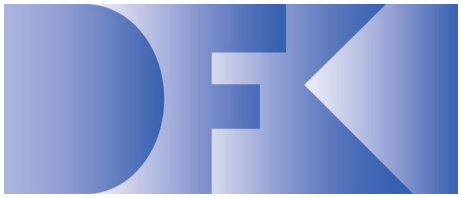
# Torque-limited Simple Pendulum: Reference

**Felix Wiebe, Jonathan Babel,**

**Contributors:**

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

January 2023



# TABLE OF CONTENTS

CHAPTER

ONE

# INTRODUCTION

## 1.1 Documentation

In this reference you will find installation and usage guides to get going with a real system, as well as dives into:

- Code testing
- The physics of a simple pendulum
- Our hardware and test bench
- Motor configuration

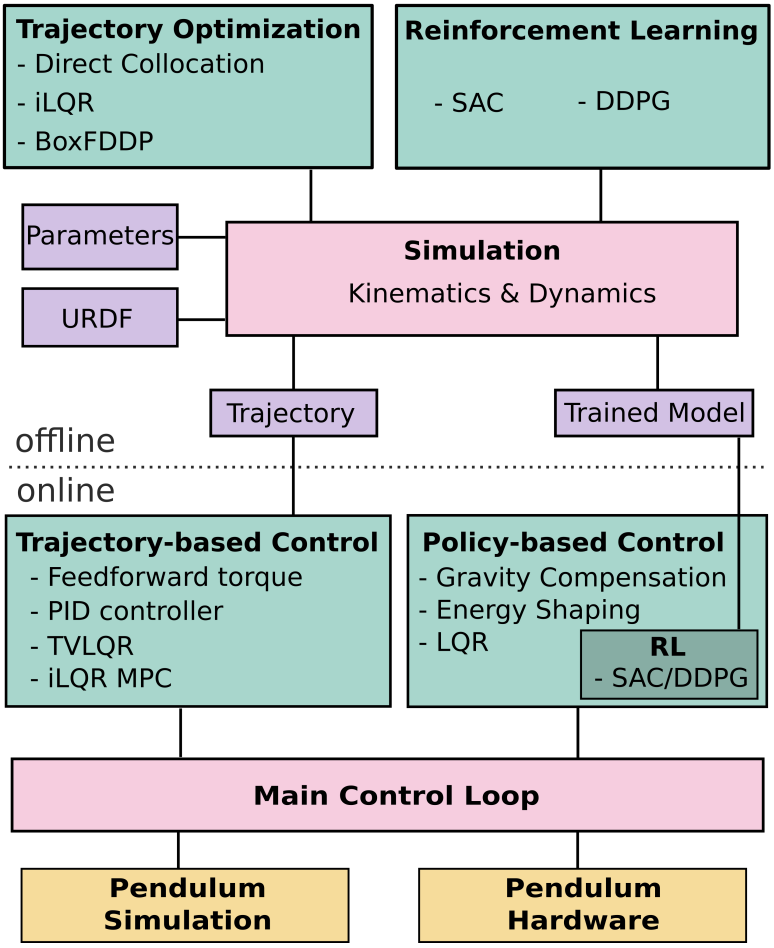
We've additionally uploaded all CAD files to grabcad.com: you can use its 3D viewer to diplay the 3D model directly within your browser.

A range of methods to solve the control problem will then be presented. Their implementation will be discussed, as well as their advantages and disadvantages observed experimentally.

Lastly you will find the API reference of the `simple_pendulum` package, including all functions available within it.

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

## 1.2 Overview of Methods




**Trajectory Optimization** tries to find a trajectory of control inputs and states that is feasible for the system while minimizing a cost function. The cost function can for example include terms which drive the system to a desired goal state and penalize the usage of high torques. The following trajectory optimization algorithms are implemented:

- Direct Collocation: A collocation method, which transforms the optimal control problem into a mathematical programming problem which is solved by sequential quadratic programming. For more information, click here
- Iterative Linear Quadratic Regulator (iLQR): A optimization algorithm which iteratively linearizes the system dynamics and applies LQR to find an optimal trajectory. For more information, click here
- Feasability driven Differential Dynamic Programming (FDDP): Trajectory optimization using locally quadratic dynamics and cost models. For more information about DDP, click here and for FDDP, click here

The optimization is done with a simulation of the pendulum dynamics.

**Reinforcement Learning** (RL) can be used to learn a policy on the state space of the robot, which then can be

used to control the robot. The simple pendulum can be formulated as a RL problem with two continuous inputs and one continuous output. Similar to the cost function in trajectory optimization, the policy is trained with a reward function. The following RL algorithms are implemented:

- Soft Actor Critic (SAC): An off-policy model free reinforcement learning algorithm. Maximizes a trade-off between expected return of a reward function and entropy, a measure of randomness in the policy. For more information, click here
- Deep Deterministic Policy Gradient (DDPG): An off-policy reinforcement algorithm which concurrently learns a Q-function and uses this Q-function to train a policy in the state space. For more information, click here.

Both methods, are model-free, i.e. they use the dynamics of the system as a black box. Currently, learning is possible in the simulation environment.

**Trajectory-based Controllers** act on a precomputed trajectory and ensure that the system follows the trajectory properly. The trajectory-based controllers implemented in this project are:

- Feed-forward torque Controller: Simple forwarding of a control signal from a precomputed trajectory.
- Proportional-Integral-Derivative (PID): A controller reacting to the position error, integrated error and error derivative to a precomputed trajectory.
- Time-varying Linear Quadreatic Regulator (tvLQR): A controller which linearizes the system dynamics at every timestep around the precomputed trajectory and uses LQR to drive the system towards this nominal trajectory.
- Model predictive control with iLQR: A controller which performs an iLQR optimization at every timestep and executes the first control signal of the computed optimal trajectory.

Feedforward and PID controller operate model independent, while the TVLQR and iLQR MPC controllers utilize knowledge about the pendulum model. In contrast to the others, the iLQR MPC controller optimizes over a predefined horizon at every timestep.

**Policy-based Controllers** take the state of the system as input and ouput a control signal. In contrast to trajectory optimization, these controllers do not compute just a single trajectory. Instead, they react to the current state of the pendulum and because of this they can cope with perturbations during the execution. The following policy-based controllers are implemented:

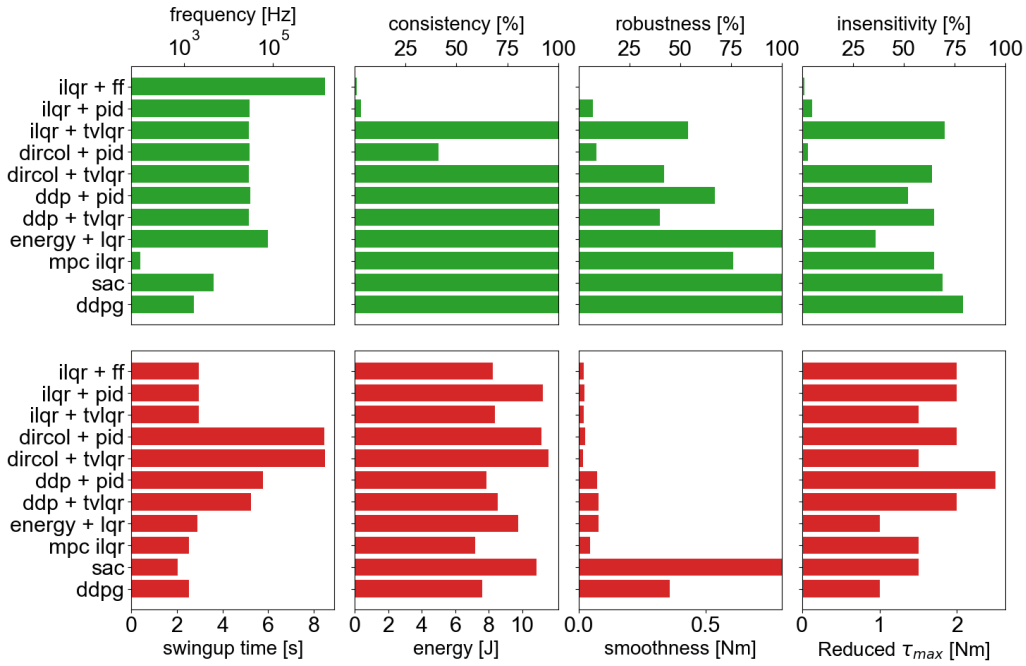
- Gravity Compensation: A controller compensating the gravitational force acting on the pendulum. The pendulum can be moved as if it was in zero-g.
- Energy Shaping: A controller regulating the energy of the pendulum. Drives the pendulum towards a desired energy level.
- Linear Quadratic Regulator (LQR): Linearizes the dynamics around a fixed point and drives the pendulum towards the fixpoint with a quadratic cost function. Only useable in a state space region around the fixpoint.

All of these controllers utilize model knowledge. Additionally, the control policies, obtained by one of the RL methods, fall in the category of policy-based control.

The implementations of direct collocation and TVLQR make use of drake, iLQR makes use of the symbolic library of drake or sympy, FDDP makes use of Crocoddyl, SAC uses the stable-baselines3 implementation and DDPG is implemented in Tensorflow. The other methods use only standard libraries.

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

The controllers can be benchmarked in simulation with a set of predefined criteria.




## 1.3 Authors

Project Supervisor:

- Shivesh Kumar

Software Maintainer:

- Felix Wiebe

Hardware Maintainer:

- Jonathan Babel

Contributors:

- Daniel Harnack
- Heiner Peters
- Shubham Vyas
- Melya Boukheddimi
- Mihaela Popescu

Feel free to contact us if you have questions about the test bench. Enjoy!

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

## 1.4 Contributing

1. Fork it
2. Create your feature branch:

```
git checkout -b feature/fooBar
```

3. Commit your changes:

```
git commit -am 'Add some fooBar'
```

4. Push to the branch:

```
git push origin feature/fooBar
```

5. Create a new Pull Request

See *How to Contribute* for more details.

## 1.5 Safety Notes

When working with a real system be careful and mind the following safety measures:

- Brushless motors can be very powerful, moving with tremendous force and speed. Always limit the range of motion, power, force and speed using configurable parameters, current limited supplies, and mechanical design.
- Stay away from the plane in which pendulum is swinging. It is recommended to have a safety net surrounding the pendulum in case the pendulum flies away.
- Make sure you have access to emergency stop while doing experiments. Be extra careful while operating in pure torque control loop.

## 1.6 Acknowledgements

This work has been performed in the VeryHuman project funded by the German Aerospace Center (DLR) with federal funds (Grant Number: FKZ 01IW20004) from the Federal Ministry of Education and Research (BMBF) and is additionally supported with project funds from the federal state of Bremen for setting up the Underactuated Robotics Lab (Grant Number: 201-001-10-3/2021-3-2).

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

## 1.7 License

This work has been released under the BSD 3-Clause License. Details and terms of use are specified in the LICENSE file within this repository. Note that we do not publish third-party software, hence software packages from other developers are released under their very own terms and conditions, e.g. Stable baselines (MIT License) and Tensorflow (Apache License v2.0). If you install third-party software packages along with this repo ensure that you follow each individual license agreement.

## 1.8 Citation

Felix Wiebe, Jonathan Babel, Shivesh Kumar, Shubham Vyas, Daniel Harnack, Melya Boukheddimi, Mihaela Popescu, Frank Kirchner. Torque-limited simple pendulum: A toolkit for getting familiar with control algorithms in underactuated robotics. In: Journal of Open Source Software (JOSS) (submitted).

CHAPTER

TWO

# 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

## 2.1 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).

## 2.2 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.

## 2.3 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
```

---

## 2.4 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
```

## 2.5 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
```

## 2.6 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
```

## 2.7 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 (except the ones which require Crocoddyl , see next section for instructions). Have fun exploring all kind of different simple pendulum controllers!

## 2.8 OPTIONAL: Crocoddyl and Gepetto Viewer

For installing the optimal control library Crocoddyl, we refer to the instructions provided in the Crocoddyl github repository and recommend the installation through robotpkg.

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

CHAPTER

THREE

# USAGE INSTRUCTIONS

## 3.1 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
```

## 3.2 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*

## 3.3 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.

## 3.4 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

## 3.5 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

FOUR

# 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
```

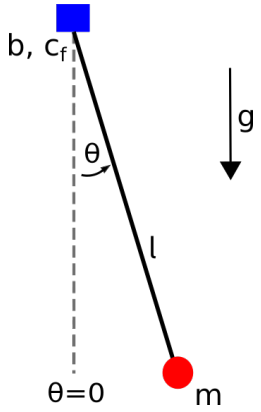
CHAPTER

FIVE

# THE PHYSICS OF A SIMPLE PENDULUM

## 5.1 Equation of Motion

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




where

- $\theta$, $\dot{\theta}$, $\ddot{\theta}$ 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)
- $\tau$ 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

## 5.2 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 ($E$)

$$E = K + U$$

## 5.3 `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 $\theta$ 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 $\tau$ 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 = $[\dot{\theta}, \ddot{theta}]$.

### 5.3.1 Usage

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

### 5.3.2 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 $\tau$, 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

### 5.3.3 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

SIX

# HARDWARE SETUP

## 6.1 Motor Configuration

The R-LINK Configuration Tool is used to configure the AK80-6 from T-Motors. Before starting to use the R-Link device make sure you have downloaded the *CP210x Universal Windows Driver* from silabs. If this isn't working properly follow the instructions at sparkfun on how to install ch340 drivers. You have to download the *CH 341SER (EXE)* file from the sparkfun webpage. Notice that you first have to select uninstall in the CH341 driver menu to uninstall old drivers before you are able to install the new driver. The configuration tool software for the R-LINK module can be downloaded on the T-Motors website.

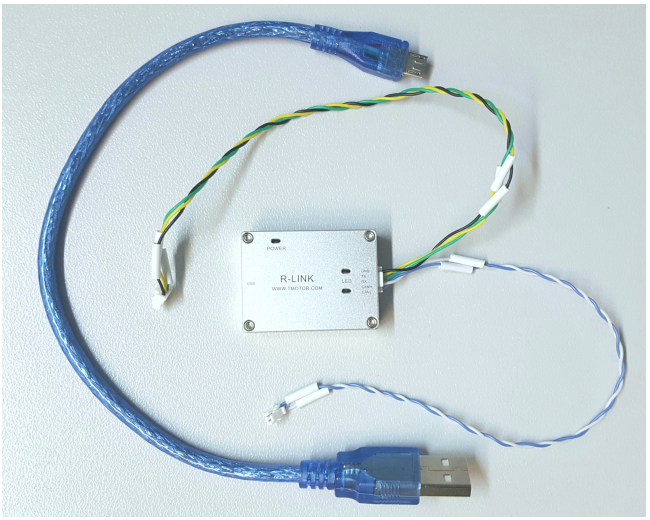
- **Silabs:** CP210x Universal Windows Driver
- **CH341:** Sparkfun - How to install CH340 drivers

### 6.1.1 Tutorials

- T-MOTOR
- Skyentific

### 6.1.2 UART Connection: R-Link module

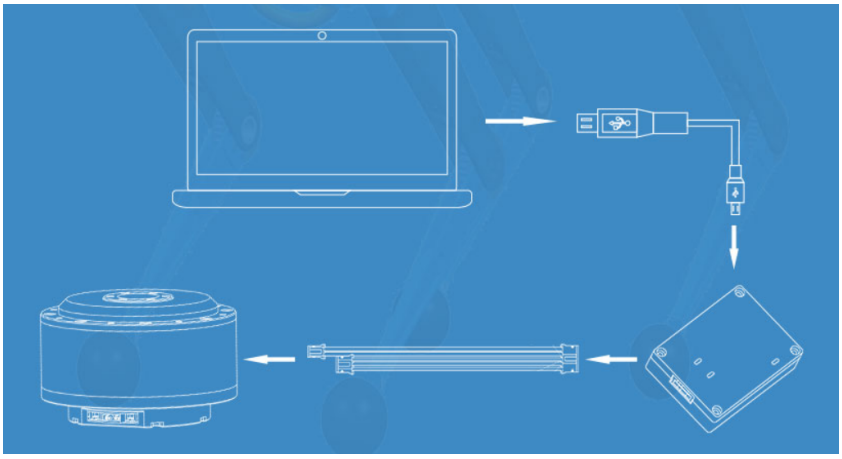
R-LINK is a USB to serial port module, specially designed for CubeMars A Series of dynamical modular motors. It is possible to calibrate the encoder in the module, change CAN ID settings, PID settings, as well as to control position, torque and speed of the motor within the configuration software tool.




### 6.1.3 Instructions: R-Link Config Tool

**User manual & configuration tool:** store-en.tmotor.com

1. Wire the R-LINK module as shown in the figure below. A USB to micro USB cable connects a pc with the R-LINK module and the 5pin cable goes between the R-LINK module and the Motor.



2. Connect the AK80-6 motor to a power supply (24V, 12A) and do not cut off the power before the setting is completed. <br/>
3. Start the R-Link Config Tool application (only runs on Windows).
4. Select serial port: USB-Serial_CH340, wch, cp along with an appropriate baud rate (both 921600 and 115200 Bd should work). If the serial port option USB-Serial_CH340,wch,cp does not show up, your pc can't establish a connection to the R-LINK module due to remaining driver issues.
3. Choose the desired motor settings on the left side of the config tool GUI. Enter the correct CAN ID of the motor under *MotorSelectEnter*. A label on the motor shows the ID.

- Velocity: 5 rad/s is a relatively slow speed of revolution, hence it offers a good starting point.
- Torque: be careful setting a fixed torque, because the friction inside the motor decreases with the speed of revolution. Therefore a fixed torque commonly leads to either no movement at all or accelerates the motor continuously.

4. Start the plotting by ticking the boxes of position, velocity, torque and select *Display*
5. Press `Run` to start recording the plots.
6. `Enter M_Mode` to control the motor. This is indicated by a color change of the plot line, from red to green.
7. In order to push changes in the settings to the motor, press *Send Once*.

> **Warning:** This button does not work reliably. Usually it has to be activated several times before the setting changes actually apply on the motor.

8. Stop the motor inside the M-Mode by setting the velocity to 0 and pressing *Send Once* until the changes apply.
9. `Exit M_Mode` to exit the control mode of the motor.

> **Warning:** The next time you start the motor control with `Enter M_Mode` the motor will restart with the exact same settings as you left the control mode with *Exit M_Mode*. This is especially dangerous if a weight is attached to the pendulum and the motor control was left with high velocity or torque settings.

10. Use `Stop` to deactivate the plotting.

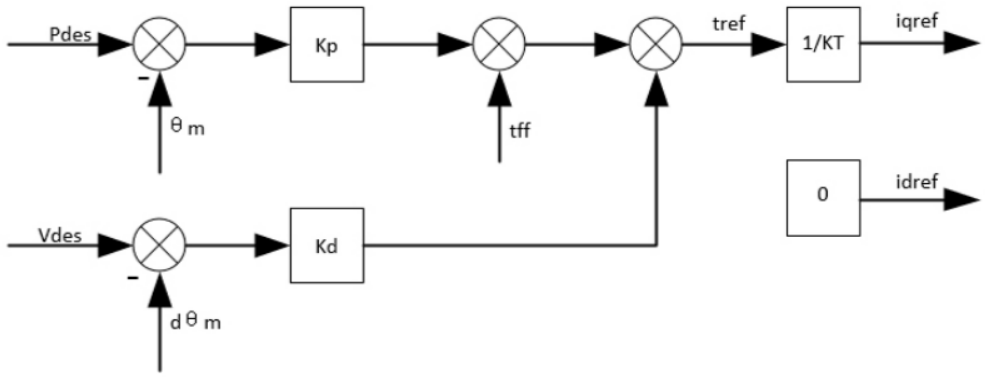
### 6.1.4 Debugging

Error messages that showed up during the configuration procedure, such as *UVLO* (VM undervoltage lockout) and *OTW* (Thermal warning and shutdown), could be interpreted with the help of the datasheet for the DRV8353M 100-V Three-Phase Smart Gate Driver from Texas Instruments:

**Datasheet:** DRV8353M (on the first Page under: 1. Features)

### 6.1.5 PD-Controller

A proportional-derivative controller, which is based on the MIT Mini-Cheetah Motor, is implemented on the motor controller board. The control block diagram of this closed loop controller is shown below. It can bee seen that the control method is flexible, as pure position, speed, feedforward torque control or any combination of those is possible.




In the motor driver,:

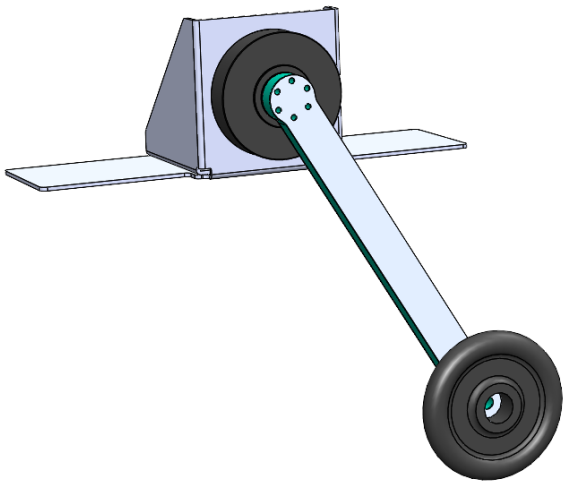
```
send_rad_command(position_in_radians, velocity_in_radians, Kp, Kd, tau_ff)
```

lets you set desired position (Pdes), velocity (Pvel), Kp, Kd and feedforward torque (tff) values at every time step.

## 6.2 Hardware & Testbench Description

The `/hardware` directory contains all information about the hardware that is used to built the simple pednulum test bench, including a bill of materials, step files of the CAD model along with wiring diagrams for the complete set up as well as the CAN bus.

We additionally uploaded all CAD files to grabcad.com. You can use the 3D viewer from their webiste to diplay the 3D model directly within your browser: grabcad.com/simple_pendulum



### 6.2.1 Physical Parameters of the Pendulum

- Point mass: $m_p$ = 0.546 Kg
- Mass of rod, mounting parts and screws: $m_r$ = 0.13 Kg
- Overall mass: $m$ = 0.676 Kg
- Length to point mass: $l$ = 0.5 m
- Length to COM: $l_{COM}$ = 0.45 m

$$l_{COM} = \frac{m_p l 0.5 m_r l}{m_p m_r}$$

### 6.2.2 Physical Parameters of the Actuator

The AK80-6 actuator from T-Motor is a quasi direct drive with a gear ratio of 6:1 and a peak torque of 12 Nm at the output shaft. The motor is equipped with an absolute 12 bit rotary encoder and an internal PD torque control loop. The motor controller is basically the same as the one used for MIT Mini-Cheetah, which is described Ben Katz's MIT Mini-Cheetah Documentation



- Voltage = 24 $V$
- Current = rated 12 $A$, peak 24 $A$
- Torque = rated 6 $Nm$, peak 12 $Nm$ (after the transmission)
- Transmission N = 6 : 1
- Weight = 485 $g$
- Dimensions = ⌀ 98 $mm$ x 38,5 $mm$
- Max. torque to weight ratio = 24 $kg$ (after the transmission)
- Max. velocity = 38.2 $s$ = 365 $rpm$ (after the transmission)
- Backlash (accuracy) = 0.15° degrees

### 6.2.3 Motor Constants

**Note:** Before the transmission

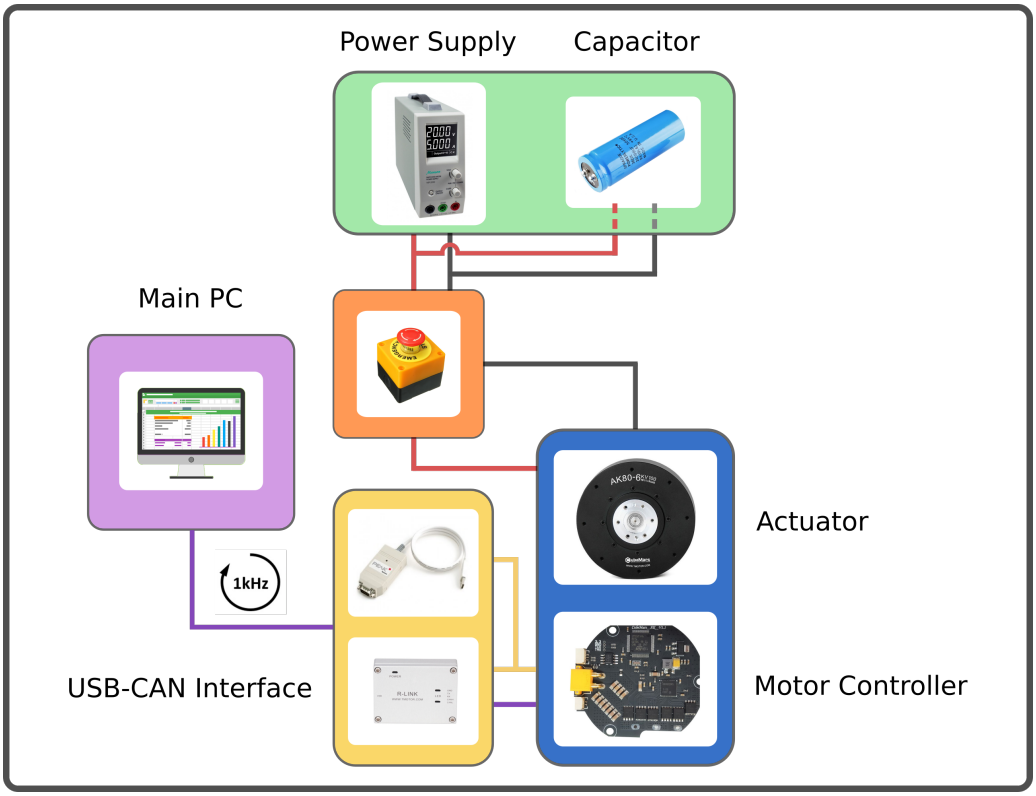
- Motor constant $km$ = 0.2206 $Nm/\sqrt{W}$
- Electric constant $ke$ = 0.009524 $V/rpm$
- Torque constant $kt$ = 0.091 $Nm/A$
- Torque = rated 1,092 $Nm$, peak 2,184 $Nm$
- Velocity / back-EMF constant $kv$ = 100 $rpm/V$
- Max. velocity at 24 $V$ = 251.2 $rad/s$ = 2400 $rpm$
- Motor wiring in $\nabla$-delta configuration
- Number of pole pairs = 21
- Resistance phase to phase = 170 $\pm$ 5 $m\Omega$
- Inductance phase to phase = 57 $pm$ 10 $\mu H$
- Rotor inertia Ir = 0.000060719 $kg/m^2$

### 6.2.4 Electrical Setup

**Note:** We do not give any safety warranties on the electrical wiring. All experiments and reproductions of our test bed are at your own risk.

The wiring diagram below shows how the simple pendulum testbench is set up. A main PC is connected to a motor controller board (**CubeMars_AK_V1.1**) mounted on the actuator (**AK80-6 from T-Motor**). The communication takes place on a CAN bus with a maximum signal frequency of 1Mbit/sec with the 'classical' CAN protocol. Furthermore, a USB to CAN interface is needed, if the main pc doesn't have a PCI CAN card. Two different devices are used in our setup: the **R-LINK module** from T-Motor and the **PCAN-USB adapter from PEAK systems**. The former has CAN and UART connectors at the output, but only works with Windows. The latter only features CAN connection, but also works with Linux. The UART connector of the R-LINK module is usefull to configure and calibrate the AK80-6.

The actuator requires an input voltage of 24 Volts and consumes up to 24 Amps under full load. A power supply that is able to deliver both and which is used in our test setup is the **EA-PS 9032-40** from Elektro-Automatik. A capacitor filters the backEMF coming from the actuator and therefore protects the power supply from high voltage peaks. This wouldn't be necessary if the actuator is powered from a battery pack, as in this case backEMF simply recharges the batteries. The capacitor we use is made of **10x single 2.7V-400 F capacitor cells** connected in series resulting a total capacity of 40 F and is wired in parallel to the motor. A emergency stop button serves as additional safety measure. It disconnects the actuator from power supply and capacitor, if only the power supply gets diconnected the actuator will keep running with the energy stored in the capacitor.




- Actuator: AK80-6
- Controller board: CubeMars_AK_V1.1
- Power supply: EA-PS 9032-40
- Capacitor: 10x 2.7V-400F cells connected in series
- USB-CAN interfaces: R-LINK module and PCAN-USB adapter.

### 6.2.5 backEMF

The reverse current resulting from switching motor speeds from high to low is called backEMF (Electro Magnetic Force). When the motor speed decreases the motor works as a generator, which converts mechanical energy into electrical energy and hence the additional current needs some path to flow. The energy recycled back into the input power supply causes a voltage spike and potential risk. It is necessary to add enough input capacitance to absorb this energy. A sufficiently large input capacitance is important in the desgin of the electric curcuit. It is beneficial to have more bulk capacitance, but the disadvantages are increased cost and physical size.

If the power source were a perfect battery, then energy would flow back into the battery and be recycled. However, in our case the power source is a DC power supply. Especially power supplies with an inverse-polarity protection diode can only source current and cannot sink current, hence the only place the energy can go is into the bulk capacitor. The amount of energy stored in the bulk capacitor can be calculated with

$$E = \frac{1}{2} \cdot C \cdot (V_{max}^2 - V_{nom}^2)$$

where **C** is the capacitance and **V** is the voltage. In the case of a Simple Pendulum max. backEMF can be estimated from the kinetic energy of the pendulum

$$E_kin = \frac{1}{2}m \cdot v^2$$

where **m** is the payload attached to the rod and **v** is the velocity of the payload. The voltage across the capacitor increases as energy flows into it, so the capacitor should be sized accordingly to the specific application requirements. Nevertheless tuning a capacitor to the acceptable min. capacity is tricky, because it depends on many factors including:

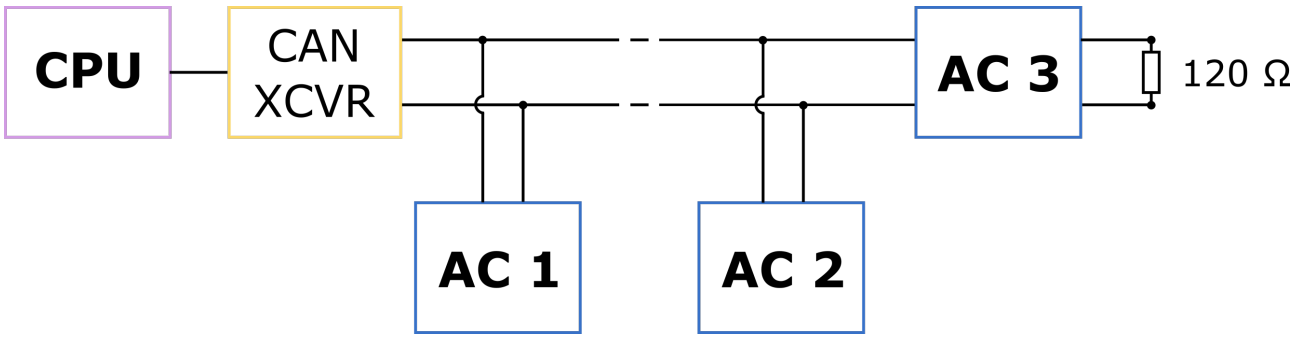
- External load
- Capacitance of the power supply to source current
- Motor braking method, output short brake or current polarity reversing brake.
- Amount of parasitic inductance between power supply and motor system, which limits the current change rate from the power supply. The larger the input capacitance, the more stable the motor voltage and higher current can be quickly supplied.
- The maximum supply voltage limit and acceptable voltage ripples

If the used capacitor is too small for your specific apllication it introduces the risk of burning the capacitor. The voltage rating for the bulk capacitors should be higher than the typical operating voltage and provide some safty margin. In our case we supply the AK80-6 with 24 V, whereas the capacitor can take up to 27 V. Therefore we have 3 V buffer, combined with a large capacity of 40 F, we ensure that during voltage spikes the capacitor never gets fully charged. If you don't want to buy a huge and expensive capacitor you may instead use a **break resistor**, which normally is cheaper to purchase. A guidance on this topic is provided here. One drawback using brake resistors is that they quickly heat up, if the motor frequently brakes and regenerates energy. Another option to prevent the bus voltage from spiking too high are **resistive shunt regulators**, e.g. like this one from polulu, but they can't dissipate much power and high-power versions also get expensive.

## 6.3 Communication: CAN Bus wiring

Along the CAN bus proper grounding and isolation is required. It is important to not connect ground pins on the CAN bus connectors between different actuators, since this would cause a critical ground loop. The ground pin should only be used to connect to systems with a ground isolated from the power ground. Additionally, isolation between the main pc and the actuators improves the signal quality. When daisy-chaining multiple actuators, only the CAN-High and CAN-Low pins between the drives must be connected. At the end of the chain a 120 Ohm resistor between CAN-H and CAN-L is used to absorb the signals. It prevents the signals from being reflected at the wire ends. The CAN protocol is differential, hence no additional ground reference is needed. The diagram below displays the wiring of the CAN bus.

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




- Main pc: CPU
- CAN transceiver: CAN XCVR
- Actuator: AC

CHAPTER

SEVEN

# TRAJECTORY OPTIMIZATION

## 7.1 Trajectory optimization using direct collocation

**Note:** Type: Trajectory Optimization

State/action space constraints: Yes

Optimal: Yes

Versatility: Swingup and stabilization

### 7.1.1 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] [2]. The formulation of the optimization problem at the collocation points is as follows:

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

- $\mathbf{x} = [\theta(.), \dot{\theta}(.)]^T$>: 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
- $t_{c,k} = \frac{1}{2}(t_k t_{k1})$: 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.

### 7.1.2 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.

### 7.1.3 Dependencies

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

### 7.1.4 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/>*

## 7.2 Iterative Linear Quadratic Regulator (iLQR)

---

**Note:** Type: Trajectory Optimization

State/action space contraints: No

Optimal: Yes

Versatility: Swingup and stabilization

---

### 7.2.1 Theory

The iterative linear quadratic regularizer (iLQR) `[1] 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.

### 7.2.2 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.

### 7.2.3 Usage

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

```
python compute_iLQR_swingup.py
```

### 7.2.4 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.

### 7.2.5 Requirements

Optional: pydrake [2] (see getting_started)

### 7.2.6 Notes

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

### 7.2.7 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

## 7.3 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.

### 7.3.1 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 $\alpha_i$ for this optimization problem are determined experimentally.

### 7.3.2 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()
```

### 7.3.3 Usage

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

```
python compute_BOXFDDP_swingup.py
```

### 7.3.4 Dependencies

Crocoddyl

Pinocchio

For the display, Gepetto

### 7.3.5 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

EIGHT

# REINFORCEMENT LEARNING

## 8.1 Soft Actor Critic Training

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

State/action space constraints: None

Optimal: Yes

Versatility: Swing-up and stabilization

### 8.1.1 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]:

### 8.1.2 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

### 8.1.3 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.

### 8.1.4 Comments

Todo: comments on training convergence stability

### 8.1.5 Requirements

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

### 8.1.6 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)

## 8.2 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

---

### 8.2.1 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 $\tau$ 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].

### 8.2.2 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

### 8.2.3 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.

### 8.2.4 Comments

Todo: comments on training convergence stability

### 8.2.5 Requirements

- Tensorflow 2.x

### 8.2.6 References

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

[2] reras guide

CHAPTER

NINE

# TRAJECTORY-BASED CONTROLLERS

## 9.1 Open Loop Control

**Note:** Type: Open loop control

State/action space contraints: -

Optimal: -

Versatility: -

### 9.1.1 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.

### 9.1.2 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.

### 9.1.3 Usage

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

### 9.1.4 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`.

## 9.2 Proportional-Integral-Derivative (PID) Control

---

**Note:** Type: Closed loop control

State/action space contraints: -

Optimal: -

Versatility: -

---

### 9.2.1 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 $\tau$ 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:

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

### 9.2.2 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.

### 9.2.3 Usage

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

### 9.2.4 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.

## 9.3 Time-varying Linear Quadratic Regulator (TVLQR)

---

**Note:** Type: Closed loop control

State/action space contraints: -

Optimal: -

Versatility: -

---

### 9.3.1 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.

### 9.3.2 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.

### 9.3.3 Usage

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

### 9.3.4 Dependencies

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

### 9.3.5 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.

### 9.3.6 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.

### 9.3.7 Region of Attraction (RoA) estimation

The region of attraction estimation is an offline process that can be applied to the closed-loop dynamics of a system (Plant and Controller) in order to analyze the control behaviour. The time-varying version of this analysis offers a very interesting point of view to understand how much the state can vary with respect to a nominal trajectory that brings a fixed initial state to a desired state. Hence, this process results in a very powerful tool to verify the robustness of a time-varying controller.

#### Probabilistic method

For simple systems, such as the torque limited simple pendulum under TVLQR control, the RoA can be estimated by a very intuitive simulation-based method. The implemented solution takes inspiration from some related works that has been studied and extended. E. Najafi proposed a computationally effective sampling approach to estimate the DoAs of nonlinear systems in real time:

E. Najafi, R. Babuška, and G. A. D. Lopes, “A fast sampling method for estimating the domain of attraction,” Nonlinear Dyn, vol. 86, no. 2, pp. 823–834, Oct. 2016, doi: 10.1007/11071-016-2926-7.

He used this method for estimating a time-invariant region of attraction in two different ways: “memoryless sampling” and “sampling with memory”. We have exploited his results from the memoryless version by extending the study to the time-varying case. This extension has been already somehow addressed by P. Reist which included it in a simulation-based variant of the LQR-Tree feedback-motion-planning approach:

P. Reist, P. V. Preiswerk, and R. Tedrake, “Feedback-motion-planning with simulation-based lqr-trees”, SAGE, 2016, https://groups.csail.mit.edu/robotics-center/public_papers/Reist15.pdf.

However, his implementation is strictly related to the LQR-Tree algorithm while we will focus on the region of attraction estimation.

### Estimation procedure

First we have to fix the number of knot points N. Except for the given last value, rho is initially fixed to an N-dimensional array of infinity. The last value of rho comes from the time-invariant region of attraction estimation, while the other ones have to be fixed big enough to overcome its "real" value. Now, after computing the nominal trajectory and the related TVLQR controller that bring to the goal the estimation process can start. Iterating backward in the knot points, we can exploit the knowledge of the final rho from the time-invariant case. The "previous" and the "next" ellipse have been considered. We sample random initial states from the "previous" ellipse and we simulate them until the "next" one. In doing so, we can check if the simulated trajectory exits from the "next" ellipse. The condition is the following one:

$$x^T S x < \rho$$

If a simulated trajectory triggers this condition, the algorithm shrinks the "previous" ellipse using the value of the computed optimal cost to go:

$$\rho_{new} = min(\rho_{old}, ctg)$$

An implementation detail gives the possibility to reduce the time consumption. The estimation of each ellipse has been considered done after a maximum number of successful simulations.

### SOS method

For the class of systems with polynomial dynamics f (x) , this problem can be formulated as a convex optimization problem using sums-of-squares (SOS) optimization. This different approach has very interesting advantages in terms of Scalability and it doesn't need any Discretization in the time domain. Furthermore, it allows to reason about the RoA algebraically and does not require numerical simulations, that can be expensive. On the other hand, it requires to express the closed-loop dynamics in a polynomial form. This might need some approximation, via Taylor series for examples, of the dynamics that can cause some difference between the simulation and the real behaviour.

My implementation of this estimation method is based on:

- Invariant Funnels around Trajectories using Sum-of-Squares Programming", Mark M. Tobenkin, Ian R. Manchester, Russ Tedrake (https://doi.org/10.3182/20110828-6-IT-1002.03098)
- Funnel libraries for real-time robust feedback motion planning", Anirudha Majumdar, Russ Tedrake (https://doi.org/10.1177/0278364917712421)
- Trajectory Optimization and Time-Varying LQR Stabilization of Airplane Longitudinal Dynamics" (https://github.com/benthomsen/mit-6832-project/blob/master/bthomsen-6832-report.pdf)

### Estimation procedure

$$\begin{aligned} \max_{\rho_i, \lambda_i} \quad & \rho_i \\ \text{subject to} \quad & -(\dot{V} - \dot{\rho}_i) + \lambda_i (V - \rho_i) \text{ is SOS} \\ & \lambda_i \text{ is SOS} \\ & \rho_i > 0 \end{aligned}$$

This is the optimization problem that has to be solved in order to obtain the value of rho. The Lyapunov condition has been imposed by exploiting the so-called S-procedure. Unfortunately, this formulation is not convex in rho, so that it has to be solved with a bilinear alternation by fixing at each step the Lagrangian multiplier or rho.

Furthermore, it is worthwhile to mention the consequences of considering the input saturation constraint. It can be addressed by using some Lagrangian multipliers. However, this can lead to add a number of SOS conditions that grows exponentially with the number of inputs. Hence, this has to be considered in systems where there is a big number of inputs.

## 9.4 iLQR Model Predictive Control

---

**Note:** Type: Model Predictive Control

State/action space contraints: No

Optimal: Yes

Versatility: Swingup and stabilization

---

### 9.4.1 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.

### 9.4.2 Requirements

pydrake (see getting_started)

### 9.4.3 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.

### 9.4.4 Usage

The controller can be tested in simulation with:

```
python sim_ilqrMPC.py
```

### 9.4.5 Comments

For coulomb_fricitons != 0 the optimization gets considerably slower.

CHAPTER

TEN

# POLICY-BASED CONTROLLERS

## 10.1 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

### 10.1.1 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 $\theta$ 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.

## 10.2 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

### 10.2.1 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.

### 10.2.2 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).

### 10.2.3 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
```

## 10.3 LQR Control

---

**Note:** Type: Closed loop control

State/action space contraints: No

Optimal: Yes

Versatility: Stabilization only

---

### 10.3.1 Theory

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

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$$

and a infinite horizon cost function in quadratic form:

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

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.

### 10.3.2 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.

### 10.3.3 Usage

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

```
python sim_lqr.py
```

### 10.3.4 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.

### 10.3.5 Region of Attraction (RoA) estimation

The RoA estimation is a process that can be applied to the closed-loop dynamics of a system, i.e. Plant + Controller, in order to analize its state-space behaviour. Two different methods have been implemented here for studying the dynamics of the underactuated pendulum coupled with an LQR controller.

#### SOS method

The implementation here is based on

> Tedrake, Russ, Ian R. Manchester, Mark Tobenkin, and John W. Roberts. "LQR-Trees: Feedback Motion Planning via Sums-of-Squares Verification." The International Journal of Robotics Research 29, no. 8 (July 2010): 1038–52. https://doi.org/10.1177/0278364910369189.

Sum of squares optimization provides a natural generalization of SDP to optimizing over positive polynomials. Hence this method can be exploited to formulate and solve problems from Lyapunov analysis, at least fot the polynomial systems. Furthermore, any sublevel set of a Lyapunov function is also an invariant set. This permitts to use sublevel sets of a Lyapunov function as approximation of the region of attraction for nonlinear systems.

First, the simple pendulum plant has been put in polynomial form via Taylor approximation around the up-right position, which will be the goal of the LQR controller. From different tests it seems that at least the third order approximation is necessary to obtain a satisfying result.

The LQR controller has been initialized to stabilize our closed-loop system around the up-right position. From that it has been obtained a state feedback matrix K and a matrix S, which is the solution of the matrix Lyapunov equation. The last one is very usefull because it can be used to obtain the Lyapunov function as $x^T S x$ After obtaining the Lyapunov function the feasibility problem can be formulated as

$$-\dot{V}(x) + \lambda(x)(V(x) - \rho) \; is \; SOS \quad and \quad \lambda(x) \; is \; SOS$$

This is coming from the so called "S-procedure" which makes use of the concept of Lagrangian multiplier to fomulate the Lyapunov conditions. It is important to highlight that other contraints are needed to include the torque inputs limits in the optimization problem.

Eventually, the above problem only verify one-sublevel set of the Lyapunov function. In order to obtain the best estimation, searching for the largest rho that can satisfy these conditions is necessary. The two different methods that have been implemented are described below.

### Maximization of $\rho$: Simple line search

Since the problem is convex with rho fixed, and $\rho$ is just a scalar, a simple line search on $\rho$ can be performed to find the maximum for which the convex optimization returns a feasible solution. In particular, a bisection-like algorithm has been implemented here for such a purpose. However, since this formulation can be computationally heavy then other formulations might be considered.

### Maximization of $\rho$: Equality-constrained formulation

This is an important variation since it makes use of the "S-procedure" to make the problem be jointly convex in $\lambda(x)$ and $\rho$, so a single convex optimization is needed. Under the assumption that $\dot{V}(x)$ is negative-definite at the fixed point, the problem can be written as

$$\begin{aligned} &\max_{\rho,\lambda} \quad \rho \\ &(x^T x)^d (V(x) - \rho) + \lambda(x)\dot{V}(x) \quad is \quad SOS \end{aligned}$$

Also in this case, input limits have then to be included to obtain the desired result.

### Probabilistic Method

For simple systems, such as the torque limited simple pendulum under LQR control, the RoA can be estimated by just evaluating the Lyapunov conditions for initial states from a continuously shrinking estimate of the ROA. This approach was introduced in:

E. Najafi, R. Babuška, and G. A. D. Lopes, "A fast sampling method for estimating the domain of attraction," Nonlinear Dyn, vol. 86, no. 2, pp. 823–834, Oct. 2016, doi: 10.1007/s11071-016-2926-7.

The RoA is estimated by a sublevel set of a quadratic Lyapunov function:

$$\mathcal{B} = \{\mathbf{x} | V(\mathbf{x}) < \rho\} = \left\{\mathbf{x} | V \bar{\mathbf{x}}^{\mathrm{T}} \mathbf{S} \bar{\mathbf{x}} < \rho\right\}.$$

Where $\mathbf{x} = \mathbf{x} - \mathbf{x}^\star$ denotes the deviation from the fixed point $\mathbf{x}^\star$

A random initial condition $V(\hat{\mathbf{x}})$ is sampled from $\mathcal{B}$ and the Lyapunov conditions ($V(\hat{\mathbf{x}}) > 0$ and $\dot{V}(\hat{\mathbf{x}}) = \nabla V \mathbf{f}(\mathbf{x}) < 0$) are evaluated. If these conditions are satisfied, the next initial state is sampled. However, if these conditions are not met, the estimate is shrunk, such that $\rho = V(\hat{\mathbf{x}})$

### Comments

Further possible improvements:

- Verifying the dynamics in implicit form will be necessary for more complex systems where the mass is not a scalar value. The SOS framework premitts it and would be usefull to try.
- Implementing more efficient ways to come up with a polynomial representation of the closed-loop dynamics. Approxiations usually generate differences between the real behaviour and the simulated one.
- Quotient ring of algebraic varieties in the "Equality-constrained formulation".

CHAPTER

ELEVEN

# CONTROLLER ANALYSIS

## 11.1 Controller Benchmarking

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

### 11.1.1 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.

### 11.1.2 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.

### 11.1.3 Usage

An example usage can be found in the examples folder in the **`benchmark_controller.py <https://github.com/dfki-ric-underactuated-lab/torque_limited_simple_pendulum/blob/master/software/python/exan** script.

## 11.2 Plotting Controllers

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

### 11.2.1 API

A controller inheriting from the abstract controller class 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.

### 11.2.2 Usage

An example usage can be found in the examples folder in the plot_controller.py script.

CHAPTER

TWELVE

# SIMULATOR

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

## 12.1 API

### 12.1.1 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)

### 12.1.2 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

## 12.2 Usage

For examples of usages of the simulator class check out the scripts in the examples folder.

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

CHAPTER

# THIRTEEN

# HOW TO CONTRIBUTE

If you want to contribute to the project, i.e. by designing new controllers, you are very welcome to do so.

If you implement a controller please consider the following guidelines:

1. Create a folder with a unique and descriptive name for your controller in software/python/controllers.
2. Make sure your controller inherits from the AbstractController class.
3. Create a `README.md` file in your controller directory in which you explain how the controller works and how it can be used.
4. Create a `requirements.txt` file where you list required packages that are not listed in the main requirements file of the project.

If you implement an offline trajectory optimization or reinforcement learning algorithm follow these guidelines:

1. Create a folder with a unique and descriptive name for your controller in software/python/simple_pendulum/trajectory_optimization.
2. Create a `README.md` file in your controller directory in which you explain how the method works and how it can be used.
3. Create a `requirements.txt` file where you list required packages that are not listed in the main requirements file of the project.
4. Either
   (a) write a controller class which makes it possible to execute your policy in simulation and on the real system. Make sure it inherits from the AbstractController class.
   (b) export your trajectory to a csv file and save it at data/trajectories. The columns of the csv file should be: time, position, velocity, control_inputs. The first line is reserved for a header. In this format the csv file can be simulated or applied to the real system with the open loop controller class

If you want to contribute software in another programming language besides python, create a new top level directory named after the programming language of your choice (eg, `./c++`) and try to follow the structure of the Python code as closely as possible, that will aid us when looking through it.

If you discover bugs, have feature requests, or want to improve the documentation, please open an issue at the issue tracker of the project.

If you want to contribute code, please open a pull request via GitHub by forking the project, committing changes to your fork, and then opening a pull request from your forked branch to the main branch of gmr.

CHAPTER

# FOURTEEN

# API REFERENCE

CHAPTER

# FIFTEEN

# ANALYSIS

CHAPTER

SIXTEEN

# BENCHMARKS

**modify_pendulum_parameter**(*par*)

Randomly modify a given parameter

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

Bases: `object`

Benchmark class

**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.

CHAPTER

# SEVENTEEN

# 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*)

CHAPTER

# EIGHTEEN

# POLICY PLOTS

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

Plot controller policy

CHAPTER

# NINETEEN

# CONTROLLERS

CHAPTER

TWENTY

# ABSTRACT CONTROLLER

Abstract controller class to which all controller classes have to adhere.

**class AbstractController**

Bases: 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.

**Parameters**

**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]

CHAPTER

# TWENTYONE

## RETURNS

**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.

CHAPTER

# TWENTYTWO

# PARAMETERS

**x0: array like**
The start state of the pendulum

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

CHAPTER

# TWENTYTHREE

# PARAMETERS

**x: array like**
The desired goal state of the controller

**ak80_6** (*controller*, *kp=0.0*, *kd=0.0*, *torque_limit=1.0*, *dt=0.005*, *tf=10.0*, *motor_id=1*, *motor_type='AK80_6_V1p1'*, *can_port='can0'*)

CHAPTER

TWENTYFOUR

# MOTOR CONTROL LOOP

The motor control loop only contains the minimum of code necessary to send commands to and receive measurement data from the motor control board in real time over CAN bus. It specifies the outgoing CAN port and the CAN ID of the motor on the CAN bus and transfers this information to the motor driver. It furthermore requires the following arguments:

The return values start, end and meas_dt are required to monitor if desired and measured time steps match.

## 24.1 Parameters

**controller**
: [controller object inheriting from the abstract controller class] Calls the controller, which executes the respective control policy and returns the input torques

**kp**
: [float, default=0.0] Weight for position control

**kd**
: [float, default=0.0] Weight for velocity control

**torque_limit**
: [float default=1.0] torque limit for the motor [Nm]

**dt**
: [float, default=0.005] time step in control loop [s]

**tf**
: [float, default=10.0] length of the experiment [s]

**motor_id**
: [int, default=1] id of the used motor

**motor_type**
: [string, default='AK80_6_V1p1'] type of the motor

**can_port**
: [string, default='can0'] can port for motor communication

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

## 24.2 Returns

**start**
[float,] start time of experiment

**end**
[float] end time of experiment

**meas_dt**
[float] measured time step

**data_dict**
[dict] dictionary containing the recorded data

CHAPTER

TWENTYFIVE

# DEEP DETERMINISTIC POLICY GRADIENT CONTROL

CHAPTER

TWENTYSIX

# DDPG CONTROLLER

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

Bases: *AbstractController*

DDPG controller class

**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.

**Parameters**

**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]

CHAPTER

# TWENTYSEVEN

## RETURNS

**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*)

CHAPTER

TWENTYEIGHT

# ENERGY SHAPING CONTROL

CHAPTER

# TWENTYNINE

# ENERGY SHAPING CONTROLLER

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

Bases: `AbstractController`

Controller which swings up the pendulum by regulating its energy.

**Parameters**

**mass**
[float, default=1.0] mass of the pendulum [kg]

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

**damping**
[float, default=0.1] damping factor of the pendulum [kg m/s]

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

**torque_limit**
[float, default=2.0] torque limit of the motor [Nm]

**k**
[float, default=1.0] the weight determining the output torque with respect to the current energy level.

**set_goal**(*x*)
Set the goal for the controller. This function calculates the energy of the goal state.

CHAPTER

THIRTY

# PARAMETERS

**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

CHAPTER

# THIRTYONE

# PARAMETERS

**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)

CHAPTER

THIRTYTWO

## RETURNS

**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*, *coulomb_fric=0.0*, *gravity=9.81*, *torque_limit=inf*, *k=1.0*, *Q=array([[10, 0], [0, 1]])*, *R=array([[1]])*, *compute_RoA=False*)

Bases: *AbstractController*

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

CHAPTER

# THIRTYTHREE

## PARAMETERS

**mass**
: [float, default=1.0] mass of the pendulum [kg]

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

**damping**
: [float, default=0.1] damping factor of the pendulum [kg m/s]

**coulomb_fric**
: [float, default=0.0] friction term, (independent of magnitude of velocity), unit: Nm

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

**torque_limit**
: [float, default=np.inf] the torque_limit of the pendulum actuator

**k**
: [float, default=1.0] (energy controller) the weight determining the output torque with respect to the current energy level.

**Q**
: [array-like, default=np.diag(10, 1)] (LQR controller) the state cost matrix, np.shape(Q) = (2,2)

**R**
: [array-like, default=np.array([[1]])] (LQR controller) the control cost matrix, np.shape(R) = (1,1)

**compute_RoA**
: [bool, default=False] (LQR controller) whether to compute the region of attraction of the LQR controller (requires drake)

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

## 33.1 Parameters

**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

## 33.2 Parameters

**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

## 33.3 Returns

**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]

CHAPTER

# THIRTYFOUR

# UNIT TESTS

**class Test**(*methodName='runTest'*)

Bases: `TestCase`

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

**epsilon = 0.2**

**test_0_energy_shaping_swingup**()

CHAPTER

# THIRTYFIVE

# CONTROLLERS

CHAPTER

THIRTYSIX

# GRAVITY COMPENSATION CONTROL

CHAPTER

# THIRTYSEVEN

# GRAVITY COMPENSATION CONTROLLER

**class GravityCompController**(*params*)

Bases: *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.

**Parameters**

**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]

CHAPTER

# THIRTYEIGHT

## RETURNS

**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]

CHAPTER

THIRTYNINE

# ILQR MODEL PREDICTIVE CONTROL

CHAPTER

FORTY

# ILQR MPC CONTROLLER

**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: *AbstractController*

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

CHAPTER

# FORTYONE

## PARAMETERS

**mass**
[float, default=1.0] mass of the pendulum [kg]

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

**damping**
[float, default=0.1] damping factor of the pendulum [kg m/s]

**coulomb_friction**
[float, default=0.0] coulomb_friciton term of the pendulum

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

**inertia**
[float, default=0.125] inertia of the pendulum

**dt**
[float, default=0.01] timestep of the simulation

**n**
[int, default=50] number of timnesteps the controller optimizes ahead

**max_iter**
[int, default=1] optimization iterations the alogrithm makes at every timestep

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

**sCu**
[float, default=10.0] running cost weight for the control input u

**sCp**
[float, default=0.001] running cost weight for the position error

**sCv**
[float, default=0.001] running cost weight for the velocity error

**sCen**
[float, default=0.0] running cost weight for the energy error

**fCp**
: [float, default=1000.0] final cost weight for the position error

**fCv**
: [float, default=10.0] final cost weight for the velocity error

**fCen**
: [float, default=300.0] final cost weight for the energy error

**dynamics**
: [string, default="runge_kutta"] string that selects the integrator to be used for the simulation options are: "euler", "runge_kutta"

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

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

## 41.1 Parameters

**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

## 41.2 Parameters

**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

## 41.3 Parameters

**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

## 41.4 Parameters

**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.

## 41.5 Parameters

**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

## 41.6 Parameters

**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)

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

## 41.7 Returns

**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]

CHAPTER

# FORTYTWO

## UNIT TESTS

**class Test**(*methodName='runTest'*)

Bases: `TestCase`

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

**epsilon = 0.2**

**test_0_iLQR_MPC_swingup_nx2**()

**test_1_iLQR_MPC_swingup_nx3**()

CHAPTER

FORTYTHREE

# LINEAR QUADRATIC REGULATOR (LQR)

CHAPTER

# FORTYFOUR

# LQR SOLVER

Adapted from Mark Wilfried Mueller

**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

CHAPTER

FORTYFIVE

# LQR CONTROLLER

**class LQRController**(*mass=1.0*, *length=0.5*, *inertia=None*, *damping=0.1*, *coulomb_fric=0.0*, *gravity=9.81*, *torque_limit=inf*, *Q=array([[10, 0], [0, 1]])*, *R=array([[1]])*, *compute_RoA=False*)

Bases: *AbstractController*

Controller which stabilizes the pendulum at its instable fixpoint. Parameters ———- mass : float, default=1.0

mass of the pendulum [kg]

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

**damping**
[float, default=0.1] damping factor of the pendulum [kg m/s]

**coulomb_fric**
[float, default=0.0] friction term, (independent of magnitude of velocity), unit: Nm

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

**torque_limit**
[float, default=np.inf] the torque_limit of the pendulum actuator

**Q**
[array-like, default=np.diag(10, 1)] the state cost matrix, np.shape(Q) = (2,2)

**R**
[array-like, default=np.array([[1]])] the control cost matrix, np.shape(R) = (1,1)

**compute_RoA**
[bool, default=False] whether to compute the region of attraction of the LQR controller (requires drake)

**set_goal**(*goal*)

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

CHAPTER

# FORTYSIX

## PARAMETERS

**x: array like**
The desired goal state of the controller

**set_clip**()

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

The function to compute the control input for the pendulum actuator Parameters ———- 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)

CHAPTER

# FORTYSEVEN

## RETURNS

**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]

CHAPTER

FORTYEIGHT

# UNIT TESTS

**class Test**(*methodName='runTest'*)

Bases: `TestCase`

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

**epsilon = 0.01**

**test_0_LQR_stabilization**()

**transform**(*q*)

**class analytic_roa**(*mass=1.0*, *length=0.5*, *damping=0.1*, *gravity=9.81*, *torque_limit=inf*, *Q_11=1*, *Q_22=1*, *R=1*)

Bases: `object`

**tstar**(*q*)

**u**(*t*, *q*)

**satisfies_theory**(*y*)

**satisfies_theory_full**(*y*)

**class najafi_oracle**(*plant*, *controller*, *n=100000*)

Bases: `object`

**query**(*q*)

**get_ellipse_params**(*rho*, *M*)

Returns ellipse params (excl center point)

**get_ellipse_patch**(*px*, *py*, *rho*, *M*, *alpha_val=1*, *linec='red'*, *facec='none'*, *linest='solid'*)

return an ellipse patch

**plot_ellipse**(*px*, *py*, *rho*, *M*, *save_to=None*, *show=True*)

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

**najafi_based_sampling**(*plant*, *controller*, *n=10000*, *rho0=100*, *M=None*, *x_star=array([3.14159265, 0.0])*)

Estimate the RoA for the closed loop dynamics using the method introduced in Najafi, E., Babuška, R. & Lopes, G.A.D. A fast sampling method for estimating the domain of attraction. Nonlinear Dyn 86, 823–834 (2016). https://doi.org/10.1007/s11071-016-2926-7

CHAPTER

# FORTYNINE

## PARAMETERS

**plant**
: [simple_pendulum.model.pendulum_plant] configured pendulum plant object

**controller**
: [simple_pendulum.controllers.lqr.lqr_controller] configured lqr controller object

**n**
: [int, optional] number of samples, by default 100000

**rho0**
: [int, optional] initial estimate of rho, by default 10

**M**
: [np.array, optional] M, such that x_barT M x_bar is the Lyapunov fct. by default None, and controller.S is used

**x_star**
: [np.array, optional] nominal position (fixed point of the nonlinear dynamics)

CHAPTER

FIFTY

# RETURNS

**rho**
: [float] estimated value of rho

**M**
: [np.array] M

**SOSequalityConstrained** (*pendulum*, *controller*)

**Estimate the RoA for the closed loop dynamics using the method described by Russ Tedrake in "Underactuated Robotics: Algorithms for Walking, Running, Swimming, Flying, and Manipulation",**

Course Notes for MIT 6.832, 2022, "http://underactuated.mit.edu", sec. 9.3.2: "The equality-constrained formulation". This is discussed a bit more by Shen Shen and Russ Tedrake in "Sampling Quotient-Ring Sum-of-Squares Programs for Scalable Verification of Nonlinear Systems", Proceedings of the 2020 59th IEEE Conference on Decision and Control (CDC) , 2020., http://groups.csail.mit.edu/robotics-center/public_papers/Shen20.pdf, pg. 2-3.

CHAPTER

# FIFTYONE

# PARAMETERS

**pendulum**
: [simple_pendulum.model.pendulum_plant] configured pendulum plant object

**controller**
: [simple_pendulum.controllers.lqr.lqr_controller] configured lqr controller object

CHAPTER

# FIFTYTWO

## RETURNS

**rho**
: [float] estimated value of rho

**S**
: [np.array] S matrix from the lqr controller

**SOSlineSearch** (*pendulum*, *controller*)

Simple line search(bisection method) on rho that search for the maximum rho which satisfies the Lyapunov conditions in order to obtain an estimate of the RoA for the closed loop dynamics.

CHAPTER

# FIFTYTHREE

# PARAMETERS

**pendulum**
: [simple_pendulum.model.pendulum_plant] configured pendulum plant object

**controller**
: [simple_pendulum.controllers.lqr.lqr_controller] configured lqr controller object

CHAPTER

# FIFTYFOUR

## RETURNS

**rho**
: [float] estimated value of rho

**S**
: [np.array] S matrix from the lqr controller

**direct_sphere**(*d*, *r_i=0*, *r_o=1*)

Direct Sampling from the d Ball based on Krauth, Werner. Statistical Mechanics: Algorithms and Computations. Oxford Master Series in Physics 13. Oxford: Oxford University Press, 2006. page 42

CHAPTER

# FIFTYFIVE

## PARAMETERS

**d**
: [int] dimension of the ball

**r_i**
: [int, optional] inner radius, by default 0

**r_o**
: [int, optional] outer radius, by default 1

CHAPTER

# FIFTYSIX

## RETURNS

**np.array**
random vector directly sampled from the solid d Ball

**sample_from_ellipsoid**(*M*, *rho*, *r_i=0*, *r_o=1*)
sample directly from the ellipsoid defined by xT M x.

CHAPTER

# FIFTYSEVEN

## PARAMETERS

**M**
: [np.array] Matrix M such that xT M x leq rho defines the hyperellipsoid to sample from

**rho**
: [float] rho such that xT M x leq rho defines the hyperellipsoid to sample from

**r_i**
: [int, optional] inner radius, by default 0

**r_o**
: [int, optional] outer radius, by default 1

CHAPTER

# FIFTYEIGHT

## RETURNS

**np.array**
random vector from within the hyperellipsoid

**quad_form**(*M*, *x*)

Helper function to compute quadratic forms such as x^TMx

**vol_ellipsoid**(*rho*, *M*)

Calculate the Volume of a Hyperellipsoid Volume of the Hyperllipsoid according to https://math.stackexchange.com/questions/332391/volume-of-hyperellipsoid/332434 Intuition: https://textbooks.math.gatech.edu/ila/determinants-volumes.html Volume of n-Ball https://en.wikipedia.org/wiki/Volume_of_an_n-ball

**rhoVerification**(*rho*, *pendulum*, *controller*)

**SOS Verification of the Lyapunov conditions for a given rho in order to obtain an estimate of the RoA for the closed loop dynamics.**
This method is described by Russ Tedrake in "Underactuated Robotics: Algorithms for Walking, Running, Swimming, Flying, and Manipulation", Course Notes for MIT 6.832, 2022, "http://underactuated.mit.edu", sec. 9.3.2: "Basic region of attraction formulation".

CHAPTER

# FIFTYNINE

# PARAMETERS

**rho: float**
value of rho to be verified

**pendulum**
[simple_pendulum.model.pendulum_plant] configured pendulum plant object

**controller**
[simple_pendulum.controllers.lqr.lqr_controller] configured lqr controller object

CHAPTER

SIXTY

## RETURNS

**result**
[boolean] result of the verification

**class PendulumPlantApprox**(*mass=1.0*, *length=0.5*, *damping=0.1*, *gravity=9.81*, *coulomb_fric=0.0*, *inertia=None*, *torque_limit=inf*, *taylorApprox_order=1*)

Bases: `object`

The PendulumPlantApprox class contains the taylor-approximated dynamics of the simple pendulum.

**The state of the pendulum in this class is described by**
state = [angle, angular velocity] (array like with len(state)=2) in units: rad and rad/s

The zero state of the angle corresponds to the pendulum hanging down. The plant expects an actuation input (tau) either as float or array like in units Nm. (in which case the first entry is used (which should be a float))

CHAPTER

# SIXTYONE

## PARAMETERS

**mass**
[float, default=1.0] pendulum mass, unit: kg

**length**
[float, default=0.5] pendulum length, unit: m

**damping**
[float, default=0.1] damping factor (proportional to velocity), unit: kg*m/s

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

**coulomb_fric**
[float, default=0.0] friction term, (independent of magnitude of velocity), unit: Nm

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

**torque_limit: float, default=np.inf**
maximum torque that the motor can apply, unit: Nm

**taylorApprox_order: int, default=1**
order of the taylor approximation of the sine term

`forward_dynamics` (*state*, *tau*)
Computes forward dynamics

### 61.1 Parameters

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

**tau**
[float] motor torque, unit: Nm

## 61.2 Returns

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

**rhs** (*t*, *state*, *tau*)

Computes the integrand of the equations of motion.

## 61.3 Parameters

**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

## 61.4 Returns

**res**

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

CHAPTER

SIXTYTWO

# OPEN LOOP CONTROL

CHAPTER

# SIXTYTHREE

# OPEN LOOP CONTROLLER

**class OpenLoopController**(*data_dict*)

Bases: *AbstractController*

Controller acts on a predefined trajectory.

CHAPTER

# SIXTYFOUR

## PARAMETERS

**data_dict**
: [dictionary] a dictionary containing the trajectory to follow 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

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

### 64.1 Parameters

**x0: array like**
: The start state of the pendulum

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

### 64.2 Parameters

**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.

## 64.3 Parameters

**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]

## 64.4 Returns

**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: `AbstractController`

Controller acts on a predefined trajectory. Switches to lqr control when in its region of attraction.

CHAPTER

SIXTYFIVE

# PARAMETERS

**data_dict**
[dictionary] a dictionary containing the trajectory to follow 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

**mass**
[float, default=1.0] mass of the pendulum [kg]

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

**damping**
[float, default=0.1] damping factor of the pendulum [kg m/s]

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

**torque_limit**
[float, default=np.inf] the torque_limit of the pendulum actuator

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

## 65.1 Parameters

**x0: array like**
The start state of the pendulum

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

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

## 65.2 Parameters

**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.

## 65.3 Parameters

**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

## 65.4 Returns

**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]

CHAPTER

SIXTYSIX

# PROPORTIONAL-INTEGRAL-DERIVATIVE (PID) CONTROL

CHAPTER

# SIXTYSEVEN

# PID CONTROLLER

**class PIDController**(*data_dict*, *Kp*, *Ki*, *Kd*, *use_feed_forward=True*)

Bases: `AbstractController`

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

CHAPTER

SIXTYEIGHT

# PARAMETERS

**data_dict**
[dictionary] a dictionary containing the trajectory to follow 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

**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

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

## 68.1 Parameters

**`x0: array like`**
The start state of the pendulum

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

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

## 68.2 Parameters

**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.

## 68.3 Parameters

**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]

## 68.4 Returns

**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]

CHAPTER

# SIXTYNINE

# SOFT ACTOR CRITIC (SAC) CONTROL

CHAPTER

SEVENTY

# SAC CONTROLLER

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

Bases: *AbstractController*

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

CHAPTER

# SEVENTYONE

# PARAMETERS

**model_path**
[string] path to the trained model in zip format

**torque_limit**
[float] torque limit of the pendulum. The output of the model will be scaled with this number

**use_symmetry**
[bool] whether to use the left/right symmetry of 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

## 71.1 Parameters

**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)

## 71.2 Returns

**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)

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

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

**`get_observation`**(*state*)

CHAPTER

SEVENTYTWO

# TIME-VARYING LINEAR QUADRATIC REGULATOR (TVLQR)

CHAPTER

# SEVENTYTHREE

# TVLQR CONTROLLER

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

Bases: `AbstractController`

Controller acts on a predefined trajectory.

CHAPTER

SEVENTYFOUR

PARAMETERS

**data_dict**
[dictionary] a dictionary containing the trajectory to follow 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

**mass**
[float, default=1.0] mass of the pendulum [kg]

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

**damping**
[float, default=0.1] damping factor of the pendulum [kg m/s]

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

**torque_limit**
[float, default=np.inf] the torque_limit of the pendulum actuator

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

## 74.1 Parameters

**x0: array like**
The start state of the pendulum

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

## 74.2 Parameters

**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.

## 74.3 Parameters

**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]

## 74.4 Returns

**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]

**set_Qf** (*Qf*)
This function is useful only for RoA purposes. Used to set the final S-matrix of the tvlqr controller.

## 74.5 Parameters

**Qf**
[matrix] the S-matrix from time-invariant RoA estimation around the up-right position.

**`get_ellipse_params`** (*rho*, *M*)
Returns ellipse params (excl center point)

**`get_ellipse_patch`** (*px*, *py*, *rho*, *M*, *alpha_val=1*, *linec='red'*, *facec='none'*, *linest='solid'*)
return an ellipse patch

**`plot_ellipse`** (*px*, *py*, *rho*, *M*, *save_to=None*, *show=True*)

**`plotFirstLastEllipses`** (*funnel_path*, *traj_path*)

**`plotRhoEvolution`** (*funnel_path*, *traj_path*)

**`plotFunnel3d`** (*csv_path*, *traj_path*, *ax*)
Function to draw a discrete 3d funnel plot. Basically we are plotting a 3d ellipse patch in each knot point. Parameters ———- rho : np.array

array that contains the estimated rho value for all the knot points

**S: np.array**
array of matrices that define ellipses in all the knot points, from tvlqr controller.

**x0: np.array**
pre-computed nominal trajectory

**time: np.array**
time array related to the nominal trajectory

**ax: matplotlib.axes**
axes of the plot where we want to add the 3d funnel plot, useful in the verification function.

**`plotFunnel`** (*funnel_path*, *traj_path*, *ax=None*)
Function to draw a continue 2d funnel plot. This implementation makes use of the convex hull concept as done in the MATLAB code of the Robot Locomotion Group (https://groups.csail.mit.edu/locomotion/software.html). Parameters ———- rho : np.array

array that contains the estimated rho value for all the knot points

**S: np.array**
array of matrices that define ellipses in all the knot points, from tvlqr controller.

**x0: np.array**
pre-computed nominal trajectory

**time: np.array**
time array related to the nominal trajectory

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

**`TVrhoVerification`** (*pendulum*, *controller*, *funnel_path*, *traj_path*, *nSimulations*, *ver_idx*)

Function to verify the time-variant RoA estimation. This implementation permitts also to choose which knot has to be tested. Furthermore the a 3d funnel plot has been implemented.

CHAPTER

# SEVENTYFIVE

# PARAMETERS

**pendulum: simple_pendulum.model.pendulum_plant**
configured pendulum plant object

**controller: simple_pendulum.controllers.tvlqr.tvlqr**
configured tvlqr controller object

**x0_t: np.array**
pre-computed nominal trajectory

**time: np.array**
time array related to the nominal trajectory

**nSimulations: int**
number of simulations for the verification

**ver_idx: int**
knot point to be verified

**`funnel2DComparison`** (*csv_pathFunnelSos*, *csv_pathFunnelProb*, *traj_path*)

**`rhoComparison`** (*csv_pathFunnelSos*, *csv_pathFunnelProb*)

**`TVprobRhoComputation`** (*pendulum*, *controller*, *x0_t*, *time*, *N*, *nSimulations*, *rhof*)

Function to perform the time-variant RoA estimation. This implementation has been inpired by "Feedback-Motion-Planning with Simulation-Based LQR-Trees" by Philipp Reist, Pascal V. Preiswerk and Russ Tedrake (https://groups.csail.mit.edu/robotics-center/public_papers/Reist15.pdf).

CHAPTER

# SEVENTYSIX

# PARAMETERS

**pendulum: simple_pendulum.model.pendulum_plant**
configured pendulum plant object

**controller: simple_pendulum.controllers.tvlqr.tvlqr**
configured tvlqr controller object

**x0_t: np.array**
pre-computed nominal trajectory

**time: np.array**
time array related to the nominal trajectory

**N: int**
number of considered knot points

**nSimulations: int**
max number of simulations for each ellipse estimation

**rhof: float**
final rho value, from the time-invariant RoA estimation

CHAPTER

# SEVENTYSEVEN

## RETURNS

**rho**
: [np.array] array that contains the estimated rho value for all the knot points

**S: np.array**
: array that contains the S matrix in each knot point

`TVsosRhoComputation` (*pendulum*, *controller*, *time*, *N*, *rhof* )

Bilinear alternationturn used for the SOS funnel estimation.

CHAPTER

# SEVENTYEIGHT

# PARAMETERS

**pendulum: simple_pendulum.model.pendulum_plant**
configured pendulum plant object

**controller: simple_pendulum.controllers.tvlqr.tvlqr**
configured tvlqr controller object

**time: np.array**
time array related to the nominal trajectory

**N: int**
number of considered knot points

**rhof: float**
final rho value, from the time-invariant RoA estimation

CHAPTER

# SEVENTYNINE

## RETURNS

**rho_t**
[np.array] array that contains the estimated rho value for all the knot points

**S: np.array**
array that contains the S matrix in each knot point

`direct_sphere`(*d*, *r_i=0*, *r_o=1*)

Direct Sampling from the d Ball based on Krauth, Werner. Statistical Mechanics: Algorithms and Computations. Oxford Master Series in Physics 13. Oxford: Oxford University Press, 2006. page 42

CHAPTER

EIGHTY

# PARAMETERS

**d**
: [int] dimension of the ball

**r_i**
: [int, optional] inner radius, by default 0

**r_o**
: [int, optional] outer radius, by default 1

CHAPTER

# EIGHTYONE

## RETURNS

**np.array**
random vector directly sampled from the solid d Ball

**sample_from_ellipsoid**(*M*, *rho*, *r_i=0*, *r_o=1*)
sample directly from the ellipsoid defined by xT M x.

CHAPTER

# EIGHTYTWO

## PARAMETERS

**M**
: [np.array] Matrix M such that xT M x leq rho defines the hyperellipsoid to sample from

**rho**
: [float] rho such that xT M x leq rho defines the hyperellipsoid to sample from

**r_i**
: [int, optional] inner radius, by default 0

**r_o**
: [int, optional] outer radius, by default 1

CHAPTER

# EIGHTYTHREE

## RETURNS

**np.array**
random vector from within the hyperellipsoid

**quad_form** (*M*, *x*)

Helper function to compute quadratic forms such as x^TMx

**projectedEllipseFromCostToGo** (*s0Idx*, *s1Idx*, *rho*, *M*)

Returns ellipses in the plane defined by the states matching the indices s0Idx and s1Idx for funnel plotting.

**getEllipseContour** (*S*, *rho*, *xg*)

Returns a certain number(nSamples) of sampled states from the contour of a given ellipse.

CHAPTER

# EIGHTYFOUR

## PARAMETERS

**S**
: [np.array] Matrix S that define one ellipse

**rho**
: [np.array] rho value that define one ellipse

**xg**
: [np.array] center of the ellipse

CHAPTER

EIGHTYFIVE

## RETURNS

**c**
: [np.array] random vector of states from the contour of the ellipse

**`TVrhoSearch`** (*pendulum*, *controller*, *knot*, *time*, *h_map*, *rho_t*)
: V step of the bilinear alternationturn used in the SOS funnel estimation.

CHAPTER

# EIGHTYSIX

## PARAMETERS

**pendulum: simple_pendulum.model.pendulum_plant**
configured pendulum plant object

**controller: simple_pendulum.controllers.tvlqr.tvlqr**
configured tvlqr controller object

**knot: int**
number of considered knot point

**time: np.array**
time array related to the nominal trajectory

**h_map: Dict[pydrake.symbolic.Monomial, pydrake.symbolic.Expression]**
map of the coefficients of the multiplier obtained from the multiplier step

**rho_t: np.array**
array that contains the evolving estimation of the rho values for each knot point

CHAPTER

# EIGHTYSEVEN

## RETURNS

**fail**
: [boolean] gives info about the correctness of the optimization problem

**rho_opt: float**
: optimized rho value for this knot point

`TVmultSearch` (*pendulum*, *controller*, *knot*, *time*, *rho_t*)

Multiplier step of the bilinear alternationturn used in the SOS funnel estimation.

CHAPTER

# EIGHTYEIGHT

# PARAMETERS

**pendulum: simple_pendulum.model.pendulum_plant**
configured pendulum plant object

**controller: simple_pendulum.controllers.tvlqr.tvlqr**
configured tvlqr controller object

**knot: int**
number of considered knot point

**time: np.array**
time array related to the nominal trajectory

**rho_t: np.array**
array that contains the evolving estimation of the rho values for each knot point

CHAPTER

# EIGHTYNINE

## RETURNS

**fail**
: [boolean] gives info about the correctness of the optimization problem

**h_map: Dict[pydrake.symbolic.Monomial, pydrake.symbolic.Expression]**
: map of the coefficients of the multiplier obtained from the multiplier step

**eps: float**
: optimal cost of the optimization problem that can be useful in the V step

CHAPTER

# NINETY

## MODEL

CHAPTER

# NINETYONE

## PARAMETERS

**get_params**(*params_path*)

Retrieve parameters from a yaml file with name `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*)

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

**calc_k_e**(*k_v*)

**calc_k_t_from_k_m**(*k_m*, *resist*)

**calc_k_t_from_k_v**(*k_v*)

CHAPTER

# NINETYTWO

# PENDULUM PLANT

**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`

The PendulumPlant class contains the kinematics and dynamics of the simple pendulum.

**The state of the pendulum in this class is described by**

state = [angle, angular velocity] (array like with len(state)=2) in units: rad and rad/s

The zero state of the angle corresponds to the pendulum hanging down. The plant expects an actuation input (tau) either as float or array like in units Nm. (in which case the first entry is used (which should be a float))

CHAPTER

# NINETYTHREE

# PARAMETERS

**mass**
[float, default=1.0] pendulum mass, unit: kg

**length**
[float, default=0.5] pendulum length, unit: m

**damping**
[float, default=0.1] damping factor (proportional to velocity), unit: kg*m/s

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

**coulomb_fric**
[float, default=0.0] friction term, (independent of magnitude of velocity), unit: Nm

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

**torque_limit: float, default=np.inf**
maximum torque that the motor can apply, unit: Nm

`load_params_from_file` (*filepath*)
Load the pendulum parameters from a yaml file.

## 93.1 Parameters

**filepath**
[string] path to yaml file

`forward_kinematics` (*pos*)
Computes the forward kinematics.

## 93.2 Parameters

pos : float, angle of the pendulum

## 93.3 Returns

**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

## 93.4 Parameters

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

## 93.5 Returns

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

**`forward_dynamics`**(*state*, *tau*)
: Computes forward dynamics

## 93.6 Parameters

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

**tau**
: [float] motor torque, unit: Nm

## 93.7 Returns

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

**`inverse_dynamics`** (*state*, *accn*)

Computes inverse dynamics

## 93.8 Parameters

**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

## 93.9 Returns

**tau**
: [float] motor torque, unit: Nm

**`rhs`** (*t*, *state*, *tau*)

Computes the integrand of the equations of motion.

## 93.10 Parameters

**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

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

## 93.11 Returns

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

**`potential_energy`** (*state*)

**`kinetic_energy`** (*state*)

**`total_energy`** (*state*)

CHAPTER

# NINETYFOUR

# UNIT TESTS

**class Test**(*methodName='runTest'*)

Bases: `TestCase`

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

**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

CHAPTER

# NINETYFIVE

## REINFORCEMENT LEARNING

CHAPTER

NINETYSIX

# DEEP DETERMININSTIC POLICY GRADIENT (DDPG) TRAINING

CHAPTER

# NINETYSEVEN

## AGENT

**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*)

CHAPTER

# NINETYEIGHT

## DDPG TRAINER

**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.

CHAPTER

# NINETYNINE

## 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.

CHAPTER

# PARAMETER

**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*)

CHAPTER

ONE

# MODELS

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

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

CHAPTER

TWO

# NOISE

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

Bases: `object`

**reset**()

CHAPTER

THREE

# REPLAY BUFFER

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

Bases: `object`

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

CHAPTER

FOUR

# PARAMETERS

**max_size: int**
maximum number of experiences to store in the repleay buffer. When adding experiences beyond this limit, the first entry is deleted.

**num_state: int**
the dimension of the state space

**num_actions: int**
the dimension of the action space

**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)

## 104.1 Parameters

**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.

## 104.2 Parameters

**batch_size: int**
number of samples in the returned batch

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

## 104.3 Returns

**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.

CHAPTER

FIVE

# SOFT ACTOR CRITIC (SAC) TRAINING

CHAPTER

SIX

# SAC TRAINER

**class sac_trainer**(*log_dir='sac_training'*)

Bases: `object`

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

CHAPTER

SEVEN

# PARAMETER

**log_dir**
[string, default="sac_training"] path to directory where results and log data will be stored

**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.

## 107.1 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.

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

## 107.2 Parameter

**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.

## 107.3 Parameters

**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

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

`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").

## 107.4 Parameter

**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

CHAPTER

# EIGHT

# SIMULATION

CHAPTER

NINE

# GYM ENVIRONMENT

**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: `Env`

An environment for reinforcement learning.

CHAPTER

# PARAMETERS

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 are: 'continuous', 'discrete', 'soft_binary',

'soft_binary_with_repellor', 'open_ai_gym'

**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

**step**(*action*)

Take a step in the environment.

## 110.1 Parameters

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

## 110.2 Returns

**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])

## 110.3 Parameters

**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

## 110.4 Returns

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

## 110.5 Raises:

**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.

## 110.6 Parameters

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

## 110.7 Returns

**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.

## 110.8 Parameters

**obs**
[array-like] observation as received from get_observation

## 110.9 Returns

**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.

## 110.10 Parameters

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

## 110.11 Returns

**reward**
[float] the reward for swinging up

## 110.12 Raises

**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.

## 110.13 Returns

**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.

## 110.14 Parameters

**obs**
[array-like] observation as received from get_observation

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

## 110.15 Returns

**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.

## 110.16 Parameters

**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.

## 110.17 Returns

**passed**
[bool] Whether the rewards pass the validation test

CHAPTER

# ONE

# SIMULATOR

**class Simulator**(*plant*)

Bases: `object`

Simulator class, can simulate and animate the pendulum

CHAPTER

TWO

# PARAMETERS

**plant: plant object**
(e.g. PendulumPlant from simple_pendulum.models.pendulum_plant.py)

`set_state` (*time*, *x*)
set the state of the pendulum plant

## 112.1 Parameters

**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

## 112.2 Returns

**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

## 112.3 Parameters

**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

## 112.4 Parameters

**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

## 112.5 Returns

array-like : the Euler integrand

**`runge_integrator`** (*t*, *y*, *dt*, *tau*)
Runge-Kutta integrator for the simulated plant

## 112.6 Parameters

**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

## 112.7 Returns

array-like : the Runge-Kutta integrand

**step**(*tau*, *dt*, *integrator='runge_kutta'*)

Performs a single step of the plant.

## 112.8 Parameters

**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.

## 112.9 Parameters

**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

## 112.10 Returns

**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.

## 112.11 Parameters

**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

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

## 112.12 Returns

**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*)

CHAPTER

THREE

# TRAJECTORY OPTIMIZATION

CHAPTER

# FOUR

# DDP

CHAPTER

FIVE

# DIRECT COLLOCATION

CHAPTER

SIX

# DIRECT COLLOCATION CALCULATOR

**class DirectCollocationCalculator**

Bases: `object`

Class to calculate a control trajectory with direct collocation.

**init_pendulum**(*mass=0.57288*, *length=0.5*, *damping=0.15*, *gravity=9.81*, *torque_limit=2.0*)

Initialize the pendulum parameters.

CHAPTER

SEVEN

# PARAMETERS

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

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

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

**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

`compute_trajectory`(*N=21*, *max_dt=0.5*, *start_state=[0.0, 0.0]*, *goal_state=[3.141592653589793, 0.0]*, *initial_x_trajectory=None*)

Compute a trajectory from a start state to a goal state for the pendulum.

CHAPTER

EIGHT

# PARAMETERS

**N**
[int, default=21] number of collocation points

**max_dt**
[float, default=0.5] maximum allowed timestep between collocation points

**start_state**
[array_like, default=[0.0, 0.0]] the start state of the pendulum [position, velocity]

**goal_state**
[array_like, default=[np.pi, 0.0]] the goal state for the trajectory

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

CHAPTER

NINE

## RETURNS

**x_trajectory**
[pydrake.trajectories.PiecewisePolynomial] trajectory in state space

**dircol**
[pydrake.systems.trajectory_optimization.DirectCollocation] DirectCollocation pydrake object

**result**
[pydrake.solvers.mathematicalprogram.MathematicalProgramResult] MathematicalProgramResult pydrake object

**plot_phase_space_trajectory**(*x_trajectory*, *save_to=None*)

Plot the computed trajectory in phase space.

CHAPTER

# PARAMETERS

**x_trajectory**
: [pydrake.trajectories.PiecewisePolynomial] the trajectory returned from the compute_trajectory function.

**save_to**
: [string, default=None] string pointing to the location where the figure is supposed to be stored. If save_to==None, the figure is not stored but shown in a window instead.

**`extract_trajectory`** (*x_trajectory*, *dircol*, *result*, *N=801*)

Extract time, position, velocity and control trajectories from the outputs of the compute_trajectory function.

CHAPTER

# ONE

# PARAMETERS

**x_trajectory**
: [pydrake.trajectories.PiecewisePolynomial] trajectory in state space

**dircol**
: [pydrake.systems.trajectory_optimization.DirectCollocation] DirectCollocation pydrake object

**result**
: [pydrake.solvers.mathematicalprogram.MathematicalProgramResult] MathematicalProgramResult pydrake object

**N**
: [int, default=801] The number of sampling points of the returned trajectories

CHAPTER

TWO

## RETURNS

**time_traj**
[array_like] the time trajectory

**theta**
[array_like] the position trajectory

**theta_dot**
[array_like] the velocity trajectory

**torque_traj**
[array_like] the control (torque) trajectory

CHAPTER

THREE

# UNIT TESTS

**class Test**(*methodName='runTest'*)

Bases: TestCase

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

**epsilon = 0.2**

**test_0_direct_collocation**()

CHAPTER

# FOUR

# ILQR

CHAPTER

FIVE

# ILQR CALCULATOR

Large parts taken from Russ Tedrake.

**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 the pydrake symbolic library.

CHAPTER

SIX

# PARAMETERS

**n_x**
[int, default=2] The size of the state space.

**n_u**
[int, default=1] The size of the control space.

**set_start** (*x0*)
Set the start state for the trajectory.

## 126.1 Parameters

**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.

## 126.2 Parameters

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

**set_stage_cost** (*stage_cost_func*)
Set the stage cost (running cost) for the ilqr optimization.

## 126.3 Parameters

**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.

## 126.4 Parameters

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

**init_derivatives**()
Initialize the derivatives of the dynamics.

**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.

## 126.5 Parameters

**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

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

## 126.6 Returns

**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

CHAPTER

# SEVEN

# SYMBOLIC

Large parts taken from Russ Tedrake.

**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.

CHAPTER

EIGHT

# PARAMETERS

**n_x**
: [int, default=2] The size of the state space.

**n_u**
: [int, default=1] The size of the control space.

**set_start** (*x0*)
: Set the start state for the trajectory.

## 128.1 Parameters

**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.

## 128.2 Parameters

**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.

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

## 128.3 Parameters

**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.

## 128.4 Parameters

**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.

## 128.5 Parameters

**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

## 128.6 Returns

**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

CHAPTER

NINE

# PENDULUM DYNAMICS

**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*)

CHAPTER

# UNIT TESTS

**class Test**(*methodName='runTest'*)

Bases: `TestCase`

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

**epsilon = 0.2**

**test_0_iLQR_computation_nx2**()

**test_0_iLQR_computation_nx3**()

**test_0_iLQR_computation_nx2_sympy**()

**test_0_iLQR_computation_nx3_sympy**()

CHAPTER

# ONE

# UTILITIES

CHAPTER

TWO

# PARSING

`syntax()`

CHAPTER

THREE

# PERFORMANCE PROFILING

**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

CHAPTER

# FOUR

# PLOTTING

**swingup**(*save*, *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*)

CHAPTER

FIVE

# DATA PROCESSING

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

**prepare_empty** (*dt*, *tf*)

**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*)

**save** (*output_folder*, *data_dict*)

**saveFunnel** (*rho*, *S_t*, *time*, *max_dt*, *N*, *estMethod=''*)

**getEllipseFromCsv** (*csv_path*, *index*)

CHAPTER

SIX

# UNITS

**rad_to_deg**(*theta_rad*)

**deg_to_rad**(*theta_deg*)

CHAPTER

SEVEN

# FILTERS

CHAPTER

EIGHT

# FAST FOURIER TRANSFORM

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

Fast Fourier transform method

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

SciPy alternative

CHAPTER

# NINE

# LOW-PASS

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

CHAPTER

# MOVING AVERAGE

**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*)

CHAPTER

# ONE

# SAVITZKY-GOLAY

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

# PYTHON MODULE INDEX

## a



## c

## m



## r



## s



## t



## u

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

# INDEX

## E



## F



## G

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



## I



## J



## K



## L



## M

## N



## O



## P

## Q



## R



## S

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

## T



## U



## V