum state and tau the motor torque. res is a numpy array with shape np.shape(res)=(2,) and res = $[\dot{\theta}, \ddot{theta}]$.

4.3.1 Usage

The class is inteded to be used inside the simulator class.

4.3.2 Parameter Identification

The rigid-body model derived from a-priori known geometry as described previously has the form

$$\tau(t) = \mathbf{Y}\left(\theta(t), \dot{\theta}(t), \ddot{\theta}(t)\right) \ \lambda$$

where actuation torques τ, joint positions $\theta(t)$, velocities $\dot{\theta}(t)$ and accelerations $\ddot{\theta}(t)$ depend on time *t* and $\lambda \in \mathbb{R}^{6n}$ denotes the parameter vector. Two additional parameters for Coulomb and viscous friction are added to the model, $F_{c,i}$ and F_{v,i}, in order to take joint friction into account. The required torques for model-based control can be measured using stiff position control and closely tracking the reference trajectory. A sufficiently rich, periodic, band-limited excitation trajectory is obtained by modifying the parameters of a Fourier-Series as described by [^fn3]. The dynamic parameters $\hat{\lambda}$ are estimated through least squares optimization between measured torque and computed torque

$$\hat{\lambda} = \underset{\lambda}{\operatorname{argmin}} \left((\lambda - \tau_m)^T (\lambda - \tau_m) \right),$$

where denotes the identification matrix.

Robotics Innovation Center
Robert-Hooke-Str. 1
28359 Bremen
Deutschland
robotik@dfki.de

4.3.3 References

- **Bruno Siciliano et al.** *Robotics*. Red. by Michael J. Grimble and Michael A. Johnson. Advanced Textbooks in Control and Signal Processing. London: Springer London, 2009. ISBN: 978-1-84628-641-4 978-1-84628-642-1. DOI: 10.1007/978-1-84628-642-1 (visited on 09/27/2021).
- **Vinzenz Bargsten, José de Gea Fernández, and Yohannes Kassahun.** *Experimental Robot Inverse Dynamics Identification Using Classical and Machine Learning Techniques*. In: ed. by International Symposium on Robotics. OCLC: 953281127. 2016. (visited on 09/27/2021).
- **Jan Swevers, Walter Verdonck, and Joris De Schutter.** *Dynamic ModelIdentification for Industrial Robots*. In: IEEE Control Systems27.5 (Oct.2007), pp. 58–71. ISSN: 1066-033X, 1941-000X.doi:10.1109/MCS.2007.904659 (visited on 09/27/2021).

CHAPTER

FIVE

TRAJECTORY OPTIMIZATION

5.1 Trajectory optimization using direct collocation

Note: Type: Trajectory Optimization

State/action space constraints: Yes

Optimal: Yes

Versatility: Swingup and stabilization

5.1.1 Theory

Direct collocation is an approach from **collocation methods** , which transforms the optimal control problem into a mathematical programming problem. The numerical solution can be achieved directly by solving the new problem using sequential quadratic programming [1] <https://arc.aiaa.org/doi/pdf/10.2514/3.20223>`_`[2]. The formulation of the optimization problem at the collocation points is as follows:

$$\begin{aligned} \min_{\mathbf{x}[.],u[.]} & \sum_{n_0}^{N-1} h_n l(u[n]) \\ \text{s.t.} \quad & \dot{\mathbf{x}}(t_{c,n}) = f(\mathbf{x}(t_{c,n}), u(t_{c,n})), \quad \forall n \in [0, N-1] \\ & |u| \leq u_{max} \\ & \mathbf{x}[0] = \mathbf{x}_0 \\ & \mathbf{x}[N] = \mathbf{x}_F \end{aligned}$$

- **x** = $[\theta(.), \dot{\theta}(.)]^T$>: Angular position and velocity are the states of the system
- u: Input torque of the system applied by motor
- *N = 21*: Number of break points in the trajectory
- $h_k = t_k - t_k$: Time interval between two breaking points
- $l(u) = u^T R u$: Running cost
- *R = 10*: Input weight

- $\dot{\mathbf{x}}(t_{c,n}$: Nonlinear dynamics of the pendulum considered as equality constraint at collocation point
- **math`t_{c,k} = frac{1}{2}left(t_k t_{k1}right)`** A collocation point at time instant $k, (i.e., \mathbf{x}[k] = \mathbf{x}(t_k))$,in which the collocation constraints depends on the decision variables $\mathbf{x}[k], \mathbf{x}[k2, B1], u[k], u[k2, B1]$
- $u_{max} = 10$: Maximum torque limit
- $\mathbf{x_0} = [\theta = \mathbf{0}, \dot{\theta} = \mathbf{0}]$: Initial state constraint
- $\mathbf{x_F} = [\theta = \pi, \dot{\theta} = \mathbf{0}]$: Terminal state constraint

Minimum and a maximum spacing between sample times set to 0.05 and 0.5`s`. It assumes a **first-order hold** on the input trajectory, in which the signal is reconstructed as a piecewise linear approximation to the original sampled signal.

5.1.2 API

The direct collocation algorithm explained above can be executed by using the `DirectCollocationCalculator` class:

```
dircal = DirectCollocationCalculator()
```

To parse the pendulum parameters to the calculator, do:

```
dircal.init_pendulum(mass=0.5,
                     length=0.5,
                     damping=0.1,
                     gravity=9.81,
                     torque_limit=1.5)
```

The optimal trajectory can be computed with:

```
x_trajectory, dircol, result = dircal.compute_trajectory(N=21,
                                                         max_dt=0.5,
                                                         start_state=[0.0, 0.0],
                                                         goal_state=[3.14, 0.0])
```

The method returns three pydrake objects containing the optimization results. The trajectory as numpy arrays can be extracted from these objects with:

```
T, X, XD, U = dircal.extract_trajectory(x_trajectory, dircol, result, N=1000)
```

where T is the time, X the position, XD the velocity and U the control trajectory. The parameter N determines here how with how many steps the trajectories are sampled.

The phase space of the trajectory can be plotted with:

```
dircal.plot_phase_space_trajectory(x_trajectory, save_to="None")
```

If a string with a path to a file location is parsed with ‘save_to’ the plot is saved there.

5.1.3 Dependencies

The trajectory optimization using direct collocation for the pendulum swing-up is accomplished by taking advantage of Drake toolbox [3].

5.1.4 References

[1] Hargraves, Charles R., and Stephen W. Paris. "Direct trajectory optimization using nonlinear programming and collocation." Journal of guidance, control, and dynamics 10.4 (1987): 338-342

[2] Russ Tedrake. Underactuated Robotics: Algorithms for Walking, Running, Swimming, Flying, and Manipulation (Course Notes for MIT 6.832).

[3] Model-Based Design and Verification for Robotics <https://drake.mit.edu/>

5.2 Iterative Linear Quadratic Regulator (iLQR)

Note: Type: Trajectory Optimization

State/action space contraints: No

Optimal: Yes

Versatility: Swingup and stabilization

5.2.1 Theory

The *iterative linear quadratic regularizer (iLQR) [1]* <https://ieeexplore.ieee.org/abstract/document/6907001>`_ is an extension of the LQR controller. The LQR controller linearizes the dynamics at a given state and and assumes that these linear dynamics are valid at every other system state as well. In contrast to that, the iLQR optimization method has the ability to take the full system dynamics into account and plan ahead by optimizing over a sequence of control inputs.

The algorithm can be described as:

1. Set an initial state x_0 and an initial control sequence $\mathbf{U} = [u_0, u_1, ..., u_{N-1}]$, where N is the number of steps that will optimized over (the time horizon).
2. Rollout the trajectory by applying the control sequence iteratively to the initial state.

The following steps are repeated until convergence:

3. Backward pass: Compute the derivatives of the cost function and the gains for the control sequence
4. Forward pass: Update the control sequence with the computed gains and rollout the new trajectory with the new control sequence. If the cost of the new trajectory is smaller than before, carry over the new control sequence and increase the gain factor. If not, keep the old trajectory and decrease the gain factor.

If the cost is below a specified threshold the algorithm stops.

5.2.2 API

The iLQR algorithm is computed in the iLQR_Calculator class. The class can be used as follows:

The iLQR calculator has to be initialized with the dimension of the state space (`n_x`) and the dimension of the actuation space (`n_u`) of the system:

```
iLQR = iLQR_Calculator(n_x=2, n_u=1)
```

For the pendulum n_x=2 (positions and velocity) and n_u=1. Next the dynamics and cost function have to be set in the calculator by using:

```
iLQR.set_discrete_dynamics(dynamics)
iLQR.set_stage_cost(stage_cost)
iLQR.set_final_cost(final_cost)
```

where dynamics is a function of the form:

```
dynamics(x, u):
    ...
    return xd
```

i.e. takes the current state and control input as inputs and returns the integrated dynamics. Note that the time step dt is set by the definition of this function.

Similarily, state_cost and final_cost are functions of the form:

```
stage_cost(x,u):
    ...
    return cost

final_cost(x):
    ...
    return cost
```

Warning: These functions have to be differentiable either with the pydrake symbolic library or with sympy! Examples for these functions for the pendulum are implemented in pendulum.py. With the 'partial' function from the 'functools' package additional input parameters of these functons can be set before passing the function with the correct input parameters to the iLQR solver. For an example usage of the partial function for this context see compute_pendulum_iLQR.py in l.80 - l.87 for the dynamics and l.93 - l.113 for the cost functions.

Next: initialize the derivatives and the start state in the iLQR solver:

```
iLQR.init_derivatives()
iLQR.set_start(x0)
```

Finally, a trajectory can now be calculated with:

```
(x_trj, u_trj, cost_trace,
regu_trace, redu_ratio_trace, redu_trace) = iLQR.run_ilqr(N=1000,
                                                          init_u_trj=None,
                                                          init_x_trj=None,
                                                          max_iter=100,
                                                          regu_init=100,
                                                          break_cost_redu=1e-6)
```

The run_ilqr function has the inputs

- `N`: The number of timesteps to plan into the future
- `init_u_trj`: Initial guess for the control sequence (optional)
- `init_u_trj`: Initial guess for the state space trajectory (optional)
- `max_iter`: Maximum number of iterations of forward and backward passes to compute
- `break_cost_redu`: Break cost at which the computation stops early

Besides the state space trajectory `x_trj` and the control trajectory `u_trj` the calculation also returns the traces of the cost, regularization factor, the ratio of the cost reduction and the expected cost reduction and the cost reduction.

5.2.3 Usage

An example script for the pendulum can be found in the examples directory. It can be started with:

```
python compute_iLQR_swingup.py
```

5.2.4 Comments

The iLQR algorithm in this form cannot respect joint and torque limits. Instead, those have to be enforced by penalizing unwanted values in the cost function.

5.2.5 Requirements

Optional: pydrake [2] (see getting_started)

5.2.6 Notes

The calculations with the pydrake symbolic library are about 30% faster than the calculations based on the sympy library in these implementations.

5.2.7 References

[1] Y. Tassa, N. Mansard and E. Todorov, "Control-limited differential dynamic programming," 2014 IEEE International Conference on Robotics and Automation (ICRA), 2014, pp. 1168-1175, doi: 10.1109/ICRA.2014.6907001.

[2] Model-Based Design and Verification for Robotics

5.3 Direct Optimal Control based on the FDDP algorithm

In this package, the single pendulum swing-up is performed using the direct optimal control based on the FDDP algorithm (C. Mastalli, 2019).

The script uses FDDP, BOXFddp can also be used with the same weights. BOXFddp allows to enforce the system's torque limits.

The urdf model is modified to fit a pinocchio model.

5.3.1 Theory

The costs functions for the **Running model** is written as

$$l = \sum_{n=1}^{T-1} \alpha_n \Phi_n(q, \dot{q}, \mathrm{T})$$

With the following costs and weights, t_s denoting the final time horizon.

- **Torque minimization**: Minimization of the joint torques for realistic dynamic motions.

$$\Phi_1 = \| \mathrm{T}(t) \|_2^2, \quad \alpha_1 = 1e-4$$

- **Posture regularization**: giving as input only the final reference posture.

$$\Phi_2 = \| q(t) - q^{ref}(t_{s-1}) \|_2^2, \quad \alpha_2 = 1e-5$$

- The costs functions for the **Terminal model** are applied to only one node (the terminal node) and is written as

$$l_T = \alpha_T \Phi_T(q, \dot{q})$$

With the following cost and weight, $T = t_{final}$ the final time horizon.

- **Posture regularization**: giving as input only the final reference posture.

$$\Phi_3 = \| \, q(T) - q^{ref}(T) \, \|_2^2, \quad \alpha_3 = 10^{10}$$

The weights α_i for this optimization problem are determined experimentally.

5.3.2 API

The BOXFDDP algorithm can be executed with the boxfddp_calculator class. It can be initialized with:

```
ddp = boxfddp_calculator(urdf_path=urdf_path,
                         enable_gui=True,
                         log_dir="log_data/ddp")
```

The urdf_path should point to the urdf of the pendulum in a format that pinochio accepts. The urdf for a simple pendulum can be found in the data folder of this repository: simplependul_dfki_pino_Modi.urdf. enable_gui can be set to True if the trajectory shall be visualized with pinocchio after computation. In the log_dir a urdf with modified pendulum parameters will be stored.

To set the correct pendulum parameters in the urdf with:

```
ddp.init_pendulum(mass=mass,
                  length=length,
                  inertia=inertia,
                  damping=damping,
                  coulomb_friction=coulomb_fric,
                  torque_limit=torque_limit)
```

After that the trajectory can be computed with:

```
T, TH, THD, U = ddp.compute_trajectory(start_state=np.array[0.0, 0.0]),
                                       goal_state=np.array[np.pi, 0.0]),
                                       weights=np.array([1] + [0.1]*1),
                                       dt=4e-2,
                                       T=150,
                                       running_cost_state=1e-5,
                                       running_cost_torque=1e-4,
                                       final_cost_state=1e10)
```

where weights contains the weights of the terminal and the running cost model. dt is the timestep length, T is the number of timesteps. running_cost_state, running_cost_torque and final_cost_state are the individal cost weights. The method returns the time trajectory T, the positiontrajectory TH, the velocity trajectory THD and the control trajectory U.

The trajectory can be plotted with:

```
ddp.plot_trajectory()
```

or simulated with gepetto with (for this enable_gui has to be set to True during the initialization):

```
ddp.simulate_trajectory_gepetto()
```

5.3.3 Usage

An example script for the pendulum can be found in the examples directory. It can be started with:

```
python compute_BOXFDDP_swingup.py
```

5.3.4 Dependencies

Crocoddyl

Pinocchio

For the display, Gepetto

5.3.5 References

[1] Mastalli, Carlos, et al. "Crocoddyl: An efficient and versatile framework for multi-contact optimal control." 2020 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2020. arxiv link

CHAPTER

SIX

REINFORCEMENT LEARNING

6.1 Soft Actor Critic Training

Note: Type: Closed loop, learning based, model free

State/action space constraints: None

Optimal: Yes

Versatility: Swing-up and stabilization

6.1.1 Theory

The soft actor critic (SAC) algorithm is a reinforcement learning (RL) method. It belongs to the class of so called 'model free' methods, i.e. no knowledge about the system to be controlled is assumed. Instead, the controller is trained via interaction with the system, such that a (sub-)optimal mapping from state space to control command is learned. The learning process is guided by a reward function that encodes the task, similar to the usage of cost functions in optimal control.

SAC has two defining features. Firstly, the mapping from state space to control command is probabilistic. Secondly, the entropy of the contol output is maximized along with the reward function during training. In theory, this leads to robust controllers and reduces the probability of ending up in suboptimal local minima.

For more information on SAC please refer to the original paper [1]:

6.1.2 API

The sac trainer can be initialized with:

```
trainer = sac_trainer(log_dir="log_data/sac_training")
```

During and after the training process the trainer will save logging data as well the best model in the directory provided via the `log_dir` parameter.

Before the training can be started the following three initialisations have to be made:

trainer.init_pendulum() trainer.init_environment() trainer.init_agent()

Warning: **Attention** Make sure to call these functions in this order as they build upon another.

The parameters of `init_pendulum` are:

- `mass`: mass of the pendulum
- `length`: length of the pendulum
- `inertia`: inertia of the pendulum
- `damping`: damping of the pendulum
- `coulomb_friction`: coulomb_friction of the pendulum
- `gravity`: gravity
- `torque_limit`: torque limit of the pendulum

The parameters of `init_environment` are:

- `dt`: timestep in seconds
- `integrator`: which integrator to use (`euler` or `runge_kutta`)
- `max_steps`: maximum number of timesteps the agent can take in one episode
- `reward_type`: Type of reward to use receive from the environment (`continuous`, `discrete`, `soft_binary`, `soft_binary_with_repellor` and `open_ai_gym`)
- `target`: the target state ([np.pi, 0] for swingup)
- `state_target_epsilon`: the region around the target which is considered as target
- `random_init`: How the pendulum is initialized in the beginning of each episode (`False`, `start_vicinity`, `everywhere`)
- `state_representation`: How to represent the state of the pendulum (should be 2 or 3). 2 for regular representation (position, velocity). 3 for trigonometric representation (cos(position), sin(position), velocity).

The parameters of `init_agent` are:

- `learning_rate`: learning_rate of the agent
- `warm_start`: whether to warm_start the agent
- `warm_start_path`: path to a model to load for warm starting
- `verbose`: Whether to print training information to the terminal

After these initialisations the training can be started with:

```
trainer.train(training_timesteps=1e6,
              reward_threshold=1000,
              eval_frequency=10000,
```

(continues on next page)

(continued from previous page)

```
            n_eval_episodes=20,
            verbose=1)
```

where the parameters are:

- `training_timesteps`: Number of timesteps to train
- `reward_threshold`: Validation threshold to stop training early
- `eval_frequency`: evaluate the model every `eval_frequency` timesteps
- `n_eval_episodes`: number of evaluation episodes
- `verbose`: Whether to print training information to the terminal

When finished the train method will save the best model in the `log_dir` of the trainer object.

Warning: **Attention**: when training is started, the log_dir will be deleted. So if you want to keep a trained model, move the saved files somewhere else.

The training progress can be observed with tensorboard. Start a new terminal and start the tensorboard with the correct path, e.g.:

```
$> tensorboard --logdir log_data/sac_training/tb_logs
```

The default reward function used during training is `soft_binary_with_repellor`

$$r = \exp -(\theta - \pi)^2/(2 * 0.25^2) - \exp -(\theta - 0)^2/(2 * 0.25^2)$$

This encourages moving away from the stable fixed point of the system at $\theta = 0$ and spending most time at the target, the unstable fixed point $\theta = \pi$. Different reward functions can be used. Novel reward funcitons can be implemented by modifying the *swingup_reward* method of the training environment with an appropriate *if* clause, and then selecting this reward function in the init_environment parameters under the key *'reward_type'*. The training enviroment is located in gym_environment

6.1.3 Usage

For an example of how to train a sac model see the train_sac.py script in the examples folder.

The trained model can be used with the sac controller.

6.1.4 Comments

Todo: comments on training convergence stability

6.1.5 Requirements

- Stable Baselines 3 (https://github.com/DLR-RM/stable-baselines3)
- Numpy
- PyYaml

6.1.6 References

[1] [Haarnoja, Tuomas, et al. “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor.” International conference on machine learning. PMLR, 2018.](https://arxiv.org/abs/1801.01290)

6.2 Deep Deterministic Policy Gradient Training

Note: Type: Closed loop, learning based, model free

State/action space constraints: None

Optimal: Yes

Versatility: Swing-up and stabilization

6.2.1 Theory

The Deep deterministic Policy Gradient (DDPG) algorithm is a reinforcement learning (RL) method. DDPG is a model-free off-policy algorithm. The controller is trained via interaction with the system, such that a (sub-)optimal mapping from state space to control command is learned. The learning process is guided by a reward function that encodes the task, similar to the usage of cost functions in optimal control.

DDPG can thought of Q-learning for continuous action spaces. It utilizes two networks, an actor and a critic. The actor receives a state and proposes an action. The critic assigns a value to a state action pair. The better an action suits a state the higher the value.

Further, DDPG makes use of target networks in order to stabilize the training. This means there are a training and a target version of the actor and critic models. The training version is used during training and the target networks are partially updated by polyak averaging:

$$\phi_{targ} = \tau\phi_{targ}(1-\tau)\phi_{train}$$

where τ is usually small.

DDPG also makes use of a replay buffer, which is a set of experiences which have been observed during training. The replay buffer should be large enough to contain a wide range of experiences. For more information on DDPG please refer to the original paper [1]:

This implementation losely follows the keras guide [2].

6.2.2 API

The ddpg trainer can be initialized with:

```
trainer = ddpg_trainer(batch_size=64,
                       validate_every=20,
                       validation_reps=10,
                       train_every_steps=1)
```

where the parameters are

- `batch_size`: number of samples to train on in one training step
- `validate_every`: evaluate the training progress every validate_every episodes
- `validation_reps`: number of episodes used during the evaluation
- `train_every_steps`: frequency of training compared to taking steps in the environment

Before the training can be started the following three initialisations have to be made:

```
trainer.init_pendulum()
trainer.init_environment()
trainer.init_agent()
```

> **Warning:** **Attention** Make sure to call these functions in this order as they build upon another.

The parameters of `init_pendulum` are:

- `mass`: mass of the pendulum
- `length`: length of the pendulum
- `inertia`: inertia of the pendulum
- `damping`: damping of the pendulum
- `coulomb_friction`: coulomb_friction of the pendulum
- `gravity`: gravity
- `torque_limit`: torque limit of the pendulum

The parameters of `init_environment` are:

- `dt`: timestep in seconds
- `integrator`: which integrator to use (“euler” or “runge_kutta”)

- `max_steps`: maximum number of timesteps the agent can take in one episode
- `reward_type`: Type of reward to use receive from the environment ("continuous", "discrete", "soft_binary", soft_binary_with_repellor" and "open_ai_gym")
- `target`: the target state ([np.pi, 0] for swingup)
- `state_target_epsilon`: the region around the target which is considered as target
- `random_init`: How the pendulum is initialized in the beginning of each episode ("False", "start_vicinity", "everywhere")
- `state_representation`: How to represent the state of the pendulum (should be 2 or 3). 2 for regular representation (position, velocity). 3 for trigonometric representation (cos(position), sin(position), velocity).
- `validation_limit`: Validation threshold to stop training early

The parameters of `init_agent` are:

- `replay_buffer_size`: The size of the replay_buffer
- `actor`: the tensorflow model to use as actor during training. If None, the default model is used
- `critic`: the tensorflow model to use as actor during training. If None, the default model is used
- `discount`: The discount factor to propagate the reward during one episode
- `actor_lr`: learning rate for the actor model
- `critic_lr`: learning rate for the critic model
- `tau`: determines how much of the training models is copied to the target models

After these initialisations the training can be started with:

```
trainer.train(n_episodes=1000,
              verbose=True)
```

where the parameters are:

- `n_episodes`: number of episodes to train
- `verbose`: Whether to print training information to the terminal

Afterwards the trained model can be saved with:

```
trainer.save("log_data/ddpg_training")
```

The save method will save the actor and critic model under the given path.

> **Warning:** **Attention**: This will delete existiong models at this location. So if you want to keep a trained model, move the saved files somewhere else.

The default reward function used during training is `open_ai_gym`

$$r = -(\theta - \pi)^2 - 0.1(\dot{\theta} - 0)^2 - 0.001u^{2"}$$

This encourages spending most time at the target (in the equation ([pi, 0]) with actions as small as possible. Different reward functions can be used by changing the reward type in init_environment. Novel reward functions can be implemented by modifying the *swingup_reward* method of the training environment with an appropriate *if* clause, and then selecting this reward function in the init_environment parameters under the key *'reward_type'*. The training enviroment is located in gym_environment

6.2.3 Usage

For an example of how to train a sac model see the train_ddpg.py script in the examples folder.

The trained model can be used with the ddpg controller.

6.2.4 Comments

Todo: comments on training convergence stability

6.2.5 Requirements

- Tensorflow 2.x

6.2.6 References

[1] Lillicrap, Timothy P., et al. "Continuous control with deep reinforcement learning." arXiv preprint arXiv:1509.02971 (2015).

[2] reras guide

CHAPTER

SEVEN

TRAJECTORY-BASED CONTROLLERS

7.1 Open Loop Control

Note: Type: Open loop control

State/action space contraints: -

Optimal: -

Versatility: -

7.1.1 Theory

This controller is designed to feed a precomputed trajectory in from of a csv file to the simulator or the real pendulum. Particulary, the controller can process trajectories that have been found with help of the trajectory optimization methods.

7.1.2 API

The controller needs pendulum parameters as input during initialization:

```
OpenLoopController.__init__(self, data_dict)
    inputs:
        data_dict: dictionary
            A dictionary containing a trajectory in the below specified format
```

The `data_dict` dictionary should have the entries:

```
data_dict["des_time_list"] : desired timesteps
data_dict["des_pos_list"] : desired positions
data_dict["des_vel_list"] : desired velocities
data_dict["des_tau_list"] : desired torques
```

The values are assumed to be in SI units, i.e. time in s, position in rad, velocity in rad/s, torque in Nm.

The control output $\mathbf{u}(\mathbf{x})$ can be obtained with the API of the abstract controller class:

```
OpenLoopController.get_control_output(mean_pos, mean_vel, meas_tau, meas_time)
    inputs:
        meas_pos: float, position of the pendulum
        meas_vel: float, velocity of the pendulum
        meas_tau: not used
        meas_time: not used
    returns:
        des_pos, des_vel, u
```

The function returns the desired position, desired velocity and a torque as specified in the csv file at the given index. The index counter is incremeted by 1 every time the `get_control_output` function is called.

7.1.3 Usage

Before using this controller, first a trajectory has to be defined/calculated and stored to csv file in a suitable format.

7.1.4 Comments

The controller will not scale the trajectory according to the time values in the csv file. All it does is return the rows of the file one by one with each call of `get_control_output`.

7.2 Proportional-Integral-Derivative (PID) Control

Note: Type: Closed loop control

State/action space contraints: -

Optimal: -

Versatility: -

7.2.1 Theory

This controller is designed to follow a precomputed trajectory from of a csv file to the simulator or the real pendulum. Particulary, the controller can process trajectories that have been found with help of the trajectory optimization methods.

The torque processed by the PID control terms via (with feed forward torque):

$$u(t) = \tau K_p e(t) K_i \int_0^t e(t')\mathrm{d}t' + K_d \frac{\mathrm{d}e(t)}{\mathrm{d}t}$$

where τ is the torque from the csv file and $e(t)$ is the position error at timestep t. Without feed forward torque, the torque from the precomputed trajectory file is omitted:

$$u(t) = K_p e(t) K_i \int_0^t e(t')\mathrm{d}t' K_d \frac{\mathrm{d}e(t)}{\mathrm{d}t}$$

7.2.2 API

The controller needs pendulum parameters as input during initialization:

```
PIDController.__init__(self, data_dict, Kp, Ki, Kd, use_feed_forward=True)
    inputs:
        data_dict: dictionary
            A dictionary containing a trajectory in the below specified format
        Kp : float
            proportional term,
            gain proportial to the position error
        Ki : float
            integral term,
            gain proportional to the integral
            of the position error
        Kd : float
            derivative term,
            gain proportional to the derivative of the position error
        use_feed_forward : bool
            whether to use the torque that is provided in the csv file
```

The `data_dict` dictionary should have the entries:

```
data_dict["des_time_list"] : desired timesteps
data_dict["des_pos_list"] : desired positions
data_dict["des_vel_list"] : desired velocities
data_dict["des_tau_list"] : desired torques
```

The values are assumed to be in SI units, i.e. time in s, position in rad, velocity in rad/s, torque in Nm.

The control output $\mathbf{u}(\mathbf{x})$ can be obtained with the API of the abstract controller class:

```
PIDController.get_control_output(mean_pos, mean_vel, meas_tau, meas_time)
    inputs:
        meas_pos: float, position of the pendulum
        meas_vel: float, velocity of the pendulum
        meas_tau: not used
        meas_time: not used
    returns:
        des_pos, des_vel, u
```

The function returns the desired position, desired velocity as specified in the csv file at the given index. The returned torque is processed by the PID controller as described in the theory section.

The index counter is incremeted by 1 every time the `get_control_output` function is called.

7.2.3 Usage

Before using this controller, first a trajectory has to be defined/calculated and stored to csv file in a suitable format.

7.2.4 Comments

The controller will not scale the trajectory according to the time values in the csv file. All it does is process the rows of the file one by one with each call of get_control_output.

7.3 Time-varying Linear Quadrativ Regulator (TVLQR)

Note: Type: Closed loop control

State/action space contraints: -

Optimal: -

Versatility: -

7.3.1 Theory

This controller is designed to follow a precomputed trajectory from of a csv file to the simulator or the real pendulum. Particulary, the controller can process trajectories that have been found with help of the trajectory optimization methods.

The Time-varying Linear Quadratic Regulator (TVLQR) is an extension to the regular LQR controller. The LQR formalization is used for a time-varying linear dynamics function

$$\dot{\mathbf{x}} = \mathbf{A}(t)\mathbf{x}\mathbf{B}(t)\mathbf{u}$$

The TVLQR controller tries to stabilize the system along a nominal trajectory. For this, at every timestep the system dynamics are linearized around the state of the nominal trajectory $\mathbf{x}_0(t), \mathbf{u}_0(t)$ at the given timestep t. The LQR formalism then can be used to derive the optimal controller at timestep t:

$$u(\mathbf{x}) = \mathbf{u}_0(t) - \mathbf{K}(t)\left(\mathbf{x} - \mathbf{x}_0(t)\right)$$

For further reading, we recommend chapter 8 of this Underactuated Robotics [1] lecture.

7.3.2 API

The controller needs pendulum parameters as input during initialization:

```
TVLQRController.__init__(self, data_dict, mass, length, damping, gravity,␣
↪torque_limit)
    inputs:
        data_dict: dictionary
            A dictionary containing a trajectory in the below specified format
        mass: float, default: 1.0
        length: float, default: 0.5
        damping: float, default: 0.1
        gravity: float, default: 9.81
        torque_limit: float, default: np.inf
```

The data_dict dictionary should have the entries:

```
data_dict["des_time_list"] : desired timesteps
data_dict["des_pos_list"] : desired positions
data_dict["des_vel_list"] : desired velocities
data_dict["des_tau_list"] : desired torques
```

The values are assumed to be in SI units, i.e. time in s, position in rad, velocity in rad/s, torque in Nm.

The control output $\mathbf{u}(\mathbf{x})$ can be obtained with the API of the abstract controller class:

```
TVLQRController.get_control_output(mean_pos, mean_vel, meas_tau, meas_time)
    inputs:
        meas_pos: float, position of the pendulum
        meas_vel: float, velocity of the pendulum
        meas_tau: not used
        meas_time: not used
    returns:
        des_pos, des_vel, u
```

The function returns the desired position, desired velocity as specified in the csv file at the given index. The returned torque is processed by the TVLQR controller as described in the theory section.

The index counter is incremeted by 1 every time the get_control_output function is called.

7.3.3 Usage

Before using this controller, first a trajectory has to be defined/calculated and stored to csv file or dictionary in a suitable format.

7.3.4 Dependencies

The trajectory optimization using direct collocation for the pendulum swing-up is accomplished by taking advantage of Drake toolbox [2].

7.3.5 Comments

The controller will not scale the trajectory according to the time values in the csv file. All it does is process the rows of the file one by one with each call of get_control_output.

7.3.6 References

[1] Russ Tedrake. Underactuated Robotics: Algorithms for Walking, Running, Swimming, Flying, and Manipulation (Course Notes for MIT 6.832).

[2] Model-Based Design and Verification for Robotics.

7.4 iLQR Model Predictive Control

Note: Type: Model Predictive Control

State/action space contraints: No

Optimal: Yes

Versatility: Swingup and stabilization

7.4.1 Theory

This controller uses the trajectory optimization from the iLQR algorithm (see iLQR) in an MPC setting. This means that this controller recomputes an optimal trajectory (including the optimal sequence of control inputs) at every time step. The first control input of this solution is returned as control input for the current state. As the optimization happens every timestep the iLQR algorithm is only executed with one forward and one backward pass. As initial trajectory the solution of the previous timestep is parsed to the iLQR solver.

7.4.2 Requirements

pydrake (see getting_started)

7.4.3 API

The controller can be initialized as:

```
controller = iLQRMPCController(mass=0.5,
                               length=0.5,
                               damping=0.1,
                               coulomb_friction=0.0,
                               gravity=9.81,
                               x0=[0.0,0.0],
                               dt=0.02,
                               N=50,
                               max_iter=1,
                               break_cost_redu=1e-1,
                               sCu=30.0,
                               sCp=0.001,
                               sCv=0.001,
                               sCen=0.0,
                               fCp=100.0,
                               fCv=1.0,
                               fCen=100.0,
                               dynamics="runge_kutta",
                               n_x=n2)
```

where

- `mass`, length, damping, coulomb_friction, gravity are the pendulum parameters (see PendulumPlant)
- `x0`: array like, start state
- `dt`: float, time step
- `N`: int, time steps the controller plans ahead
- `max_iter`: int, number of optimization loops
- `break_cost_redu`: cost at which the optimization stops
- `sCu`: float, stage cost coefficient penalizing the control input every step
- `sCp`: float, stage cost coefficient penalizing the position error every step
- `sCv`: float, stage cost coefficient penalizing the velocity error every step
- `sCen`: float, stage cost coefficient penalizing the energy error every step
- `fCp`: float, final cost coefficient penalizing the position error at the final state
- `fCv`: float, final cost coefficient penalizing the velocity error at the final state
- `fCen`: float, final cost coefficient penalizing the energy error at the final state

- `dynamics`: string, “euler” for euler integrator, “runge_kutta” for Runge-Kutta integrator
- `nx`: int, nx=2, or n_x=3 for pendulum, n_x=2 uses $[\theta, \dot{\theta}]$ as pendulum state during the optimization, n_x=3 uses $[\cos(\theta), \sin(\theta), \dot{\theta}]$ as state

Before using the controller a goal has to be set via:

```
controller.set_goal(goal=[np.pi, 0])
```

which initializes the cost function derivatives inside the controller for the specified goal.

It is possible to either load an initial guess for the trajectory from a csv file by using:

```
controller.load_initial_guess(filepath="../../../../data/trajectories/iLQR/
↪trajectory.csv"
```

for example a trajectory that has been found with the offline trajectory optimization iLQR. Alternatively, it is possible to compute a new initial guess with:

```
controller.compute_initial_guess(N=50)
```

With an initial guess set the control output can be obtained with the standard controller api:

```
controller.get_control_output(meas_pos, meas_vel,
                              meas_tau=0, meas_time=0):
```

where only the measured position (meas_pos) and measured velocity (meas_vel) are used in the control loop. The function returns

- None, None, u

`get_control_output` returns None for the desired position and desired velocity (the iLQR controller is a pure torque controller). `u` is the first control input of the computed control sequence. as described in the Theory section.

7.4.4 Usage

The controller can be tested in simulation with:

```
python sim_ilqrMPC.py
```

7.4.5 Comments

For coulomb_fricitons != 0 the optimization gets considerably slower.

CHAPTER

EIGHT

POLICY-BASED CONTROLLERS

8.1 Gravity Compensation Control

Note: Type: Closed loop control

State/action space constraints: -

Optimal: -

Versatility: only compensates for gravitational force acting on the pendulum, no swing-up or stabilization at the upright position

8.1.1 Theory

A controller compensating the gravitational force acting on the pendulum. The control function is given by:

$$u(\theta) = mgl\sin(\theta)$$

where *u* is commanded torque, *m* is a weight of *0,5 kg* attached to the rod together with the mass of the rod and the mounting parts, *l* is the length of *0,5 m* of the rod, *g* is gravitational acceleration on earth of $9.81\ ms^{-2}$ and θ is the current position of the pendulum.

While the controller is running it actively compensates for the gravitational force acting on the pendulum, therefore the pendulum can be moved as if it was in zero-g.

8.2 Energy Shaping Control

Note: Type: Closed loop control

State/action space constraints: No

Optimal: No

Versatility: Swingup only, additional stabilization needed at the upright unstable fixed point via LQR controller

8.2.1 Theory

The energy of a simple pendulum in state $x = [\theta, \dot{\theta}]$ is given by:

$$E(\theta, \dot{\theta}) = \frac{1}{2} m l^2 \dot{\theta}^2 - mgl\cos(\theta)$$

where the first term is the kinetic energy of the system and the second term is the potential energy. In the standing upright position *[pi, 0]* the whole energy of the pendulum is potential energy and the kinetic energy is zero. As that is the goal state of a swingup motion, the desired energy can be defined as the energy of that state:

$$E_{des} = mgl$$

The idea behind energy-shaping control is simple:

- If $E < E_{des}$, the controller adds energy to the system by outputting torque in the direction of motion.
- If $E > E_{des}$, the controller subtracts energy from the system by outputting torque in the opposite direction of the direction of motion.

Consequently, the control function reads:

$$u(\theta, \dot{\theta}) = -k\dot{\theta}\left(E(\theta, \dot{\theta}) - E_{des}\right), \quad kgt; 0$$

This controller is applicable in the whole state space of the pendulum, i.e. it will always push the system towards the upright position. Note however, that the controller does not stabilize the upright position! If the pendulum overshoots the unstable fixpoint, the controller will make the pendulum turn another round.

In order to stabilize the pendulum at the unstable fixpoint, energy-shaping control can be combined with a stabilizing controller such as the LQR controller.

8.2.2 API

The controller needs pendulum parameters as well as the control term `k` (see equation (3)) as input during initialization:

```
EnergyShapingController.__init__(self, mass=1.0, length=0.5, damping=0.1,␣
↪gravity=9.81, k=1.0)
    inputs:
        mass: float, default: 1.0
        length: float, default: 0.5
        damping: float, default: 0.1
        gravity: float, default: 9.81
        k: float, default: 1.0
```

Before using the controller, the function `EnergyShapingController.set_goal` must be called with input:

```
EnergyShapingController.set_goal(x)
    inputs:
        x: list of length 2
```

where `x` is the desired goal state. This function sets the desired energy for the output calculation.

The control output $\mathbf{u}(\mathbf{x})$ can be obtained with the API of the abstract controller class:

```
EnergyShapingController.get_control_output(mean_pos, mean_vel, meas_tau, meas_
↪time)
    inputs:
        meas_pos: float, position of the pendulum
        meas_vel: float, velocity of the pendulum
        meas_tau: not used
        meas_time: not used
    returns:
        None, None, u
```

`get_control_output` returns `None` for the desired position and desired velocity (the energy shaping controller is a pure torque controller). The returned torque u is the result of equation (3).

8.2.3 Usage

A usage example can be found in the examples folder. Start a simulation with energy-shaping control for pendulum swingup and lqr control stabilization at the unstable fixpoint:

```
python sim_energy_shaping.py
```

8.3 LQR Control

Note: Type: Closed loop control

State/action space contraints: No

Optimal: Yes

Versatility: Stabilization only

8.3.1 Theory

A linear quadratic regulator (LQR) can be used to stabilize the pendulum at the unstable fixpoint. For a linear system of the form

$$\dot{\mathbf{x}} = \mathbf{AxBu}$$

and a infinite horizon cost function in quadratic form:

$$J = \int_0^\infty \left(\mathbf{x}^T\mathbf{Qxu}^T\mathbf{Ru}\right) \mathrm{d}t, \quad \mathbf{Q} = \mathbf{Q} \succeq 0, \mathbf{R} = \mathbf{R} \succeq 0$$

the (provably) optimal controller is

$$u(\mathbf{x}) = -\mathbf{R}^{-1}\mathbf{B}^T\mathbf{S}\mathbf{x} = -\mathbf{K}\mathbf{x}$$

where $\mathbf{S}$ has to fulfill the algebraic Riccati equation

$$\mathbf{S}\mathbf{A}\mathbf{A}^T\mathbf{S} - \mathbf{S}\mathbf{B}\mathbf{R}^{-1}\mathbf{B}\mathbf{S}\mathbf{Q} = 0$$

There are many solvers for the algebraic Riccati equation. In this library the solver from the scipy package is used.

8.3.2 API

The controller needs pendulum parameters as input during initialization:

```
LQRController.__init__(self, mass=1.0, length=0.5, damping=0.1, gravity=9.81,␣
↪torque_limit=np.inf)
    inputs:
        mass: float, default: 1.0
        length: float, default: 0.5
        damping: float, default: 0.1
        gravity: float, default: 9.81
        torque_limit: float, default: np.inf
```

The control output $\mathbf{u}(\mathbf{x})$ can be obtained with the API of the abstract controller class:

```
LQRController.get_control_output(mean_pos, mean_vel, meas_tau, meas_time)
    inputs:
        meas_pos: float, position of the pendulum
        meas_vel: float, velocity of the pendulum
        meas_tau: not used
        meas_time: not used
    returns:
        None, None, u
```

`get_control_output` returns None for the desired position and desired velocity (the LQR controller is a pure torque controller). The returned torque u is the result of equation (3). If the calculated torque is out of bounds of the pendulum's torque limits the controller will return u=None as torque.

8.3.3 Usage

A usage example can be found in the examples folder. The controller can be tested in simulation with:

```
python sim_lqr.py
```

Robotics Innovation Center
Robert-Hooke-Str. 1
28359 Bremen
Deutschland
robotik@dfki.de

8.3.4 Comments

Without torque limits the LQR controller can drive the pendulum up from any position in a straight way. In practice this controller should only be used for stabilizing the pendulum at the unstable fixpoint. If the controller would require a torque larger than the pendulums torque limit, the controller returns None instead. This makes it possible to combine this controller with another controller and only use the LQR control if the output is not None. The region of attraction where this controller is able to stabilize the pendulum depends on the pendulum parameters and especially its torque limits.

CHAPTER

NINE

CONTROLLER ANALYSIS

9.1 Controller Benchmarking

The controller benchmarking class can benchmark the controllers in simulation with respect to a predefined properties.

9.1.1 Definitions

The controller benchmark currently computes the following properties

- **Controller Frequency**: How fast can the controller process a pendulum state and return a control output. Measured in calls per second (Hz).
- **Swingup time**: How long does it take for the controller to swing-up the pendulum from the lower fixpoint to the upper fixpoint. Units: Seconds (s)
- **Energy consumption**: How much energy does the controller use during the swingup motion and holding the pendulum stable afterwards. Energy usage is measured by comparing the energy level of the actuated pendulum with a free falling pendulum. Units: Joule (J).
- **Smoothness** measures how much the controller changes the control output during execution. The calculated value is the standard deviation of the differences of all consecutive control signals.
- **Consistency** measures if the controller is able to drive the pendulum to the unstable fixpoint for varying starting positions and velocities. The start position is randomly chosen between [-pi, pi] and the velocity is drawn from the intervall [-3pi/s, 3pi/s].
- **Robustness** tests the controller abilities to recover from perturbations during the swingup motions. The controller is perturbed four times for 1 entire second with a random amount of torque.
- **Sensitivity**: Here the pendulum parameters (mass, length, friction) are modified without using this knowledge in the controller. This tests how sensitive the controller is to the model parameters.
- **Reduced torque limit**: This test checks a list of torque limits to find the minimal torque limit with which the controller is still able to swing-up the pendulum.

9.1.2 API

To initialize the benchmark class do:

```
from simple_pendulum.analysis.benchmark import benchmarker

ben = benchmarker(dt=dt,
                  max_time=max_time,
                  integrator=integrator,
                  benchmark_iterations=benchmark_iterations)
```

where dt is the control frequency, max_time the time of a single motion, integrator the integrator to be used ("euler" or "runge_kutta", see simulator) and benchmark_iterations is the number iterations that are used to do the benchmark test.

The parameters of the pendulum can be parsed to the benchmark class with:

```
ben.init_pendulum(mass=mass,
                  length=length,
                  inertia=inertia,
                  damping=damping,
                  coulomb_friction=coulomb_fric,
                  gravity=gravity,
                  torque_limit=torque_limit)
```

The controller to be tested has to be set by:

```
ben.set_controller(controller)
```

where controller is a controller inheriting from the abstract controller class.

The benmark calculations are then started with:

```
ben.benchmark(check_speed=True,
              check_energy=True,
              check_time=True,
              check_smoothness=True,
              check_consistency=True,
              check_robustness=True,
              check_sensitivity=True,
              check_torque_limit=True,
              save_path="benchmark.yml")
```

The individual checks can be turned off. The results will be stored in the filed specified in save_path.

9.1.3 Usage

An example usage can be found in the [examples folder](https://github.com/dfki-ric-underactuated-lab/torque_limited_simple_pendulum/tree/master/software/python/examples) in the [benchmark_controller.py](https://github.com/dfki-ric-underactuated-lab/torque_limited_simple_pendulum/blob/master/software/python/examples/benchmark_controller.py) script.

9.2 Plotting Controllers

Controllers can be plotted by plotting their control signal in the pendulum's state space.

9.2.1 API

A controller inheriting from the *abstract controller class* <https://github.com/dfki-ric-underactuated-lab/torque_limited_simple_pendulum/blob/master/software/python/simple_pendulum/controllers/abstract_controller.py>`_ can be plotted by using:

```
from simple_pendulum.analysis.plot_policy import plot_policy

plot_policy(controller,
            position_range=[-3.14, 3.14],
            velocity_range=[-2. 2],
            samples_per_dim=100,
            plotstyle="3d",
            save_path=None)
```

The plotstyle can also be set to "2d". If a save_path is specified the plot will be stored in that location.

9.2.2 Usage

An example usage can be found in the *examples folder* <https://github.com/dfki-ric-underactuated-lab/torque_limited_simple_pendulum/tree/master/software/python/examples) in the [plot_controller.py](https://github.com/dfki-ric-underactuated-lab/torque_limited_simple_pendulum/blob/master/software/python/examples/plot_controller.py>`_ script.

CHAPTER

TEN

SIMULATOR

The simulator class can simulate and animate the pendulum motion forward in time. The gym environment can be used for reinforcement learning.

10.1 API

10.1.1 The simulator

The simulator should by initialized with a plant (here the PendulumPlant) as follows:

```
pendulum = PendulumPlant()
sim = Simulator(plant=pendulum)
```

To simulate the dynamics of the plant forward in time call:

```
T, X, TAU = sim.simulate(t0=0.0,
                         x0=[0.5, 0.0],
                         tf=10.0,
                         dt=0.01,
                         controller=None,
                         integrator="runge_kutta")
```

The inputs of the function are:

- t0: float, start time, unit: s
- x0: start state (dimension as the plant expects it)
- tf: float, final time, unit: s
- dt: float, time step, unit: s
- controller: controller that computes the motor torque(s) to be applied. The controller should have the structure of the AbstractController class in utilities/abstract_controller. If controller=None, no controller is used and the free system is simulated.
- **integrator: string, "euler" for euler integrator,** "runge_kutta" for Runge-Kutta integrator

The function returns three lists:

- T: List of time values
- X: List of states
- TAU: List of actuations

The same simulation can be executed together with an animation of the plant (only implemented for 2d serial chains). For the simuation with animation call:

```
T, X, TAU = sim.simulate_and_animate(t0=0.0,
                                     x0=[0.5, 0.0],
                                     tf=10.0,
                                     dt=0.01,
                                     controller=None,
                                     integrator="runge_kutta",
                                     phase_plot=True,
                                     save_video=False,
                                     video_name="")
```

The additional parameters are:

- phase plot: bool, whether to show a phase plot along the animation plot
- save_video: bool, whether to save the animation as mp4 video
- video_name: string, name of the file where the video should be saved (only used if save_video=True)

10.1.2 The gym environment

The environment can be initialized with:

```
pendulum = PendulumPlant()
sim = Simulator(plant=pendulum)
env = SimplePendulumEnv(simulator=sim,
                        max_steps=5000,
                        target=[np.pi, 0.0],
                        state_target_epsilon=[1e-2, 1e-2],
                        reward_type='continuous',
                        dt=1e-3,
                        integrator='runge_kutta',
                        state_representation=2,
                        validation_limit=-150,
                        scale_action=True,
                        random_init="False")
```

The parameters are:

- simulator : simulator object
- max_steps : int, default=5000, maximum steps the agent can take before the episode is terminated
- target : array-like, default=[np.pi, 0.0], the target state of the pendulum
- state_target_epsilon: array-like, default=[1e-2, 1e-2], target epsilon for discrete reward type

- **reward_type** [string, default='continuous', the reward type selects the reward function which is used] options: 'continuous', 'discrete', 'soft_binary', 'soft_binary_with_repellor'
- dt : float, default=1e-3, timestep for the simulation
- **integrator** [string, default='runge_kutta', the integrator which is used by the simulator] options : 'euler', 'runge_kutta'
- **state_representation** [int, default=2, determines how the state space of the pendulum is represented] 2 means state = [position, velocity] 3 means state = [cos(position), sin(position), velocity]
- validation_limit : float, default=-150, If the reward during validation episodes surpasses this value the training stops early
- scale_action : bool, default=True, whether to scale the output of the model with the torque limit of the simulator's plant. If True the model is expected so return values in the intervall [-1, 1] as action.
- **random_init** [string, default="False",] A string determining the random state initialisation "False" : The pendulum is set to [0, 0], "start_vicinity" : The pendulum position and velocity

  are set in the range [-0.31, -0.31],

  "everywhere" [The pendulum is set to a random state in the whole] possible state space

10.2 Usage

For examples of usages of the simulator class check out the scripts in the examples folder <https://github.com/dfki-ric-underactuated-lab/torque_limited_simple_pendulum/tree/master/software/python/examples>`_.

The gym environment is used for example in the ddpg training.

CHAPTER

ELEVEN

API REFERENCE

11.1 Analysis

modify_pendulum_parameter(*par*)

class benchmarker(*dt=0.01*, *max_time=10.0*, *integrator='runge_kutta'*, *benchmark_iterations=10*)

Bases: `object`

init_pendulum(*mass=0.57288*, *length=0.5*, *inertia=None*, *damping=0.15*, *coulomb_friction=0.0*, *gravity=9.81*, *torque_limit=2.0*)

Initialize the pendulum parameters.

mass [float, default=0.57288] mass of the pendulum [kg]

length [float, default=0.5] length of the pendulum [m]

inertia [float, default=None] inertia of the pendulum [kg m^2] defaults to point mass inertia (mass*length^2)

damping [float, default=0.15] damping factor of the pendulum [kg m/s]

coulomb_friction [float, default=0.0] coulomb friciton of the pendulum [Nm]

gravity [float, default=9.81] gravity (positive direction points down) [m/s^2]

torque_limit [float, default=2.0] the torque_limit of the pendulum actuator

set_controller(*controller*)

check_regular_execution()

check_consistency()

check_robustness()

check_sensitivity()

check_reduced_torque_limit(*tl=inf*)

check_speed(*N=1000*)

benchmark(*check_speed=True*, *check_energy=True*, *check_time=True*, *check_smoothness=True*, *check_consistency=True*, *check_robustness=True*, *check_sensitivity=True*, *check_torque_limit=True*, *save_path=None*)

plot_policy(*controller*, *position_range=[- 3.141592653589793, 3.141592653589793]*, *velocity_range=[-8, 8]*, *samples_per_dim=100*, *plotstyle='2d'*, *save_path=None*)

11.2 Controllers

class AbstractController

Bases: abc.ABC

Abstract controller class. All controller should inherit from this abstract class.

abstract get_control_output(*meas_pos*, *meas_vel*, *meas_tau*, *meas_time*)

The function to compute the control input for the pendulum actuator. Supposed to be overwritten by actual controllers. The API of this method should be adapted. Unused inputs/outputs can be set to None.

meas_pos [float] the position of the pendulum [rad]

meas_vel [float] the velocity of the pendulum [rad/s]

meas_tau [float] the meastured torque of the pendulum [Nm]

meas_time [float] the collapsed time [s]

des_pos [float] the desired position of the pendulum [rad]

des_vel [float] the desired velocity of the pendulum [rad/s]

des_tau [float] the torque supposed to be applied by the actuator [Nm]

init(*x0*)

Initialize the controller. May not be necessary.

x0 [array like] the start state of the pendulum

set_goal(*x*)

Set the desired state for the controller. May not be necessary.

x [array like] the desired goal state of the controller

ak80_6(*control_method*, *name*, *attribute*, *params*, *data_dict*, *motor_id='0x01'*, *can_port='can0'*)

11.2.1 Deep Deterministic Policy Gradient Control

class ddpg_controller(*model_path*, *torque_limit*, *state_representation=3*)

Bases: *simple_pendulum.controllers.abstract_controller.AbstractController*

get_control_output(*meas_pos*, *meas_vel*, *meas_tau=0*, *meas_time=0*)

The function to compute the control input for the pendulum actuator. Supposed to be overwritten by actual controllers. The API of this method should be adapted. Unused inputs/outputs can be set to None.

meas_pos [float] the position of the pendulum [rad]

meas_vel [float] the velocity of the pendulum [rad/s]

meas_tau [float] the meastured torque of the pendulum [Nm]

meas_time [float] the collapsed time [s]

des_pos [float] the desired position of the pendulum [rad]

des_vel [float] the desired velocity of the pendulum [rad/s]

des_tau [float] the torque supposed to be applied by the actuator [Nm]

get_observation(*state*)

11.2.2 Energy Shaping Control

class EnergyShapingController(*mass=1.0*, *length=0.5*, *damping=0.1*, *gravity=9.81*, *torque_limit=2.0*, *k=1.0*)

Bases: *simple_pendulum.controllers.abstract_controller.AbstractController*

Controller which swings up the pendulum by regulating its energy.

set_goal(*x*)

Set the goal for the controller. This function calculates the energy of the goal state.

x [array-like] the goal state for the pendulum

get_control_output(*meas_pos*, *meas_vel*, *meas_tau=0*, *meas_time=0*)

The function to compute the control input for the pendulum actuator

meas_pos [float] the position of the pendulum [rad]

meas_vel [float] the velocity of the pendulum [rad/s]

meas_tau [float] the meastured torque of the pendulum [Nm] (not used)

meas_time [float] the collapsed time [s] (not used)

des_pos [float] the desired position of the pendulum [rad] (not used, returns None)

des_vel [float] the desired velocity of the pendulum [rad/s] (not used, returns None)

des_tau [float] the torque supposed to be applied by the actuator [Nm]

class EnergyShapingAndLQRController(*mass=1.0*, *length=0.5*, *damping=0.1*, *gravity=9.81*, *torque_limit=inf*, *k=1.0*)

Bases: *simple_pendulum.controllers.abstract_controller.AbstractController*

Controller which swings up the pendulum with the energy shaping controller and stabilizes the pendulum with the lqr controller.

set_goal(*x*)
Set the goal for the controller.

x [array-like] the goal state for the pendulum

get_control_output(*meas_pos*, *meas_vel*, *meas_tau=0*, *meas_time=0*, *verbose=False*)
The function to compute the control input for the pendulum actuator

meas_pos [float] the position of the pendulum [rad]

meas_vel [float] the velocity of the pendulum [rad/s]

meas_tau [float] the meastured torque of the pendulum [Nm] (not used)

meas_time [float] the collapsed time [s] (not used)

verbose [bool, default=False] whether to print when the controller switches between energy shaping and lqr

des_pos [float] the desired position of the pendulum [rad] (not used, returns None)

des_vel [float] the desired velocity of the pendulum [rad/s] (not used, returns None)

des_tau [float] the torque supposed to be applied by the actuator [Nm]

class Test(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

epsilon = 0.2

test_0_energy_shaping_swingup()

11.2.3 Gravity Compensation Control

class GravityCompController(*params*)
Bases: *simple_pendulum.controllers.abstract_controller.AbstractController*

get_control_output(*meas_pos*, *meas_vel*, *meas_tau*, *meas_time*)
The function to compute the control input for the pendulum actuator. Supposed to be overwritten by actual controllers. The API of this method should be adapted. Unused inputs/outputs can be set to None.

meas_pos [float] the position of the pendulum [rad]

meas_vel [float] the velocity of the pendulum [rad/s]

meas_tau [float] the meastured torque of the pendulum [Nm]

meas_time [float] the collapsed time [s]

des_pos [float] the desired position of the pendulum [rad]

des_vel [float] the desired velocity of the pendulum [rad/s]

des_tau [float] the torque supposed to be applied by the actuator [Nm]

11.2.4 iLQR Model Predictive Control

class iLQRMPCController(*mass=0.5*, *length=0.5*, *damping=0.15*, *coulomb_friction=0.0*, *gravity=9.81*, *inertia=0.125*, *dt=0.01*, *n=50*, *max_iter=1*, *break_cost_redu=1e-06*, *sCu=10.0*, *sCp=0.001*, *sCv=0.001*, *sCen=0.0*, *fCp=1000.0*, *fCv=10.0*, *fCen=300.0*, *dynamics='runge_kutta'*, *n_x=3*)

Bases: *simple_pendulum.controllers.abstract_controller.AbstractController*

Controller which computes an ilqr solution at every timestep and uses the first control output.

init(*x0*)

Initialize the controller. May not be necessary.

x0 [array like] the start state of the pendulum

load_initial_guess(*filepath='Pendulum_data/trajectory.csv'*, *verbose=True*)

load initial guess trajectory from file

filepath [string, default="Pendulum_data/trajectory.csv"] path to the csv file containing the initial guess for the trajectory

verbose [bool, default=True] whether to print from where the initial guess is loaded

set_initial_guess(*u_trj=None*, *x_trj=None*)

set initial guess from array like object

u_trj [array-like, default=None] initial guess for control inputs u ignored if u_trj==None

x_trj [array-like, default=None] initial guess for state space trajectory ignored if x_trj==None

compute_initial_guess(*N=None*, *verbose=True*)

compute initial guess

N [int, default=None] number of timesteps to plan ahead if N==None, N defaults to the number of timesteps that is also used during the online optimization (n in the class __init__)

verbose [bool, default=True] whether to print when the initial guess calculation is finished

set_goal(*x*)

Set a goal for the controller. Initializes the cost functions.

x [array-like] goal state for the pendulum

get_control_output(*meas_pos*, *meas_vel*, *meas_tau=0*, *meas_time=0*)

The function to compute the control input for the pendulum actuator

meas_pos [float] the position of the pendulum [rad]

meas_vel [float] the velocity of the pendulum [rad/s]

meas_tau [float, default=0] the meastured torque of the pendulum [Nm] (not used)

meas_time [float, default=0] the collapsed time [s] (not used)

des_pos [float] the desired position of the pendulum [rad] (not used, returns None)

des_vel [float] the desired velocity of the pendulum [rad/s] (not used, returns None)

des_tau [float] the torque supposed to be applied by the actuator [Nm]

class Test(*methodName='runTest'*)
Bases: `unittest.case.TestCase`

epsilon = 0.2

test_0_iLQR_MPC_swingup_nx2()

test_1_iLQR_MPC_swingup_nx3()

11.2.5 Linear Quadratic Regulator (LQR)

lqr(*A*, *B*, *Q*, *R*)
Solve the continuous time lqr controller. dx/dt = A x + B u cost = integral x.T*Q*x + u.T*R*u ref: Bertsekas, p.151

dlqr(*A*, *B*, *Q*, *R*)
Solve the discrete time lqr controller. x[k+1] = A x[k] + B u[k] cost = sum x[k].T*Q*x[k] + u[k].T*R*u[k] ref: Bertsekas, p.151

class LQRController(*mass=1.0*, *length=0.5*, *damping=0.1*, *gravity=9.81*, *torque_limit=inf*)
Bases: *simple_pendulum.controllers.abstract_controller.AbstractController*

Controller which stabilizes the pendulum at its instable fixpoint.

set_goal(*x*)
Set the desired state for the controller. May not be necessary.

x [array like] the desired goal state of the controller

get_control_output(*meas_pos*, *meas_vel*, *meas_tau=0*, *meas_time=0*)
The function to compute the control input for the pendulum actuator

meas_pos [float] the position of the pendulum [rad]

meas_vel [float] the velocity of the pendulum [rad/s]

meas_tau [float, default=0] the meastured torque of the pendulum [Nm] (not used)

meas_time [float, default=0] the collapsed time [s] (not used)

des_pos [float] the desired position of the pendulum [rad] (not used, returns None)

des_vel [float] the desired velocity of the pendulum [rad/s] (not used, returns None)

des_tau [float] the torque supposed to be applied by the actuator [Nm]

class Test(*methodName='runTest'*)

Bases: unittest.case.TestCase

epsilon = 0.01

test_0_LQR_stabilization()

11.2.6 Open Loop Control

class OpenLoopController(*data_dict*)

Bases: *simple_pendulum.controllers.abstract_controller.AbstractController*

Controller acts on a predefined trajectory.

init(*x0*)

Initialize the controller. May not be necessary.

x0 [array like] the start state of the pendulum

set_goal(*x*)

Set the desired state for the controller. May not be necessary.

x [array like] the desired goal state of the controller

get_control_output(*meas_pos=None*, *meas_vel=None*, *meas_tau=None*, *meas_time=None*)

The function to read and send the entries of the loaded trajectory as control input to the simulator/real pendulum.

meas_pos [float, deault=None] the position of the pendulum [rad]

meas_vel [float, deault=None] the velocity of the pendulum [rad/s]

meas_tau [float, deault=None] the meastured torque of the pendulum [Nm]

meas_time [float, deault=None] the collapsed time [s]

des_pos [float] the desired position of the pendulum [rad]

des_vel [float] the desired velocity of the pendulum [rad/s]

des_tau [float] the torque supposed to be applied by the actuator [Nm]

class OpenLoopAndLQRController(*data_dict*, *mass=1.0*, *length=0.5*, *damping=0.1*, *gravity=9.81*, *torque_limit=inf*)

Bases: *simple_pendulum.controllers.abstract_controller.AbstractController*

init(*x0*)

Initialize the controller. May not be necessary.

x0 [array like] the start state of the pendulum

set_goal(*x*)

Set the desired state for the controller. May not be necessary.

x [array like] the desired goal state of the controller

get_control_output (*meas_pos*, *meas_vel*, *meas_tau=0*, *meas_time=0*, *verbose=False*)
The function to read and send the entries of the loaded trajectory as control input to the simulator/real pendulum. Switches to lqr control when in its region of attraction.

meas_pos [float] the position of the pendulum [rad]

meas_vel [float] the velocity of the pendulum [rad/s]

meas_tau [float, deault=0] the meastured torque of the pendulum [Nm]

meas_time [float, deault=0] the collapsed time [s]

verbose [bool, default=False] whether to print when the controller switches between preloaded trajectory and lqr

des_pos [float] the desired position of the pendulum [rad]

des_vel [float] the desired velocity of the pendulum [rad/s]

des_tau [float] the torque supposed to be applied by the actuator [Nm]

11.2.7 Proportional-Integral-Derivative (PID) Control

class PIDController (*data_dict*, *Kp*, *Ki*, *Kd*, *use_feed_forward=True*)
Bases: *simple_pendulum.controllers.abstract_controller.AbstractController*

Controller acts on a predefined trajectory and adds PID control gains.

init (*x0*)
Initialize the controller. May not be necessary.

x0 [array like] the start state of the pendulum

set_goal (*x*)
Set the desired state for the controller. May not be necessary.

x [array like] the desired goal state of the controller

get_control_output (*meas_pos=None*, *meas_vel=None*, *meas_tau=None*, *meas_time=None*)
The function to read and send the entries of the loaded trajectory as control input to the simulator/real pendulum.

meas_pos [float, deault=None] the position of the pendulum [rad]

meas_vel [float, deault=None] the velocity of the pendulum [rad/s]

meas_tau [float, deault=None] the meastured torque of the pendulum [Nm]

meas_time [float, deault=None] the collapsed time [s]

des_pos [float] the desired position of the pendulum [rad]

des_vel [float] the desired velocity of the pendulum [rad/s]

des_tau [float] the torque supposed to be applied by the actuator [Nm]

11.2.8 Soft Actor Critic (SAC) Control

class SacController(*model_path*, *torque_limit*, *use_symmetry=True*, *state_representation=2*)

Bases: *simple_pendulum.controllers.abstract_controller.AbstractController*

Controller which acts on a policy which has been learned with sac.

get_control_output(*meas_pos*, *meas_vel*, *meas_tau=0*, *meas_time=0*)

The function to compute the control input for the pendulum actuator

meas_pos [float] the position of the pendulum [rad]

meas_vel [float] the velocity of the pendulum [rad/s]

meas_tau [float] the meastured torque of the pendulum [Nm] (not used)

meas_time [float] the collapsed time [s] (not used)

des_pos [float] the desired position of the pendulum [rad] (not used, returns None)

des_vel [float] the desired velocity of the pendulum [rad/s] (not used, returns None)

des_tau [float] the torque supposed to be applied by the actuator [Nm]

get_observation(*state*)

11.2.9 Time-varying Linear Quadratic Regulator (TVLQR)

11.3 Model

get_params(*params_path*)

class Environment

Bases: object

Environmental parameters

class Robot

Bases: object

Robot parameters

class Joints

Bases: object

Joint parameters

class Links

Bases: `object`

Link parameters

calc_m_l (*mass*, *mass_p*)

calc_length_com (*mass_p*, *mass_l*, *length*)

calc_inertia (*mass_p*, *mass_l*, *length*)

calc_inertia_com (*mass_p*, *mass_l*, *length*)

class Actuators

Bases: `object`

Motor parameters

calc_k_m (*k_t*, *resist*)

calc_k_v (*v_per_hz*)

calc_k_e (*k_v*)

calc_k_t_from_k_m (*k_m*, *resist*)

calc_k_t_from_k_v (*k_v*)

class PendulumPlant (*mass=1.0*, *length=0.5*, *damping=0.1*, *gravity=9.81*, *coulomb_fric=0.0*, *inertia=None*, *torque_limit=inf*)

Bases: `object`

load_params_from_file (*filepath*)

Load the pendulum parameters from a yaml file.

filepath [string] path to yaml file

forward_kinematics (*pos*)

Computes the forward kinematics.

pos : float, angle of the pendulum

list [A list containing one list (for one end-effector)] The inner list contains the x and y coordinates for the end-effector of the pendulum

inverse_kinematics (*ee_pos*)

Comutes inverse kinematics

ee_pos [array like,] len(state)=2 contains the x and y position of the end_effector floats, units: m

pos [float] angle of the pendulum, unit: rad

forward_dynamics (*state*, *tau*)

Computes forward dynamics

state [array like] len(state)=2 The state of the pendulum [angle, angular velocity] floats, units: rad, rad/s

Robotics Innovation Center
Robert-Hooke-Str. 1
28359 Bremen
Deutschland
robotik@dfki.de

tau [float] motor torque, unit: Nm

- float, angular acceleration, unit: rad/s^2

inverse_dynamics (*state*, *accn*)
Computes inverse dynamics

state [array like] len(state)=2 The state of the pendulum [angle, angular velocity] floats, units: rad, rad/s

accn [float] angular acceleration, unit: rad/s^2

tau [float] motor torque, unit: Nm

rhs (*t*, *state*, *tau*)
Computes the integrand of the equations of motion.

t [float] time, not used (the dynamics of the pendulum are time independent)

state [array like] len(state)=2 The state of the pendulum [angle, angular velocity] floats, units: rad, rad/s

tau [float or array like] motor torque, unit: Nm

res [array like] the integrand, contains [angular velocity, angular acceleration]

potential_energy (*state*)

kinetic_energy (*state*)

total_energy (*state*)

class Test (*methodName='runTest'*)
Bases: `unittest.case.TestCase`

MASSES = [0.01, 0.1, 1.0, 10.0, 100.0]

LENGTHS = [0.01, 0.1, 1.0, 10.0, 100.0]

DAMPINGS = [0.0, 0.01, 0.1, 1.0]

GRAVITY = [0.0, 9.81]

CFRICS = [0.0, 0.01, 0.1, 1.0]

TLIMITS = [1.0, 2.0, 3.0, 5.0, inf]

iterations = 10

epsilon = 0.0001

max_angle = 5.0

max_vel = 5.0

max_tau = 5.0

test_0_kinematics()
Unit test for pendulum kinematics

test_1_dynamics()
Unit test for pendulum dynamics

11.4 Reinforcement Learning

11.4.1 Deep Determininstic Policy Gradient (DDPG) Training

class Agent(*state_shape*, *n_actions*, *action_limits*, *discount*, *actor_lr*, *critic_lr*, *actor_model*, *critic_model*, *target_actor_model*, *target_critic_model*, *tau=0.005*)
Bases: `object`

prep_state(*state*)

get_action(*state*, *noise_object=None*)

scale_action(*action*, *mini*, *maxi*)

train_on(*batch*)

update_target_weights(*tau=None*)

save_model(*path*)

load_model(*path*)

class ddpg_trainer(*batch_size*, *validate_every=None*, *validation_reps=None*, *train_every_steps=inf*)
Bases: `object`

init_pendulum(*mass=0.57288*, *length=0.5*, *inertia=None*, *damping=0.15*, *coulomb_friction=0.0*, *gravity=9.81*, *torque_limit=2.0*)
Initialize the pendulum parameters.

mass [float, default=0.57288] mass of the pendulum [kg]

length [float, default=0.5] length of the pendulum [m]

inertia [float, default=None] inertia of the pendulum [kg m^2] defaults to point mass inertia (mass*length^2)

damping [float, default=0.15] damping factor of the pendulum [kg m/s]

coulomb_friction [float, default=0.0] coulomb friciton of the pendulum [Nm]

gravity [float, default=9.81] gravity (positive direction points down) [m/s^2]

torque_limit [float, default=2.0] the torque_limit of the pendulum actuator

init_environment(*dt=0.01*, *integrator='runge_kutta'*, *max_steps=1000*, *reward_type='open_ai_gym'*, *state_representation=2*, *validation_limit=- 150*, *target=[3.141592653589793, 0.0]*, *state_target_epsilon=[0.01, 0.01]*, *scale_action=True*, *random_init='everywhere'*)
Initialize the training environment. This includes the simulation parameters of the pendulum.

dt [float, default=0.01] time step [s]

integrator: string integration method to be used “euler” for euler integrator, “runge_kutta” for Runge-Kutta integrator

max_steps [int, default=1000] maximum number of timesteps for one training episode i.e. One episode lasts at most max_stepd*dt seconds

reward_type [string, default=soft_binary_with_repellor] string which defines the reward function options are: ‘continuous’, ‘discrete’, ‘soft_binary’,

‘soft_binary_with_repellor’

state_representation [int, default=2] determines how the state space of the pendulum is represented state_representation=2 means state = [position, velocity] state_representation=3 means state = [cos(position),

sin(position), velocity]

target [array-like, default=[np.pi, 0.0]] The target state of the pendulum

state_target_epsilon [array-like, default=[1e-2, 1e-2]] In this vicinity the target counts as reached.

scale_action [bool, default=True] whether to scale the output of the model with the torque limit of the simulator’s plant. If True the model is expected so return values in the intervall [-1, 1] as action.

init_agent(*replay_buffer_size=50000*, *actor=None*, *critic=None*, *discount=0.99*, *actor_lr=0.0005*, *critic_lr=0.001*, *tau=0.005*)

train(*n_episodes*, *verbose=True*)

save(*path*)

load(*path*)

get_actor(*state_shape*, *upper_bound=2.0*, *verbose=False*)

get_critic(*state_shape*, *n_actions*, *verbose=False*)

class OUActionNoise(*mean*, *std_deviation*, *theta=0.15*, *dt=0.01*, *x_initial=None*)

Bases: `object`

reset()

class ReplayBuffer(*max_size*, *num_states*, *num_actions*)

Bases: `object`

Replay buffer class to store experiences for a reinforcement learning agent.

append(*obs_tuple*)

Add an experience to the replay buffer. When adding experiences beyond the max_size limit, the first entry is deleted. An observation consists of (state, action, next_state, reward, done)

obs_tuple: array-like an observation (s,a,s’,r,d) to store in the buffer

sample_batch(*batch_size*)

Sample a batch from the replay buffer.

batch_size: int number of samples in the returned batch

tuple (s_batch,a_batch,s'_batch,r_batch,d_batch) a tuple of batches of state, action, reward, next_state, done

clear()
Clear the Replay Buffer.

11.4.2 Soft Actor Critic (SAC) Training

class sac_trainer(*log_dir='sac_training'*)
Bases: object

Class to train a policy for pendulum swingup with the state actor critic (sac) method.

init_pendulum(*mass=0.57288*, *length=0.5*, *inertia=None*, *damping=0.15*, *coulomb_friction=0.0*, *gravity=9.81*, *torque_limit=2.0*)
Initialize the pendulum parameters.

mass [float, default=0.57288] mass of the pendulum [kg]

length [float, default=0.5] length of the pendulum [m]

inertia [float, default=None] inertia of the pendulum [kg m^2] defaults to point mass inertia (mass*length^2)

damping [float, default=0.15] damping factor of the pendulum [kg m/s]

coulomb_friction [float, default=0.0] coulomb friciton of the pendulum [Nm]

gravity [float, default=9.81] gravity (positive direction points down) [m/s^2]

torque_limit [float, default=2.0] the torque_limit of the pendulum actuator

init_environment(*dt=0.01*, *integrator='runge_kutta'*, *max_steps=1000*, *reward_type='soft_binary_with_repellor'*, *state_representation=2*, *validation_limit=- 150*, *target=[3.141592653589793, 0.0]*, *state_target_epsilon=[0.01, 0.01]*, *random_init='everywhere'*)
Initialize the training environment. This includes the simulation parameters of the pendulum.

dt [float, default=0.01] time step [s]

integrator: string integration method to be used "euler" for euler integrator, "runge_kutta" for Runge-Kutta integrator

max_steps [int, default=1000] maximum number of timesteps for one training episode i.e. One episode lasts at most max_stepd*dt seconds

reward_type [string, default=soft_binary_with_repellor] string which defines the reward function options are: 'continuous', 'discrete', 'soft_binary',

'soft_binary_with_repellor'

state_representation [int, default=2] determines how the state space of the pendulum is represented state_representation=2 means state = [position, velocity] state_representation=3 means state = [cos(position),

sin(position), velocity]

target [array-like, default=[np.pi, 0.0]] The target state of the pendulum

state_target_epsilon [array-like, default=[1e-2, 1e-2]] In this vicinity the target counts as reached.

init_agent(*learning_rate=0.0003*, *warm_start=False*, *warm_start_path=''*, *verbose=1*)

Initilize the agent.

learning_rate [float, default=0.0003] learning rate of the agent

warm_start [bool, default=False] whether to use a pretrained model as initial model for training

warm_start_path [string, default=""] path to the model to load for warm start if warm_start==True

verbose [int, default=1] enable/disable printing of training progression to terminal

train(*training_timesteps=1000000.0*, *reward_threshold=1000.0*, *eval_frequency=10000*, *n_eval_episodes=20*, *verbose=1*)

Train the agent and save the model. The model will be saved to os.path.join(self.logdir, "best_model").

training_timesteps [int, default=1e6] total number of training steps. After training_steps steps the training terminates

reward_threshold [float, default=1000.0] If the evaluation of the agent surpasses this reward_threshold the training terminates early

n_eval_episodes [int, default=20] number of episodes used during evaluation

verbose [int, default=1] enable/disable printing of training progression to terminal

11.5 Simulation

class SimplePendulumEnv(*simulator*, *max_steps=5000*, *target=[3.141592653589793, 0.0]*, *state_target_epsilon=[0.01, 0.01]*, *reward_type='continuous'*, *dt=0.001*, *integrator='runge_kutta'*, *state_representation=2*, *validation_limit=- 150*, *scale_action=True*, *random_init='False'*)

Bases: `gym.core.Env`

An environment for reinforcement learning

step(*action*)

Take a step in the environment.

action [float] the torque that is applied to the pendulum

observation [array-like] the observation from the environment after the step

reward [float] the reward received on this step

done [bool] whether the episode has terminated

info [dictionary] may contain additional information (empty at the moment)

`reset` (*state=None*, *random_init='start_vicinity'*)

Reset the environment. The pendulum is initialized with a random state in the vicinity of the stable fixpoint (position and velocity are in the range[-0.31, 0.31])

state [array-like, default=None] the state to which the environment is reset if state==None it defaults to the random initialisation

random_init [string, default=None] A string determining the random state initialisation if None, defaults to self.random_init "False" : The pendulum is set to [0, 0], "start_vicinity" : The pendulum position and velocity

are set in the range [-0.31, -0.31],

"everywhere" [The pendulum is set to a random state in the whole] possible state space

observation [array-like] the state the pendulum has been initilized to

NotImplementedError when state==None and random_init does not indicate one of the implemented initializations

`render` (*mode='human'*)

Renders the environment.

The set of supported modes varies per environment. (And some environments do not support rendering at all.) By convention, if mode is:

- human: render to the current display or terminal and return nothing. Usually for human consumption.
- rgb_array: Return an numpy.ndarray with shape (x, y, 3), representing RGB values for an x-by-y pixel image, suitable for turning into a video.
- ansi: Return a string (str) or StringIO.StringIO containing a terminal-style text representation. The text can include newlines and ANSI escape sequences (e.g. for colors).

Note:

Make sure that your class's metadata 'render.modes' key includes the list of supported modes. It's recommended to call super() in implementations to use the functionality of this method.

Args: mode (str): the mode to render with

Example:

class MyEnv(Env): metadata = {'render.modes': ['human', 'rgb_array']}

def render(self, mode='human'):

if mode == 'rgb_array': return np.array(…) # return RGB frame suitable for video

elif mode == 'human': … # pop up a window and render

else: super(MyEnv, self).render(mode=mode) # just raise an exception

close()
Override close in your subclass to perform any necessary cleanup.

Environments will automatically close() themselves when garbage collected or when the program exits.

get_observation(*state*)
Transform the state from the simulator an observation by wrapping the position to the observation space. If state_representation==3 also transforms the state to the trigonometric value form.

state [array-like] state as output by the simulator

observation [array-like] observation in environment format

get_state_from_observation(*obs*)
Transform the observation to a pendulum state. Does nothing for state_representation==2. If state_representation==3 transforms trigonometric form back to regular form.

obs [array-like] observation as received from get_observation

state [array-like] state in simulator form

swingup_reward(*observation*, *action*)
Calculate the reward for the pendulum for swinging up to the instable fixpoint. The reward function is selected based on the reward type defined during the object inizialization.

state [array-like] the observation that has been received from the environment

reward [float] the reward for swinging up

NotImplementedError when the requested reward_type is not implemented

check_final_condition()
Checks whether a terminating condition has been met. The only terminating condition for the pendulum is if the maximum number of steps has been reached.

done [bool] whether a terminating condition has been met

is_goal(*obs*)
Checks whether an observation is in the goal region. The goal region is specified by the target and state_target_epsilon parameters in the class initialization.

obs [array-like] observation as received from get_observation

goal [bool] whether to observation is in the goal region

validation_criterion(*validation_rewards*, *final_obs=None*, *criterion=None*)
Checks whether a list of rewards and optionally final observations fulfill the validation criterion. The validation criterion is fulfilled if the mean of the validation_rewards id greater than criterion. If final obs is also given, at least 90% of the observations have to be in the goal region.

validation_rewards [array-like] A list of rewards (floats).

final_obs [array-like, default=None] A list of final observations. If None final observations are not considered.

criterion: float, default=None The reward limit which has to be surpassed.

passed [bool] Whether the rewards pass the validation test

class Simulator(*plant*)

Bases: `object`

set_state(*time*, *x*)

set the state of the pendulum plant

time: float time, unit: s

x: type as self.plant expects a state, state of the pendulum plant

get_state()

Get current state of the plant

self.t [float,] time, unit: s

self.x [type as self.plant expects a state] plant state

reset_data_recorder()

Reset the internal data recorder of the simulator

record_data(*time*, *x*, *tau*)

Records data in the internal data recorder

time [float] time to be recorded, unit: s

x [type as self.plant expects a state] state to be recorded, units: rad, rad/s

tau [type as self.plant expects an actuation] torque to be recorded, unit: Nm

euler_integrator(*t*, *y*, *tau*)

Euler integrator for the simulated plant

t [float] time, unit: s

y: type as self.plant expects a state state of the pendulum

tau: type as self.plant expects an actuation torque input

array-like : the Euler integrand

runge_integrator(*t*, *y*, *dt*, *tau*)

Runge-Kutta integrator for the simulated plant

t [float] time, unit: s

y: type as self.plant expects a state state of the pendulum

dt: float time step, unit: s

tau: type as self.plant expects an actuation torque input

array-like : the Runge-Kutta integrand

step(*tau*, *dt*, *integrator='runge_kutta'*)
Performs a single step of the plant.

tau: type as self.plant expects an actuation torque input

dt: float time step, unit: s

integrator: string "euler" for euler integrator "runge_kutta" for Runge-Kutta integrator

simulate(*t0*, *x0*, *tf*, *dt*, *controller=None*, *integrator='runge_kutta'*)
Simulates the plant over a period of time.

t0: float start time, unit s

x0: type as self.plant expects a state start state

tf: float final time, unit: s

controller: A controller object of the type of the AbstractController in simple_pendulum.controllers.abstract_controller.py If None, a free pendulum is simulated.

integrator: string "euler" for euler integrator, "runge_kutta" for Runge-Kutta integrator

self.t_values [list] a list of time values

self.x_values [list] a list of states

self.tau_values [list] a list of torques

simulate_and_animate(*t0*, *x0*, *tf*, *dt*, *controller=None*, *integrator='runge_kutta'*, *phase_plot=False*, *save_video=False*, *video_name='video'*)
Simulation and animation of the plant motion. The animation is only implemented for 2d serial chains. input: Simulates the plant over a period of time.

t0: float start time, unit s

x0: type as self.plant expects a state start state

tf: float final time, unit: s

controller: A controller object of the type of the AbstractController in simple_pendulum.controllers.abstract_controller.py If None, a free pendulum is simulated.

integrator: string "euler" for euler integrator, "runge_kutta" for Runge-Kutta integrator

phase_plot: bool whether to show a plot of the phase space together with the animation

save_video: bool whether to save the animation as mp4 video

video_name: string if save_video, the name of the file where the video will be stored

self.t_values [list] a list of time values

self.x_values [list] a list of states

self.tau_values [list] a list of torques

get_arrow(*radius*, *centX*, *centY*, *angle_*, *theta2_*, *color_='black'*)

set_arrow_properties(*arc*, *head*, *tau*, *x*, *y*)

11.6 Trajectory Optimization

11.6.1 DDP

11.6.2 Direct Collocation

11.6.3 iLQR

class iLQR_Calculator(*n_x=2*, *n_u=1*)

Bases: `object`

Class to calculate an optimal trajectory with an iterative linear quadratic regulator (iLQR). This implementation uses sympy.

set_start(*x0*)

Set the start state for the trajectory.

x0 [array-like] the start state. Should have the shape of (n_x,)

set_discrete_dynamics(*dynamics_func*)

Sets the dynamics function for the iLQR calculation.

danamics_func [function] dynamics_func should be a function with inputs (x, u) and output xd

rollout(*u_trj*)

set_stage_cost(*stage_cost_func*)

Set the stage cost (running cost) for the ilqr optimization.

stage_cost_func [function] stage_cost_func should be a function with inputs (x, u) and output cost

set_final_cost(*final_cost_func*)

Set the final cost for the ilqr optimization.

final_cost_func [function] final_cost_func should be a function with inputs x and output cost

cost_trj(*x_trj*, *u_trj*)

init_derivatives()

Initialize the derivatives of the dynamics.

compute_stage_cost_derivatives(*x*, *u*)

compute_final_cost_derivatives(*x*)

Q_terms(*l_x*, *l_u*, *l_xx*, *l_ux*, *l_uu*, *f_x*, *f_u*, *V_x*, *V_xx*)

gains(*Q_uu*, *Q_u*, *Q_ux*)

V_terms(*Q_x*, *Q_u*, *Q_xx*, *Q_ux*, *Q_uu*, *K*, *k*)

`expected_cost_reduction` (*Q_u*, *Q_uu*, *k*)

`forward_pass` (*x_trj*, *u_trj*, *k_trj*, *K_trj*)

`backward_pass` (*x_trj*, *u_trj*, *regu*)

`run_ilqr` (*N=50*, *init_u_trj=None*, *init_x_trj=None*, *shift=False*, *max_iter=50*, *break_cost_redu=1e-06*, *regu_init=100*)

Run the iLQR optimization and receive a optimal trajectory for the defined cost function.

N [int, default=50] The number of waypoints for the trajectory

init_u_trj [array-like, default=None] initial guess for the control trajectory ignored if None

init_x_trj [array_like, default=None] initial guess for the state space trajectory ignored if None

shift [bool, default=False] whether to shift the initial guess trajectories by one entry (delete the first entry and duplicate the last entry)

max_iter [int, default=50] optimization iterations the alogrithm makes at every timestep

break_cost_redu [float, default=1e-6] cost at which the optimization breaks off early

regu_init [float, default=100] initialization value for the regularizer

x_trj [array-like] state space trajectory

u_trj [array-like] control trajectory

cost_trace [array-like] trace of the cost development during the optimization

regu_trace [array-like] trace of the regularizer development during the optimization

redu_ratio_trace [array-like]

trace of ratio of cost_reduction and expected cost reduction during the optimization

redu_trace [array-like] trace of the cost reduction development during the optimization

`check_type` (*x*)

checks the type of x and returns the suitable library (pydrake.symbolic, sympy or numpy) for furhter calculations on x.

`pendulum_continuous_dynamics` (*x*, *u*, *m=0.5*, *l=0.5*, *b=0.15*, *cf=0.0*, *g=9.81*, *inertia=0.125*)

`pendulum_discrete_dynamics_euler` (*x*, *u*, *dt*, *m=0.57288*, *l=0.5*, *b=0.15*, *cf=0.0*, *g=9.81*, *inertia=0.125*)

`pendulum_discrete_dynamics_rungekutta` (*x*, *u*, *dt*, *m=0.5*, *l=0.5*, *b=0.15*, *cf=0.0*, *g=9.81*, *inertia=0.125*)

`pendulum_swingup_stage_cost` (*x*, *u*, *goal=[3.141592653589793, 0]*, *Cu=10.0*, *Cp=0.01*, *Cv=0.01*, *Cen=0.0*, *m=0.5*, *l=0.5*, *b=0.15*, *cf=0.0*, *g=9.81*)

`pendulum_swingup_final_cost` (*x*, *goal=[3.141592653589793, 0]*, *Cp=1000.0*, *Cv=10.0*, *Cen=0.0*, *m=0.5*, *l=0.5*, *b=0.15*, *cf=0.0*, *g=9.81*)

pendulum3_discrete_dynamics_euler (*x*, *u*, *dt*, *m=0.5*, *l=0.5*, *b=0.15*, *cf=0.0*, *g=9.81*, *inertia=0.125*)

pendulum3_discrete_dynamics_rungekutta (*x*, *u*, *dt*, *m=0.5*, *l=0.5*, *b=0.15*, *cf=0.0*, *g=9.81*, *inertia=0.125*)

pendulum3_swingup_stage_cost (*x*, *u*, *goal=[- 1, 0, 0]*, *Cu=10.0*, *Cp=0.01*, *Cv=0.01*, *Cen=0.0*, *m=0.5*, *l=0.5*, *b=0.15*, *cf=0.0*, *g=9.81*)

pendulum3_swingup_final_cost (*x*, *goal=[- 1, 0, 0]*, *Cp=1000.0*, *Cv=10.0*, *Cen=0.0*, *m=0.57288*, *l=0.5*, *b=0.15*, *cf=0.0*, *g=9.81*)

11.7 Utilities

syntax ()

motor_send_n_commands (*numTimes*)

motor_speed_test (*motor_id='0x01'*, *can_port='can0'*, *n=1000*)

profiler (*data_dict*, *start*, *end*, *meas_dt*)
validate avg dt of the control loop with (start time - end time) / numSteps

swingup (*args*, *output_folder*, *data_dict*)

grav_comp (*args*, *output_folder*, *data_dict*)

sys_id_unified (*output_folder*, *meas_time=None*, *meas_pos=None*, *meas_vel=None*, *meas_tau=None*, *acc=None*)

sys_id_comparison (*output_folder*, *meas_time*, *vel_dict*, *tau_dict*, *acc_dict*)

sys_id_result (*output_folder*, *t*, *ref_trq*, *est_trq*)

read (*WORK_DIR*, *params_file*, *urdf_file*, *csv_file*)

prepare_empty (*params*)

prepare_trajectory (*csv_path*)

inputs:

csv_path: string path to a csv file containing a trajectory in the below specified format

The csv file should have 4 columns with values for [time, position, velocity, torque] respectively. The values shopuld be separated with a comma. Each row in the file is one timestep. The number of rows can vary. The values are assumed to be in SI units, i.e. time in s, position in rad, velocity in rad/s, torque in Nm. The first line in the csv file is reserved for comments and will be skipped during read out.

Example:

time, position, velocity, torque 0.00, 0.00, 0.00, 0.10 0.01, 0.01, 0.01, -0.20 0.02, ….

cut (*data_measured*, *data_desired*)

Robotics Innovation Center
Robert-Hooke-Str. 1
28359 Bremen
Deutschland
robotik@dfki.de

save (*output_folder*, *data_dict*)

rad_to_deg (*theta_rad*)

deg_to_rad (*theta_deg*)

fast_fourier_transform (*data_measured*, *data_desired*, *n*, *t*)

scipy_fft (*data_measured*, *smooth_freq=100*)

online_filter (*data_measured*, *n*, *alpha*)

data_filter (*x*, *N*)

data_filter_realtime_1 (*data_measured_list*, *data_measured*, *window=10*)

data_filter_realtime_2 (*i*, *data_measured_list*, *window=10*)

savitzky_golay_filter (*data_measured*, *window*, *degree*)

PYTHON MODULE INDEX

s

t

u

INDEX

Robotics Innovation Center
Robert-Hooke-Str. 1
28359 Bremen
Deutschland
robotik@dfki.de

E

F

G

I

Robotics Innovation Center
Robert-Hooke-Str. 1
28359 Bremen
Deutschland
robotik@dfki.de

Robotics Innovation Center
Robert-Hooke-Str. 1
28359 Bremen
Deutschland
robotik@dfki.de

O

P

Q

R

S

Robotics Innovation Center
Robert-Hooke-Str. 1
28359 Bremen
Deutschland
robotik@dfki.de

Robotics Innovation Center
Robert-Hooke-Str. 1
28359 Bremen
Deutschland
robotik@dfki.de

Robotics Innovation Center
Robert-Hooke-Str. 1
28359 Bremen
Deutschland
robotik@dfki.de

T

U

V