QuaSI - Quadrotor System Identification and Simulation Pipeline

February 25, 2025

User Manual (V1.0.a)

Jasper van Beers

Contents

1 Introduction

The quadrotor identification and simulation pipeline (QuaSI) aims to facilitate the rapid development and deployment of quadrotor aerodynamic models, be this to further our understanding of the underlying physics behind the quadrotor, or to improve safety through better controllers and reachable set analysis.

QuaSI aims to offer both convenience and flexibility to its users. To this end, the pipeline is designed to be as user-friendly as possible. While Python is necessary to run the scripts, users primarily interact with these scripts through configuration files and thus require limited (Python) programming knowledge. This also reduces the need to manipulate variables directly in the scripts themselves.

QuaSI is designed with modularity in mind such that, for those more experience with Python, they may import the relevant modules directly in their own workflow. Furthermore, for those more comfortable with MATLAB, QuaSI also offers a MATLAB port of the identified polynomial quadrotor models. A base simulation environment in Simulink is also provided to compliment the MATLAB port. A more flexible and feature-rich simulation environment is also available in Python directly.

An overview of QuaSI is depicted in fig. 1. The general procedure is to first fuse the different data sources of a given flight and synchronize the time series. Subsequently, the quadrotor states are generally filtered via an extended Kalman filter. Other variables (e.g. rotor speeds) may also be low-pass or notch-filtered if necessary. Model identification follows the data processing. In principle, the base system identification module allows for the identification of both polynomial and neural network models. Currently, however, QuaSI only fully supports the polynomial model variants as they are more portable and interpretable. These polynomial models are identified through step-wise regression, for which the candidate regressor pools are fully configurable by the users. Finally, the identified models may be deployed in either a Python or MATLAB simulation environment for synthetic data generation or (initial) controller development.

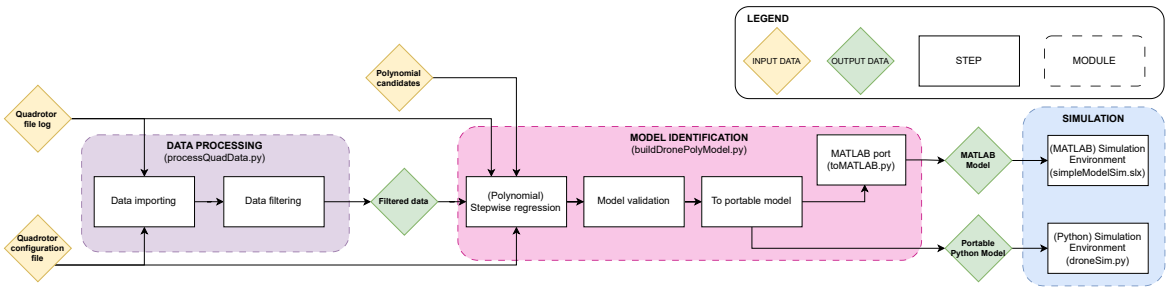


Figure 1: Schematic overview of the quadrotor identification and simulation pipeline.

In order to acquaint users with the quadrotor identification and simulation pipeline, this document aims to provide basic documentation as well as various example results at various stages of the pipeline. The remainder of the document is thus structured as follows: section 3 gives an installation guide for QuaSI and its dependencies. Section 4 provides an overview how QuaSI expects the raw data to be managed. Information on the raw data processing can be found in section 5. The identification module is outlined in section 6 whereas the simulation module is summarized in section 7. Additional information and documentation can be found in section 9. Finally, to expedite navigation in this document, section 2 provides and index of quick access links to various points in the document.

2 Quick Access Index

Below are useful links to frequently used files and sections in this document. Note, click the [Index] at the bottom of any page to return to this list!

Getting started

Configuration files

3 Installation Guide

Pre-requisites

The identification pipeline is primarily programmed in Python 3.8.12. In principle, the pipeline is compatible with Python versions 3.8+. However, as the pipeline has been verified to work with Python 3.8.12, it is the preferred installation and will be used in this guide[1]. If you do not yet have Python (3.8.12) installed, then you may download it from the official website here[2]. Alternatively, if you have anaconda installed, then you can create a new virtual environment with python 3.8.12 as below

```
conda create -n envQuaSI python==3.8.12
```

Follow the environment creation prompts, and after the new environment is created, activate it through

```
conda activate envQuaSI
```

(Recommended)[3] We moreover make use of a `pipenv` virtual environment to facilitate version consistency across the pipeline (and its users, such that the model files can be shared and run), as well as to manage the required libraries and their versions. Moreover, the pipenv should not interfere with your own local installations of python (with the exception of installing the `pipenv` library itself). `pipenv` may be installed using:

```
pip install pipenv
```

or, if you have both Python 2 and Python 3 on your system, then use

```
pip3 install pipenv
```

Those using `conda` should **use the following procedure** (in the `envQuaSI` environment) to install pipenv (**do not use conda install**!)[4]:

```
pip install virtualenv
pip install pipenv
```

Furthermore, in order to access the tool, you will need to have git installed. See the following guide[5] for details. If you would like to save animations, then you should also install ffmpeg[6] if you do not have it already. The default processor (i.e. `Pillow`) is rather slow and only supports GIFs with restricted control for the speed of animation.

After having installed these pre-requisites, the QuaSI itself may be installed. For convenience, section 3.1 provides the full terminal commands needed to install the pipeline. The following steps summarize the different installation steps.

[1] If you would instead prefer to use an alternative distribution of Python, then you may find the necessary dependencies under `[packages]` in the `Pipfile` file of the root directory in the `QuaSI` repository.

[2] URL: https://www.python.org/downloads/

[3] Those experienced with Python/conda can configure their own environments as per footnote 1.

[4] Sometimes, `conda` appears to fetch incompatible versions of `pipenv` and its dependencies, resulting in a `macOS error` regardless of your actual OS.

[5] URL: https://github.com/git-guides/install-git

[6] URL: https://ffmpeg.org/

1) Installing the main pipeline

Open the command/terminal prompt or anaconda shell and clone the QuaSI from the GitHub repository[7] and change directory to that folder. Granting access through ssh is preferred as it is more secure than using the credential helper for git (which stores your login information in plain text). Note that, if you have not already, then you will need to set up your ssh keys to access the GitHub repositories. Here are guides for creating a ssh key and linking it to your GitHub account.

Using `ssh`:

```
git clone git@github.com:Jaspervbeers/QuaSI.git
cd "QuaSI"
```

Or, using https:

```
git clone https://github.com/Jaspervbeers/QuaSI.git
cd "QuaSI"
```

If you get an `access denied` or `permission denied` error then you have not set up your ssh keys or git credentials properly, or, at least, git is using the incorrect files.

If using https, then it may be that you need to set up your credentials. Note that this method will store your login information in plain text and is therefore *not recommended* (we urge you to use ssh instead!): `git config -global credential.helper store`.

Alternatively, those encountering a `permission denied (publickey)` error when using `ssh` can try the following steps:

1. In your user `.shh` folder[8] create a file called `config` (no extension).
2. In the `config` file, write the following where *path/to* is the path to the `.shh` directory and *id_file* is either `id_ed25519` or `id_rsa` depending on your method for generating ssh keys. If these files are not in your `.ssh` folder, but you have some other `.pub` file, use that instead. Otherwise, your `.ssh` keys are located elsewhere.

   ```
   Host github.com
   IdentityFile path/to/.ssh/id_file
   IdentitiesOnly yes
   ```

3. Save the file, restart your terminal window and try cloning the repositories again

If this does not fix the issue, then please refer to the ssh GitHub page for further help.

2a) Installing pipeline repository dependencies - Windows

In the cloned `QuaSI` folder, you should see an `install-repos.bat` file. Run this script in the command window in order to install the system identification, drone identification, drone visualization, and drone simulation repositories from GitHub. Note, as with the main repository, you may need permission to access these[9].

```
install-repos.bat
```

These repositories may also be updated through (not required just after installation)

```
update-repos.bat
```

[7] Note: Some repositories are not yet released publicly as of 25-02-2025. Please contact me to request access.
[8] Typical locations are: Windows - `C:/Users/<youraccount>/.ssh` and Linux - ~/.ssh
[9] See footnote 7

2b) Installing pipeline repository dependencies - Linux

In the cloned `QuaSI` folder, you should see an `install-repos.sh` file. Run this script in the terminal window to install the system identification, drone identification, drone visualization, and drone simulation repositories from GitHub. Note, as with the main repository, you may need permission to access these[10]. Moreover, you may need to make the script executable through `chmod +x`.

```
chmod +x install-repos.sh
./install-repos.sh
```

These repositories may also be updated through (not required just after installation)

```
chmod +x update-repos.sh
./update-repos.sh
```

3) Installing Python dependencies

Now we need to set up the Python environment such that we can actually use the pipeline. As aforementioned, this is accomplished through `pipenv`. In the cloned `QuaSI` folder, you should see a `pipfile` and `pipfile.lock`. These contain the Python environment information we require.

To install the Python dependencies from these files, we run

```
pipenv install
```

The installation itself may take a while to complete. Once the installation is complete, we can initialize a shell in this virtual environment (recommended) through

```
pipenv shell
```

To confirm that the shell is active, you should see something similar to `(QuaSI-*)` at the beginning of your command line where $*$ is some combination of numbers and letters. Within this shell, we can call our Python scripts.

Alternatively, you can also select this virtual environment in your IDE (editor) of choice, if supported by the IDE. Please check whether your IDE supports this, and where to find the appropriate setting menu to change the workspace environment. Through this, you can run the scripts as normal from the IDE.

If you instead prefer to keep your workspace environment unchanged, then you can invoke relevant pipeline scripts using the `pipenv` environment through

```
pipenv run <script_name>.py
```

where `<script_name>` is the script you would like to run.

[10] See footnote 7

3.1 Full installation commands

Note that, if you use ssh keys, then you may be asked for your passphrase at various points during the installation in order to gain access to clone the GitHub repositories.

Windows

```
git clone git@github.com:Jaspervbeers/QuaSI.git
cd "QuaSI"
install-repos.bat
pipenv install
```

Linux

```
git clone git@github.com:Jaspervbeers/QuaSI.git
cd "QuaSI"
chmod +x install-repos.sh
./install-repos.sh
pipenv install
```

4 Preliminaries - Data Management

Before running any identification scripts, it is crucial inform QuaSI on how the flight data is organized. Moreover, understanding how pipeline data is managed will help once you want to add your own flight logs to the database, or create an entirely new database. Throughout QuaSI, data files are created and saved in a systematic way. We will see later that we can control, to a certain extent, where these files are saved. However, some of the subsequent folder structure and file naming scheme is kept consistent by QuaSI.

There are two core files for each quadrotor. The first is the 'experiment file log', which is essentially a log book of the flight experiments. The second is the quadrotor configuration file, which details various generic properties of the quadrotor such as the rotor layout, minimum and maximum rotor (e)RPMs, flight controller software etc.

Furthermore, for convenience, QuaSI itself can be managed by one script, `main.py` (in the root directory of `QuaSI`), which calls the relevant modules and supplies them with the required information. Of course, the individual modules may also be used within your own workflow but the `main.py` file is a good starting point to identify relevant scripts and what information they require.

4.1 Experiment File Log

This is a `.csv` 'logbook' of the quadrotor flight experiments; it details the directories of the raw data files, experiment parameters (e.g. quadrotor mass), the relevant quadrotor configuration file, and any additional comments observed during the experiment (e.g. drone lost power around two minutes into the flight). Each row of the file log corresponds to an individual flight of the quadrotor and new flights can be recorded by adding a new row-wise entry. QuaSI uses this file as a map in order to find the relevant flight data logs and other useful metadata.

Moreover, the columns of this file log may be extended with any additional information you find relevant to your experiments. As the log file is converted to a `pandas.DataFrame` object in QuaSI scripts, the ordering of the columns does not matter so long as the standard column names are kept and additional columns are unique.

For the most basic use of QuaSI, the columns outlined in table 1 are necessary in any log file. Here, the column name, description, and (expected) data type are given.

Table 1: Quadrotor experiment file log necessary column entries

Column name	Data type	Description
Flight ID	integer	The (unique) flight number of a given quadrotor flight
Rigid Body Name	string	The name assigned to the "RigidBody" constructed in the OptiTrack Motive software for external motion capturing. Only relevant if OptiTrack is used
Quadrotor	string	Name of the quadrotor
Has OT data	restricted string	Indication of whether OptiTrack data is available (or should be used) for the flight. Choice of 'Y' or 'N'
OptiTrack time offset (s)	float	Time difference, in seconds, between OptiTrack clock and Open-Jet-Facility wind tunnel clock. Only relevant for experiments with wind tunnel data. Required to synchronize measurements
Raw OptiTrack Name	string	Filename (without extension) of the OptiTrack data, if present

Raw OT Path	string	(Relative or absolute) path to the directory of the raw `.csv` OptiTrack file. Only relevant if OptiTrack data is available
Onboard Name	string	Filename (without extension) of the on-board data
OB Path	string	(Relative or absolute) path to the directory of the raw on-board file
OB row skip	int	**Deprecated** Number of header rows in the BetaFlight on-board data .csv file. Only relevant if data is exported to .csv using the blackbox viewer. Necessary for legacy (**V1**) cleanflight.py importing scripts. Current defaults use (**V2**) which utilize `blackboxtools` to systematically decode .BFL and .BBL files.
OB num columns	int	**Deprecated** Number of columns in the BetaFlight on-board data .csv file. Only relevant if data is exported to .csv using the blackbox viewer. Necessary for legacy (**V1**) cleanflight.py importing scripts. Current defaults use (**V2**) which utilize `blackboxtools` to systematically decode .BFL and .BBL files.
Has OJF data	restricted string	Indication of whether OJF wind tunnel data is available (or should be used) for the flight. Choice of 'Y' or 'N'
OJF path	string	(Relative or absolute) path to the directory of the raw OJF wind tunnel wind speed file. Only necessary for wind tunnel flights with wind.
OJF name	string	Filename (without extension) of the OJF wind data. Only necessary for wind tunnel flights with wind.
Static wind (m/s)	float	Wind tunnel wind speed to use if OJF file is unavailable or corrupted. Only necessary for wind tunnel flights with wind but no data file.
Indoor or Outdoor	restricted string	Indication of whether the flight was indoors or outdoors. Choice of 'Outdoor' or 'Indoor'
Flight controller	string	Indication of the flight controller software used during experiment. Dictates which processing scripts should be used. Currently only support for cleanflight/betaflight
Batteries	string	The type of battery used for the experiment. Used in consistency checks for model identification (such that different battery types, e.g. 4S and 6S, are not combined in one model).
Mode	string	Type of flight, LOS (line-of-sight) or FPV (first-person-view).

Mass	float	Mass, in grams, of the quadrotor measured for each flight (may change due to battery swaps; mainly relevant for TinyWhoops).
Drone Config File	string	Filename (without extension) of the corresponding quadrotor configuration file (see section 4.2 for more details)
Drone Config Path	string	(Relative or absolute) path to the directory of the corresponding quadrotor configuration file
Video	string	Filename of any accompanying video
Video Path	string	(Relative or absolute) path to the directory of any accompanying video
Comments	string	Any additional comments/information about the flight

4.2 Quadrotor Configuration File

The quadrotor configuration file is a `.json` that summarizes various properties of the quadrotor. Listing 1 provides an example of such a configuration file, with comments highlighting its contents. Note that `.json` files do not support comments, these are only present in this document for clarity. Additional `key-value` pairs may be added to the quadrotor configuration file so long as they do not interfere with the current entries. When creating your own quadrotor configuration file, ensure that the standard key and value entries have the same structure as in listing 1 with the same data types (mostly strings, with the exception of integers for the `rotor config` entry).

Listing 1: Example of a quadrotor configuration file with comments for clarity preceeded by %

```
{
    % Top-down view of rotor layout with the quadrotor (x-axis) facing forward
    "rotor config": {
        "front left": 4,
        "front right": 2,
        "aft right": 1,
        "aft left": 3
    },
    % Rotation direction of rotor 1; Clockwise (CW) or counter-clockwise (CCW)
    "rotor1 rotation direction": "CW",
    % Radius of a propeller, in meters
    "rotor radius": "0.0381",
    % Diagonal distance, in meters, between rotor hub and quadrotor c.g
    "b": "0.077",
    % Offset between OptiTrack marker centroid and quadrotor c.g. in meters. Translation from OptiTrack to Quadrotor.
    "optitrack marker cg offset": {
        "x": "0",
        "y": "-0.01",
        "z": "-0.048"
```

```
    },
    % eRPM of idling rotor speeds (BetaFlight). eRPM is a scaled
       variant of the true RPM. Scaling depends on motor.
    "idle RPM": "200",
    % eRPM of max rotor speeds (BetaFlight)
    "max RPM": "2100",
    % Moment of inertia of quadrotor in kgm^2 inputted as a matrix:
        [Ixx, Ixy, Ixz; Iyx, Iyy, Iyz; Izx, Izy, Izz]
    "moment of inertia": "[0.000865, 0, 0; 0, 0.00107, 0; 0, 0,
       0.00171]",
    % Deprecated! Scaling factor for acceleration units (legacy
       only)
    "betaflight raw acceleration correction factor": "2048",
    % (Default) Flight controller software, file log takes priority
    "flight controller": "BetaFlight",
    % (Default) Mass in kg, file log mass takes priority
    "mass": "0.433",
    % (Optional) Estimated interial measurement unit noise
       statistics. For each state, both a bias/mean and the
       standard deviation are required.
    "(estimated) imu sensor noise statistics": {
        "ax": {
            "mean": "2.4753479513872104e-18",
            "std": "0.04927457197551707"
        },
        "ay": {
            "mean": "-3.064716511241308e-18",
            "std": "0.0457691686522913"
        },
        "az": {
            "mean": "1.538251941219195e-16",
            "std": "0.04205521067418906"
        },
        "p": {
            "mean": "-2.5784874493616773e-19",
            "std": "0.01665816411596316"
        },
        "q": {
            "mean": "1.399750329653482e-19",
            "std": "0.016930797413400457"
        },
        "r": {
            "mean": "2.596905216857118e-19",
            "std": "0.012040058284801657"
        },
        "roll": {
            "mean": "-3.9782377790151595e-19",
            "std": "0.0014902562844350954"
        },
        "pitch": {
            "mean": "-1.6207635395987686e-18",
            "std": "0.0016066723147324337"
        },
```

```
        "yaw": {
            "mean": "3.1638961892042554e-17",
            "std": "0.0020454533464972064"
        }
    }
}
```

4.3 Pipeline manager - main.py

The `main.py` file in the root directory of the `QuaSI` repository is what connects the different modules of QuaSI together. Users can, in general, leave this file untouched but there are a few booleans which can be toggled to (de-)activate various modules of QuaSI. These are:

```
'''
Script flow booleans
'''
doProcessing = True          # Run processing scripts
doIdentification = True      # Run model identification
makePortableModel = True     # Make a portable version of a model
doMATLAB = True              # Create MATLAB port of model
doSimulation = True          # Run Python simulation
```

Setting any of these booleans to `False` will deactivate the associated module. Note that, when you run QuaSI for the first time, then you need to do so in order (i.e. first processing, identification, then simulation and/or MATLAB port) as shown in fig. 1. Once a given module has completed (e.g. all desired flight data has been filtered), then the associated module may be deactivated to avoid redundant processing when experimenting with model identification. Thus, it is recommended that users set completed modules to `False` when completed.

The primary way users interact with QuaSI is through various configuration files associated with each module. These configuration files are actually used internally by the module scripts, and thus, the `main.py` essentially just passes these configuration files over to the relevant modules, manages data flow, and invokes the appropriate scripts. It is simply a convenient way to run the entire pipeline from one script and directory without worrying about the interfacing between modules.

5 Raw Data Processing

After having gathered raw quadrotor flight data, the first step of the identification process is to estimate (or improve estimates of) the quadrotor states and measurements. In QuaSI this is achieved by fusing the different data sources and filtering the results.

The behavior of this processing module can be configured through the `processConfig.json`. Documentation on the entries of this file are given in section 5.1. Details on the alignment and fusion itself can be found in section 5.2 while information on filtering is given in section 5.3. Finally, some examples regarding the use of this module are presented in section 5.4.

5.1 Configuration files - processConfig.json

The `processConfig.json` file handles the raw data fusion, synchronization, and filtering parameters. Listing 2 summarizes the contents of this file, with comments detailing what the different fields entail. This configuration file is used by the `droneidentification/processQuadData.py` script.

Listing 2: Configurable entries of the processConfig.json file with comments for clarity preceeded by %

```
{
    % Information pertaining to the Experiment File Log (section 4.1)
    "logging file":{
        % Directory of the file log
        "directory":"data",
        % Filename of the file log, without extension
        "filename":"HDBeetle_file_log",
        % Which flight data logs to import from the Experiment File
            Log with header = row 1. See (section 9.1) for details on
            row selection syntax.
        "rows of flights to use (all)":["2-14"]
    },

    % Configuration of data fusion and synchornization parameters
    "data importing":{
        % Boolean to dictate if data should be imported and
           processed. Set to false if data is already imported and
           only filtering (see below) needs to be run.
        "import raw data":true,
        % Desired sampling rate for the fused data, in Hz
        "resampling rate":500,
        % Boolean to dictate whether on-board attitude estimates
           should be used over external attitude estimates
        "use onboard attitude":true,
        % Boolean to determine whether the discrepancy between the
           on-board x-y IMU and (estimated) x-y c.g. locations
           should be corrected for (NED frame). This will modify
           the acceleration measurements to align them with the c.g
           . instead of the IMU. The c.g. is estimated through the
           non-zero pitching and rolling moments during hovering
           flight.
        "do onboard c.g. correction":true,
        % Boolean to determine whether OptiTrack outliers (e.g. due
            to loss-of-tracking) should be filtered out
        "filter optitrack outliers":true,
```

```
24          % Low-pass filter bandwidth to smoothen the OptiTrack
                velocity measurements (see section 5.2)
25          "optitrack velocity Hz cutoff":10,
26          % Boolean of whether to save the processed data or not
27          "save raw data":true,
28          % Directory in which the processed data should be saved, if
                the above is true.
29          "imported data save directory":"data/processed/imported",
30          % List of sates which should be used to synchronize the
                OptiTrack measurements with the on-board measurements
31          "align with optitrack using":["pitch", "yaw"],
32          % Maximum permitted lag between the on-board data and
                OptiTrack data, in seconds. This relates to the time
                difference between arming the quadrotor and starting the
                OptiTrack measurements.
33          "max permitted lag for optitrack":25,
34          % Boolean on whether the results of the synchornization
                should be plotted and shown. Recommended to verify
                whether the synchronization worked as intended.
35          "show onboard and optitrack alignment plots":true
36      },
37
38      % Configuration of data filtering parameters (requires that
            data is first fused and synchornized).
39      "data filtering":{
40          % Whether noise statistics given in the quadrotor
                configuration file (section 4.2) should be used in the
                extended Kalman filter. If true, but no noise statistics
                are found in the quadrotor configuration file, then a [
                WARNING] will be raised and the script will continue
                with default values.
41          "use noise statistics in drone config":true,
42          % Whether filtering should be run
43          "run extended kalman filter":true,
44          % Whether the Kalman filter convergence results (state
                covaiance and state standard deviations over iterations)
                should also be saved in the subsequent files, or only
                the filtered data. If true, an additional file ending in
                "EKFR.pkl" will also be saved alongside the filtered
                data.
45          "save kalman filter convergence results":false,
46          % Boolean indicating whether the gravity vector should be
                removed from the acceleration measurements (and
                associated variables). Not recommended, as for hovering
                flight the gravity vector gives the (hovering) thrust
                produced by the system. Removing gravity will then
                result in models that are relative to the hovering
                condition.
47          "remove influence of gravity":false,
48          % Whether the results of the extended Kalman filter should
                be plotted and shown
49          "show filtering results":true,
50          % Whether the induced velocity computed from the filtered
```

```
            data. Set to true if you would like to use the induced
            velocity in subsequent models. Note that, for descending
             flight, the underlying assumptions are violated, and
            thus the induced velocity is nonsensical.
        "add induced velocity":false,
        % Whether or not the filtered data should be saved
        "save filtered data":true,
        % Directory in which the filtered data is to be saved
        "filtered data save directory":"data/processed/filtered"
    }
}
```

5.2 Data Alignment and Fusion

In many experiments, quadrotor data may stem from various data sources such as the onboard IMU(s), an external motion capturing system like OptiTrack, wind tunnel speed measurements and so forth. Thus, the first step is to organize these data streams and consolidate them into a single data set. This is the goal of the `data importing` step. The `.json` configurable parameters can be found here. By default, the processing pipeline will use the **V2** version of the scripts, which are more consistent and reliable than the **V1** variants (see `droneidentification/processing/importing.py`).

In the majority of applications, the onboard IMU data will be used for model identification. Currently, only BetaFlight and INDIFlight flight logs are supported. This script filters sudden spikes from the attitude (as it is estimated onboard) and transforms the flight data from the BetaFlight axis system to the North-East-Down (NED) reference frame. If users opted to correct for any IMU-c.g. offsets, these will also be conducted.

Moreover, we will often be dealing with the fusion of IMU and external motion data. Currently, QuaSI supports external motion capturing files from OptiTrack Motive 2. The relevant script can be found under `droneidentification/processing/optitrack.py`. While these external motion capturing systems boast high tracking accuracy (and are thus often taken as ground truth), artefacts may still arise in the subsequent data due to poor marker layout and/or reflectivity on the quadrotor, poor OptiTrack calibration, inadequate OptiTrack camera settings (sampling rate, exposure, etc.), a combination of these, or other factors. This is especially true when working with TinyWhoops such as the *Eachine Trashcan* (75mm). Figures 2 and 3 illustrate some of these OptiTrack artefacts (—line) which occur for one such Trashcan (called the DataCan75).

After completing the OptiTrack artefact removal steps, QuaSI resamples the IMU and OptiTrack data to the desired sampling rate and synchronizes their time series. Any excess data is trimmed. An example of this alignment and trimming step is depicted in fig. 4, where the yaw angle is used as the basis for alignment given its distinctive shape.

The output of the importing step is a '.csv' file saved to the `imported/<quadrotor name>` subfolder within the specified save directory. This '.csv' file contains information on the time vector (columns: t), measured rotor speed eRPMs (columns: w1, w2, w3, w4), commanded rotor speed eRPMS (columns: w1_CMD, w2_CMD, w3_CMD, w4_CMD), accelerations (columns: ax, ay, az), rotational rates (columns: p, q, r), attitude (columns: roll, pitch, yaw), position (columns: x, y, z) and body linear velocities (columns: u, v, w).

5.3 Data Filtering

The main goal of the data filtering scripts (see `droneidentification/processing/filtering.py`) is to improve state estimates (e.g. through sensor fusion of onboard and OptiTrack data), mitigate noise, and estimate IMU biases. This is achieved through a combination of an extended Kalman filter (EKF) and various low-pass filters (e.g. for the rotor speeds). The EKF is based on the sensor fusion and bias estimator of [1].

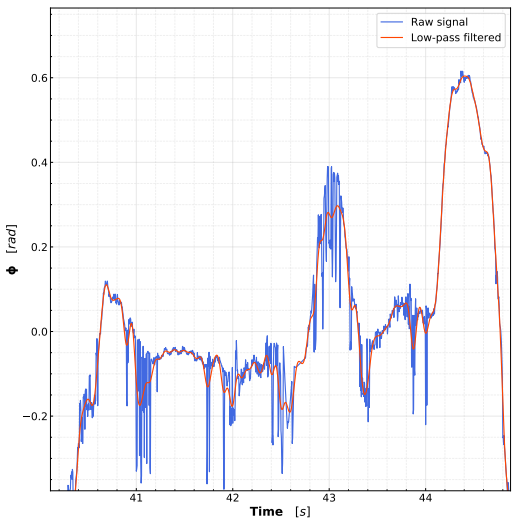


Figure 2: Comparison of the OptiTrack measured roll angle, ϕ, from an arbitrary flight of the DataCan75 before (—) and after (—) the application of the low-pass filter.

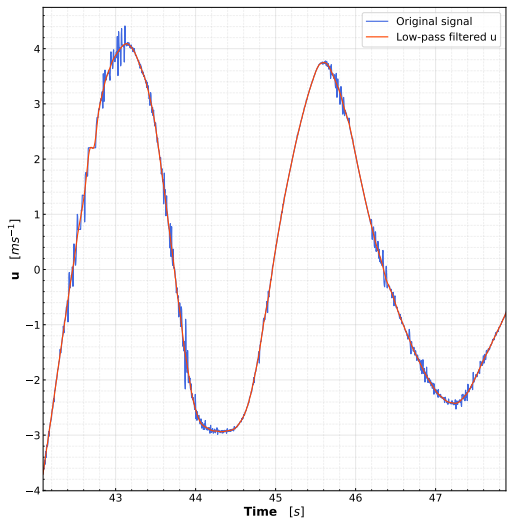


Figure 3: Comparison of the OptiTrack-derived x-axis velocity, u, from an arbitrary flight of the DataCan75 before (—) and after (—) the application of the low-pass filter.

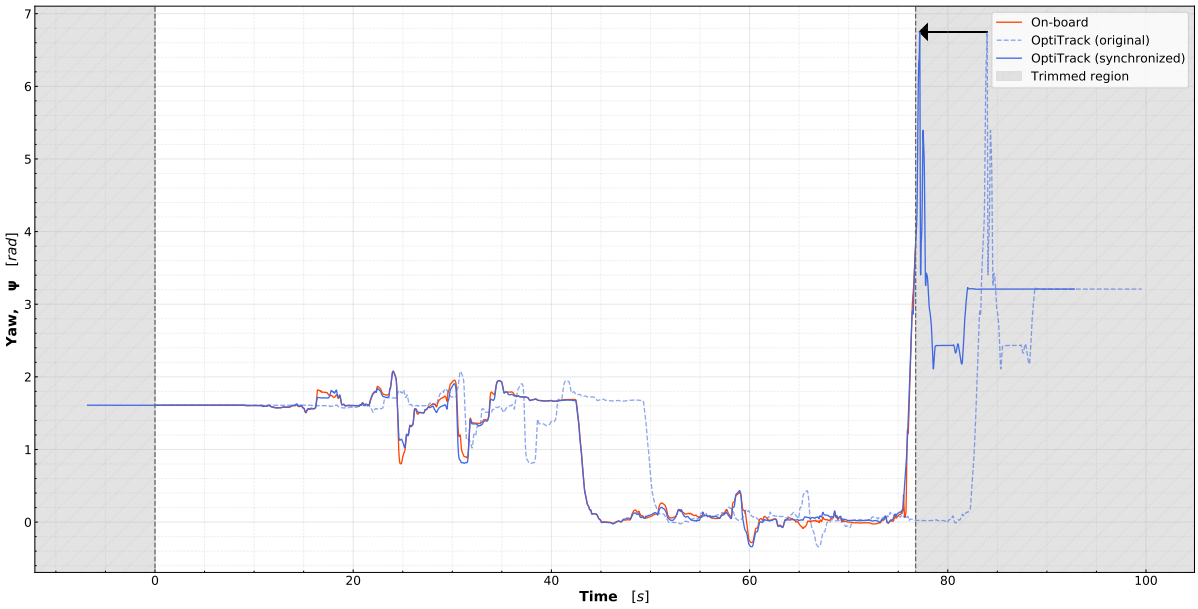


Figure 4: Example alignment of the original OpitTrack data (- -) with the on-board measurements (—) and subsequent trimming of excess data (grey region). The synchronize OptiTrack signal is given in (—). Here, the alignment of the yaw angle, ψ for a flight from the MetalBeetle is shown. The black arrow indicates the direction of delay shift applied to the original OptiTrack signal.

Note that the sensor biases may not be reliably estimated throughout an entire flight as the sensors do not (consistently) measure all the necessary information (e.g. drag). Thus, any unmodelled (in the state equations) or partially measured (e.g. unobservable in OptiTrack) effects will be lumped into the sensor bias estimates, which will therefore vary across a flight. Thus, it is recommended to leave the quadrotor stationary before take-off for a number of seconds in order to facilitate the accurate estimation of sensor biases[11]. The `filtering.py` checks for this stationary period, and if sufficiently long, will apply the bias correction estimated in this period to the rest of the flight log. However, if this period is deemed to short then *no bias correction will*

[11] The downside of this approach is that it cannot account for sensor bias drift or changes throughout the flight.

be made as they cannot be reliably estimated.

Additionally, the filtering steps derive the aerodynamic forces and moments acting on the quadrotor. The resultant `pandas.DataFrame` is saved to a '.csv' file located under the `filtered/<quadrotor name>/withGravity` or `filtered/<quadrotor name>/noGravity` subfolder in the designated save directory, depending on if gravity is to be corrected for. This file contains information on the time vector (columns: t), filtered rotor speed eRPMs (columns: w1, w2, w3, w4), filtered commanded rotor speed eRPMS (columns: w1_CMD, w2_CMD, w3_CMD, w4_CMD), filtered accelerations (columns: ax, ay, az), filtered rotational rates (columns: p, q, r), filtered attitude (columns: roll, pitch, yaw), position[12] (columns: x, y, z), filtered body linear velocities (columns: u, v, w), aerodynamic forces (columns: Fx, Fy, Fz) and aerodynamic moments (columns: Mx, My, Mz).

5.4 Example - Data Processing

Listing 3 gives the processing configuration parameters used in the examples shown in this section. Occasionally, changes to certain parameters are made within examples to demonstrate their effect, such changes will be highlighted where relevant in the `processConfig.json` file.

Listing 3: Processing configuration parameters of the example `processConfig.json`

```
{
    "logging file":{
        "directory":"data",
        "filename":"HDBeetle_file_log",
        "rows of flights to use (all)":["2-5", 8, 13]
    },
    "data importing":{
        "import raw data":true,
        "resampling rate":500,
        "use onboard attitude":true,
        "do onboard c.g. correction":true,
        "filter optitrack outliers":true,
        "optitrack velocity Hz cutoff":10,
        "save raw data":true,
        "imported data save directory":"data/processed/imported",
        "align with optitrack using":["pitch", "yaw"],
        "max permitted lag for optitrack":25,
        "show onboard and optitrack alignment plots":true
    },
    "data filtering":{
        "use noise statistics in drone config":true,
        "run extended kalman filter":true,
        "save kalman filter convergence results":false,
        "remove influence of gravity":false,
        "show filtering results":true,
        "add induced velocity":false,
        "save filtered data":true,
        "filtered data save directory":"data/processed/filtered"
    }
}
```

For those using the `main.py` file, ensure that `doProcessing = True` (set all other script flow flags to `False` to only run processing). Moreover, the relevant `processConfig.json` file is located

[12] The position measurements are left unfiltered by default as they do not enter the state equation nor are they used for model identification.

in the root directory of QuaSI. For those using the droneidentification module directly, the relevant processConfig.json file can be found in that module's root directory and the associated script that handles the processing is processQuadData.py. Ensure that (relative) paths are modified accordingly in the processConfig.json.

5.4.1 Data importing

In the processConfig.json, it is specified to use rows ["2-5", 8, 13]. This will load the flights in rows 2, 3, 4, 5, 8 and 13 from the data/HDBeetle_file_log.csv file log (included in the main git branch). From this file log, it can be seen that rows 2-5 and 8 are system identification flights composed of both first person view (FPV) and line-of-sight (LOS) flights while row 13 is a hovering flight (useful for identifying thrust models). Moreover, all of these flights harbor both on-board and OptiTrack files. Moreover, deactivate the filtering module such that the scripts terminate after the "data importing" module is done:[13]

```
22        "run extended kalman filter": false,
```

When running the processing module[14], the following output (or similar) is broadcast to the terminal window for the first flight (row 2):

> [INFO] Importing rowIdx: 2
>
> [INFO] Importing On-Board data from: data/raw/HDBeetle/Onboard/14-02-2022-CYBERZOO/Demo2-FPV.csv

Such [INFO] messages are prevalent throughout QuaSI to inform the users of what is going on and can act as a verification check that QuaSI is working as intended. Occasionally, users may see [WARNING] prompts. In fact, some may even encounter this warning:

> QuaSI/droneidentification/processing/importing.py:33: **DtypeWarning:** Columns (1) have mixed types.Specify dtype option on import or set low_memory=False. OB_data = import_OB(rowIdx, log, rowBreak = log.loc[rowIdx, 'OB row skip'], numCols = log.loc[rowIdx, 'OB num columns'], doCGCorrection = doCGCorrection)

which actually stems from pandas. It is essentially informing users that there is a mix of data types in the on-board raw data columns. This stems from the BetaFlight .csv exporter which adds text to the time column to mark mode switches (e.g. air mode to horizon mode) or when arming occurs. QuaSI is aware of this and removes these automatically.

After the onboard data is processed, QuaSI loads the OptiTrack data and seeks to synchronize the data streams. Below, QuaSI indicates that it has found the associated OptiTrack file and the rigid body model as specified in the data/HDBeetle_file_log.csv file log:

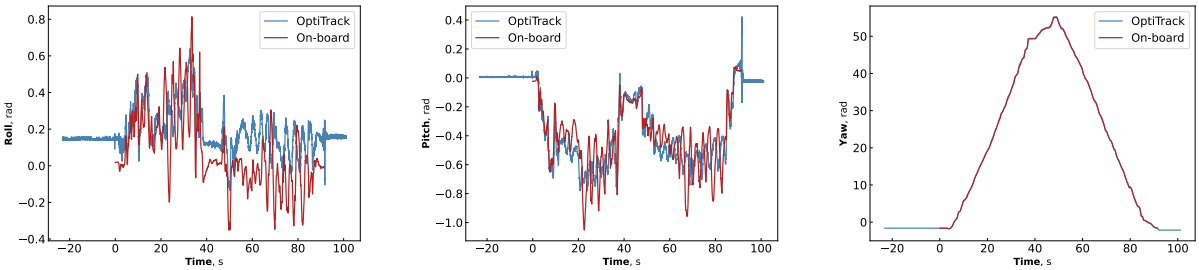
> [INFO] Importing OptiTrack data from: data/raw/HDBeetle/OptiTrack/14-02-2022-CYBERZOO/Demo2-FPV.csv
>
> [INFO] Found MetalBeetle as rigidBodyName

After the flight data is loaded and synchronized, six plots should appear (recall, "show onboard and optitrack alignment plots" is True in our processConfig.json). These depict the alignment results of QuaSI, when using the "pitch" and "yaw" variables[15] for alignment. Figure 5 illustrates the results of this OptiTrack (—) and on-board IMU (—) "pitch" and "yaw" alignment. Again, while showing these plots is optional, it is a good way to verify whether the alignment works as intended.

[13] Of course, when using QuaSI nominally, you may run both the importing and filtering in the same function call. The filtering will start after the importing is completed.

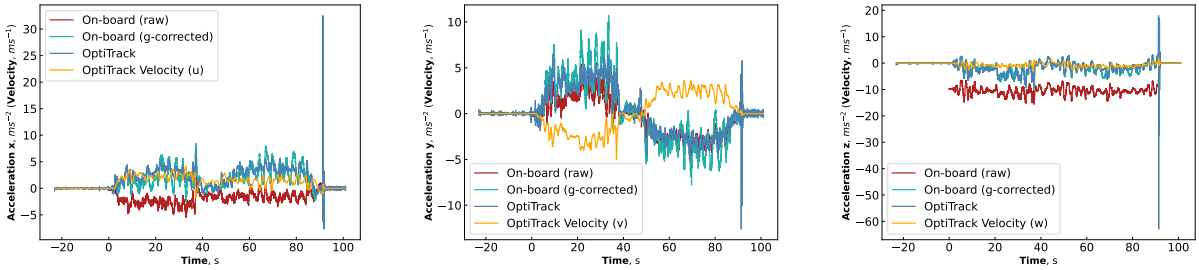
[14] Through either main.py or droneidentification/processQuadData.py

[15] Variables available for alignment are: accelerations ("ax", "ay", "az"), body rotational rates ("p", "q", "r") and attitude ("roll", "pitch", "yaw").



(a) Roll (ϕ), rad (b) Pitch (θ), rad (c) Yaw (ψ), rad

Figure 5: Example result of the on-board and OptiTrack attitude alignment (using "pitch" and "yaw") for flight row 2 in `HDBeetle_file_log`.



(a) Along x-axis (a_x), ms^{-2} (b) Along y-axis (a_y), ms^{-2} (c) Along z-axis (a_z), ms^{-2}

Figure 6: Example result of the on-board and OptiTrack acceleration alignment (using "pitch" and "yaw") for flight row 2 in `HDBeetle_file_log`.

While there are differences in the roll and pitch response, the alignment in fig. 5 is good. Indeed, noisy and inaccurate measurements challenge the alignment capabilities. This is the main reason why the angular measurements are used by default over the rotational rates (even if there may be a slight phase delay introduced). The noise present in the rotational rate alignment often results in a worse phase lag than that introduced by angular alignment. In fact, the "yaw" channel is often good for alignment as it is not as noisy as other channels. However, yaw responses may be absent in many flight logs, thus other variables should be used in tandem. Alternatively, yaw 'alignment markers' can be encoded within the flight manoeuvres (e.g. yaw 90 degrees at the start and end of a flight). This is done for the hovering flight (row 13) precisely to aid in this alignment. Moreover, the alignment markers may also be employed to resolve any discrepancies between a faulty on-board clock (which may run slightly fast or slow) and OptiTrack.

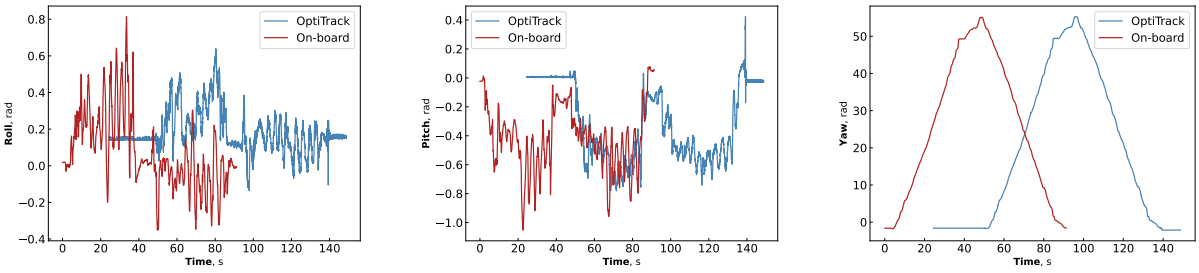
Figure 6 depicts the alignment (using "pitch" and "yaw") of the acceleration data in addition to the OpitTrack derived velocities. As OptiTrack does not measure gravity directly, there initially appears to be a substantial discrepancy between the accelerations. However, after correcting for gravity, the discrepancy disappears and the data is well aligned. Note that this gravity correction is only applied here to verify the alignment, and is not modified in the data itself. Users can instead opt to conduct this correction in the filtering step.

To demonstrate an example of a poor alignment, set the alignment variable to the acceleration along the x-axis ("ax"), that is:

```
16          "align with optitrack using": ["ax"],
```

This results in see fig. 7 wherein a significant delay between the two data streams is visible. If the alignment is bad, users should change the alignment variables[16] (indeed, some flights have more distinct responses in certain axes, while others do not). Another option is manipulate the "max permitted lag for optitrack" parameter.

[16] See footnote 15



(a) Roll (ϕ), rad (b) Pitch (θ), rad (c) Yaw (ψ), rad

Figure 7: Example result of a bad on-board and OptiTrack attitude alignment (using "ax") for flight row 2 in `HDBeetle_file_log`.

Upon the completion of the importing step, the combined data is saved to the directory specified in "imported data save directory". This location is also broadcast by QuaSI:

```
[ INFO ] Saving imported data to data/processed/imported
```

In this directory, there should be files ending in '*-IM.csv' and '*-metadata.json' where "*" is the on-board file name of the imported raw data. The '*-IM.csv' file contains the combined flight data while the '*-metadata.json' file contains information on the relevant configuration parameters. For example, which row and file log the data belongs to, the alignment variables, resampling rate and so on.

The script then continues on to any other flights (i.e. rows) scheduled for importing using the same procedure. Note that, while not done by the default pipeline, it is possible to parallelize this process by flights. The function `droneidentification/importing.runImport()` can be used in this capacity. However, it may become difficult to distinguish between alignment plots in such a design.

5.4.2 Data filtering

In the processConfig.json, the setting of "rows of flights to use (all)" to ["2-5", 8, 13] will load the on-board and OptiTrack combined flights (i.e. output of the data importing step in section 5.4.1) corresponding in rows 2, 3, 4, 5, 8 and 13 in the `data/HDBeetle_file_log.csv` file log. Note that, the importing of these rows needs to be done prior to filtering. If the importing step has already been completed, then the importing module should be deactivated through:

```
22        "import raw data": false,
```

Moreover, ensure that the filtering module is active before running the module:

```
22        "run extended kalman filter": true,
```

When running the filtering module[17], for each flight (i.e. row in the file log) scheduled for filtering, the following status information (or similar) will be shown:

```
[ INFO ] Loading rawData from data/processed/imported
[ INFO ] Running EKF (Demo2-FPV)...
[ INFO ] Found drone specific noise statistics in config file, using these over default.
```

[17] Through either `main.py` or `droneIdentification/processQuadData.py`. Ensure relative paths in processConfig.json are from the appropriate root directory of the script used.

Here, users may confirm that the correct flight data file is loaded from the correct directory. If "use noise statistics in drone config" is `true` and quadrotor-specific noise statistics are available in the quadrotor configuration file, then these will be used (as is broadcast by QuaSI scripts above). If "use noise statistics in drone config" is `true` but no quadrotor-specific noise statistics are available, the following message will be displayed instead:

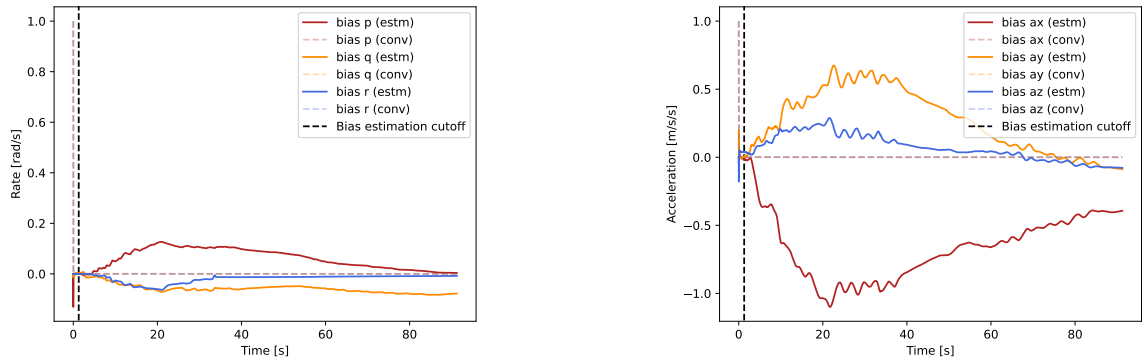
> [WARNING] Could not find drone-specific noise statistics. Using default values. Kalman filtering results will likely not estimate noise well.

Alternatively, if "use noise statistics in drone config" is `false` then no message will be displayed regarding quadrotor noise. The default behaviour of the filtering scripts when no quadrotor-specific noise is given is to assume that the OptiTrack results are highly reliable whereas the IMU results are not reliable. Hence, the estimated states will be weighted towards the OptiTrack measurements.

The progress of the EKF is summarized through a progress bar similar to:

> 73%|==================> | 33304/45622 [00:16<00:07, 1919.47it/s]

Upon completion, five plots should appear summarizing the EKF results (if "show filtering results" is `true`). The most important plots to pay attention to are those pertaining to the bias estimation. An example of these plots is depicted in fig. 8, wherein the estimated biases are given by the solid lines (—) and the associated state covariance represented by the corresponding opaque dashed lines (- -).



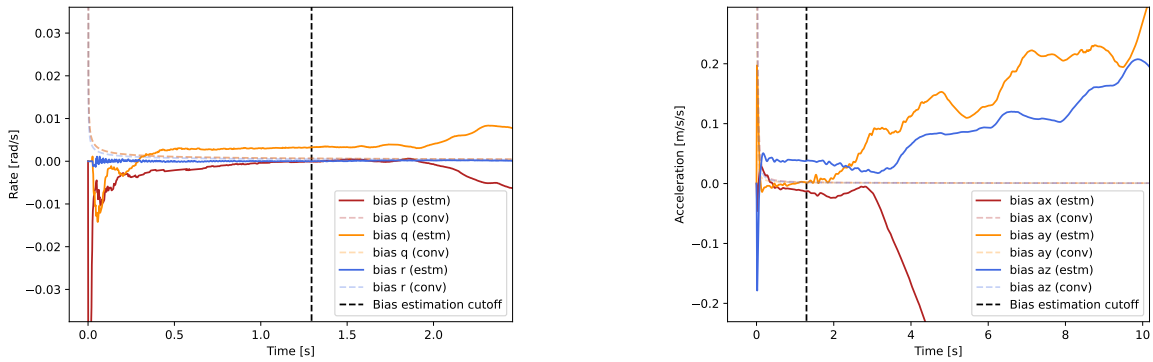
(a) Rotational rate

(b) Acceleration

Figure 8: Example result of the extend Kalman filter estimated IMU sensor biases for flight row 2 in `HDBeetle_file_log`.

Notice how the bias can vary considerably throughout the rest of the flight (i.e. when manoeuvres are conducted) as un-modelled dynamics or sensor measurement discrepancies are lumped into the bias term estimation. Thus, the bias can only be reliably estimated within the region before the dashed vertical black line. This corresponds to the so-called stationary period for which more details can be found in section 5.3. The key takeaway in this example is to verify whether the biases have satisfactorily converged or not within this region. A zoomed in version of these bias estimation plots is given by Figure 9, for which the biases appear to have converged by the vertical black dashed line.

This convergence is more dubious for rate bias estimation results of `Demo4-LOS` (row 4 in `HDBeetle_file_log`), shown in fig. 10. While the roll rate converges rapidly, it still harbors a relatively large value by the end of the stationary period (especially in comparison to that of



(a) Rotational rate

(b) Acceleration

Figure 9: Example result of the extend Kalman filter estimated IMU sensor biases for flight row 2 in `HDBeetle_file_log`, zoomed into the stationary period region.

`Demo2-FPV`). Moreover, the initial estimate spikes up to around 300 rad/s which is indicative of an artefact in data[18].

Included in the output plots are the pre- and post- filtered attitude and velocity. Figure 11 illustrates an example of the velocity corrections wherein the solid lines (—) indicated the filtered state and the original (unfiltered) state is given by the corresponding opaque dashed lines (- -). Indeed, there are some slight corrections made to the unfiltered measurements by the Kalman filter. In particular, the jitter-like responses have largely been removed.

[18]While poor initialization is often the culprit in such extended Kalman filter designs, the initial states are given by the measurements directly whereas the biases are estimated to be zero initially. Jitter in the OptiTrack or IMU data may, at first, lead to a large bias estimate for one of the states which quickly converges to more appropriate values.

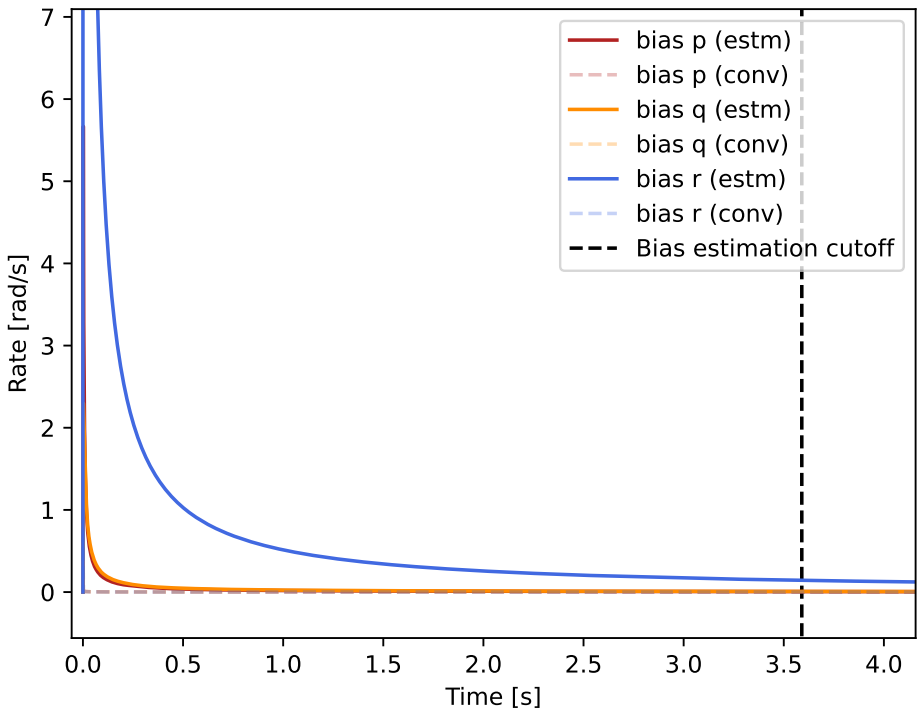


Figure 10: Example result of the extend Kalman filter estimated IMU gyro biases for flight row 4 in `HDBeetle_file_log`, zoomed into the stationary period region.

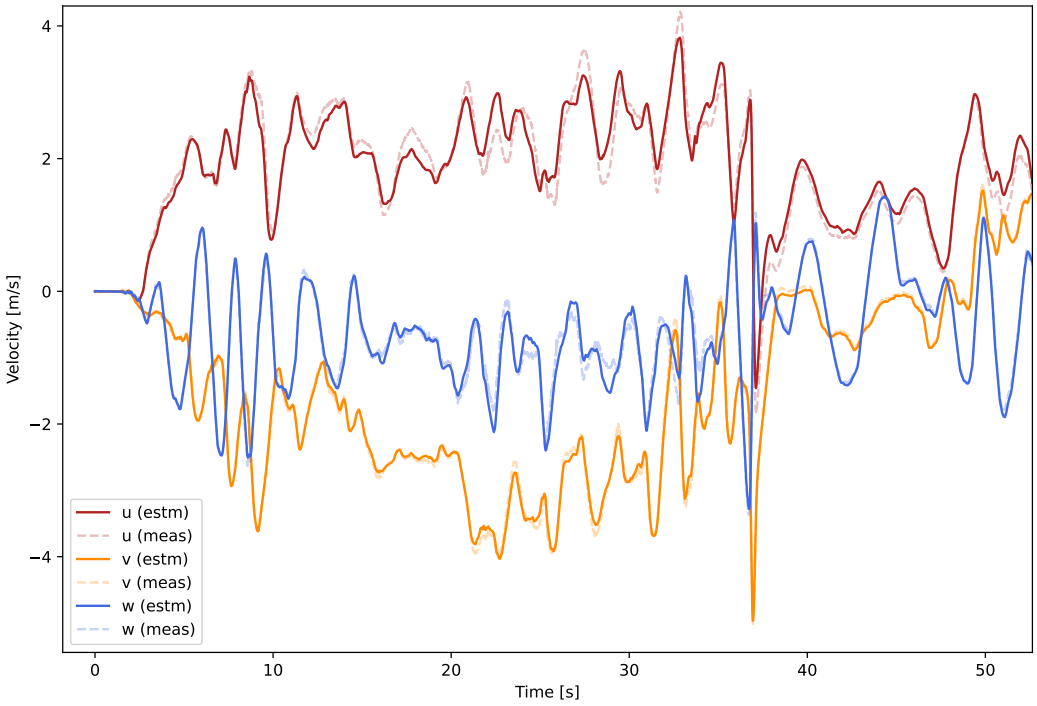


Figure 11: Comparison between the unfiltered and extend Kalman filter estimated velocity for flight row 2 in `HDBeetle_file_log`

Subsequently, the filtering scripts derive the aerodynamic forces and moments and save the data to the designated directory

```
[ INFO ] Calculating aerodynamic moments
[ INFO ] Saving filtered data to data/processed/filtered/withGravity
```

In the save directory, users will find files ending in "*-FL.csv" and "*-metadata.json". These correspond to the filtered data and corresponding metadata respectively. The metadata details the raw flight data source files, along with information on the filtering parameter configuration, used to construct the filtered data set.

Currently, QuaSI applies the extended Kalman filter to the flight data sequentially. This process can be parallelized across flights. A parallelized version of the filtering scripts can be constructed through calls to the function `droneidentification/filtering.runFilter()`. Note that, this script by itself does not support parallelization.

6 Quadrotor Model Identification

Following the data processing (section 5), polynomial aerodynamic models of the quadrotor are identified through step-wise regression. Here, the term 'aerodynamic models' refer to the forces and moments acting along each of the quadrotor's axes. The main nodes of the identification module are parameterized through various configuration `.json` files. Documentation on these files can be found in section 6.1. Background information on step-wise regression and polynomial candidate generation are given in section 6.2 and section 6.4 respectively. Details on the (optional) normalization scheme are summarized in section 6.3. A brief description of the motivation and purpose of the portable models is given in section 6.6

Examples on the usage of the model identification scripts and portable model creation are given in section 6.5 and section 6.6.1 respectively.

6.1 Configuration files

This section provides documentation on the configuration files used by the model identification scripts. The `identificationConfig.json` (section 6.1.2) file controls the actual polynomial model identification whereas the `standaloneConfig.json` (section 6.1.2) manages the parameters need to create portable versions of identified models.

6.1.1 identificationConfig.json

As the name suggests, the `identificationConfig.json` file hosts the configurable quadrotor model identification parameters. Listing 4 gives a commented overview of the entry fields in this configuration file.

Listing 4: Configurable entries of the identificationConfig.json file with comments for clarity preceeded by %

```
{
    % Information pertaining to the Experiment File Log (section 4.1)
    "logging file": {
        % Same as line 4 in listing 2
        "directory": "data",
        % Same as line 6 in listing 2
        "filename": "HDBeetle_file_log",
        % Which flight data logs to use. Rows correspcond to
           entries in the Experiment File Log (section 4.1) with
           header = row 1. For row selection syntax see
           (section 9.1).
        "rows of flights to use (all)": ["2-14"],
        % Subset of the loaded flight logs above to use for model
           validation (i.e. unseen during training).
        "rows of flights for validation": [3,6]
    },

    % Directory of the filtered data
    "filtered data save directory": "data/processed/filtered/
       withGravity",

    % Training filtered data overview
    "plotting": {
        % Boolean to show plots of the xyz trajectories of the
           training flight logs
        "show trajectories": false,
```

```
21          % Number of quadrotors renders to show in the xyz
                trajectories above. Quadrotor frame opacity corresponds
                to time where the less transparent, the more recent.
22          "number of drone stills": 5,
23          % Boolean to show animation of the xyz trajectories
24          "show animation": false,
25          % Factor by which to increase the animation speed. Should
                be greater than 1 (i.e. slow downs not permitted). In
                optimal conditions, the animation should be
                approximately real-time. However, depending on the
                system it may be faster or slower.
26          "animation speed up factor": 1
27      },
28
29      % Data normalization options
30      "data normalization": {
31          % Whether data should be normalized. See section 6.3 for
                details on the normalization scheme
32          "normalize data": false
33      },
34
35      % Data partitioning options
36      "data partitioning": {
37          % Ratio of an individual training flight log available for
                model identification. The ratio is centered around the
                mid-point of the flight. That is, a usable data ratio =
                0.9 will remove the first and last 5% of the flight log.
                 Typically, we would like to avoid take-off and landing
                phases as there may be external forces acting on the
                system. This trimming is done before the subsequent
                partitioning.
38          "usable data ratio": 0.9,
39          % Whether the training data should be further partitioned
                into training and test subsets (with ratio on line 42).
                Currently, only a pseudo-random partition is supported.
                If true, then the test flight subset will be used to
                evaluate the performance of the models. If false, and
                there exists validation flights (see line 11), then the
                validation flights will be used to evaluate model
                performance. In both cases, the data set used to
                evaluate performance is not used during training for
                that run. To check for over-fitting, set to true and add
                 validation flights to line 11. The identification script
                 will warn users if the difference in test and
                validation  RMSE is too large and over-fitting may be
                occuring.
40          "random partition": true,
41          % The ratio of training data to test data, applied if "
                random parition" is true.
42          "partition ratio":0.75
43      },
44
45      % Manoeuvre partitioning options
```

```
    "manoeuvre excitations": {
        % Dictates whether the training data should first be
           isolated to regions of excitations (i.e. where there are
            relevant force/moment excitations). This is
           particularly useful for moment models wherein
           excitations are temporarly short and sparse. Excitation
           isolation is redundant, and perhaps detrimental, if the
           base flights logs are already information rich.
        "isolate to regions of excitation": false,
        % Whether the results of the excitation isolation algorithm
            should be plotted for viewing
        "show isolation results":true,
        % A hyper-parameter to control the excitation isolation
           algorithm behaviour, such as minimum magnitude and
           prominence of excitations. A higher value is generally
           more strict and requires more prominent and obvious
           excitations whereas a lower value is less stringent but
           may consider the full flight log as sufficiently rich in
            excitations.
        "excitation threshold": 0.5,
        % Controls the region surrounding a manoeuvre which will be
            permitted for identification. Essentially, the run-in
           and run-off regions of a manoeuvre. Given as a ratio of
           the total flight log length. The resultant region is [-
           spread, excitation, +spread].
        "spread": 0.02
    },

    % Model identification options
    "identification parameters": {
        % Whether to show box plots of the spread in estimated
           rotor speed rate constants between flight logs.
        "show actuator constant estimation": false,
        % Path to the fixed and candidate regressor basis file.
           Used to generate candidate pool of regressors and
           defines fixed regressors for each aerodynamic model.
           Refer to (section 6.4) for more details.
        "polynomial specification file":"droneidentification/
           polynomialCandidates/
           polyCands_DiffSys_quadraticU_symmetric_SIMPLE_FM.json",
        % Booleans on whether a bias (i.e. constant state-
           independent) term should be added to the polynomial
           aerodynamic models.
        "add bias term fx":true,
        "add bias term fy":true,
        "add bias term fz":true,
        "add bias term mx":true,
        "add bias term my":true,
        "add bias term mz":true,
        % Boolean to control which models should be identified. In
           principle, all models can be identified in one go (i.e.
           all true). However, depending on the size of the
           training data, we may encounter memory issues which
```

```
            necessitates sequential identification.
        "identify fx": true,
        "identify fy": true,
        "identify fz": true,
        "identify mx": true,
        "identify my": true,
        "identify mz": true,
        % Confidence level of the accompanying prediction intervals
            . These prediction intervals give insight into the model
             reliability and certainty; they bound the uncertainty
            of a prediction.
        "prediction interval confidence level": 0.95,

        % sysidpipeline hyper-parameters
        "polynomial": {
            % Upper limit on the allowable number of steps of the
                stepwise regression algorithm. It will force a
                termination of the algorithm if it does not
                terminate naturally after the "regressor cap" number
                 of forward-backward cycles.
            "regressor cap": 3
        }
    },

    % Model saving options
    "saving models": {
        % Determines whether models should be saved or not
        "save identified models": true,
        % Directory in which to save the identified models
        "save directory": "models/HDBeetle",
        % Name of the aerodynamic model
        "model ID": "MDL-HDBeetle-NN-II-TEST"
    }
}
```

6.1.2 standaloneConfig.json

Also part of the `droneIdentification` library, the `standaloneConfig.json` file is used to configure the generation of portable polynomial model objects. Listing 5 summarizes the entries of this configuration file, supplemented with comments for clarity.

Listing 5: Configurable entries of the standaloneConfig.json file with comments for clarity preceeded by %

```
{
    % Directory in which the model folder is located
    "model path": "models/HDBeetle",
    % Name of the model (folder)
    "model ID": "MDL-HDBeetle-NN-II-TEST",
    % Directory of the quadrotor configuration file (section 4.2)
    "droneConfig path": "data",
    % Name of the quadrotor configuration file
    "droneConfig name": "HDBeetleConfig.json",
```

```
10      % Whether a (moment model) differential form of the polynomial
            model should be created and added to the standalone model.
            Currently, only differential forms of the moment model are
            supported.
11      "make moment DiffSys": true,
12      % The differential form input vector
13      "DiffSys u": ["U_p","U_q","U_r"]
14  }
```

6.2 Polynomial stepwise regression

In general, a polynomial model may be expressed in the form of eq. (1) where $z \in \mathbf{R}^N$ denotes the target measurement data of length N. The matrix A represents arbitrary combinations (e.g. power series) of the independent variables and Θ denotes the parameters (i.e. coefficients) of the polynomial model. Residuals between the model predictions and the measurements are encompassed by ϵ.

$$z = A\Theta + \epsilon \tag{1}$$

For systems which are linear in the parameters, the optimal model parameters, $\hat{\Theta}$, can be estimated through ordinary least squares (eq. (2)).

$$\hat{\Theta} = (A^T A)^{-1} A^T z \tag{2}$$

Stepwise regression seeks to determine the model structure of A that best fits the target data, given a candidate pool of regressors. The principle underlying the approach is to, in the forward step, select a regressor from the candidate pool which best improves on model accuracy. Subsequently, in the backward step, the selected model regressors are reassessed by evaluating the model performance with each of the regressors removed. The goal is to identify any redundant regressors which may have become superfluous due to (combinations) of other selected regressors. The algorithm terminates once the regressor that was removed in the backward step is the same as that added in the forward step, otherwise the forward step commences again and the process repeats itself.

The stepwise regression is traditionally initialized with eq. (1) and $\mathbf{A} = [1, \ldots, 1]^T$. This is done by default in QuaSI, unless "add bias term _" is `false` in line 58. Moreover, any fixed regressors are also included in the initial $\mathbf{A}$ matrix. Candidate regressors are then added to the model based on their correlation with the measured targets, $\mathbf{z}$, after orthogonalizing them with respect to the current model regressors. This is done to search for regressors, correlated with the targets, which contain information that is currently absent in the current model [2].

1. The candidate regressors in the pool are orthogonalized with respect to the selected regressors in $\mathbf{A}$. This is accomplished through eq. (3). Here, ξ_j represents the current regressor.

2. The measurement data, $\mathbf{z}$, are likewise modified using eq. (4) to account for the adjustments of the candidate regressors in the previous step. Note that $\mathbf{A}$ is the same in both eq. (3) and eq. (4)

$$\boldsymbol{\epsilon}_{\xi,j} = \boldsymbol{\xi}_j - \mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\boldsymbol{\xi}_j \tag{3}$$

$$\boldsymbol{\epsilon}_{z,j} = \mathbf{z}_j - \mathbf{A}(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{z}_j \tag{4}$$

Subsequently, the correlation of the (adjusted) candidate regressors with the (modified) measurement points is evaluated for each regressor through eq. (5) [2].

$$r_j = \frac{\sum_{i=1}^{N} \left(\epsilon_{\xi,j}[i] - \bar{\epsilon}_{\xi,j}\right)\left(\epsilon_{z,j}[i] - \bar{\epsilon}_{z,j}\right)}{\sqrt{\sum_{i=1}^{N} \left(\epsilon_{\xi,j}[i] - \bar{\epsilon}_{\xi,j}\right)^2 \sum_{i=1}^{N} \left(\epsilon_{z,j}[i] - \bar{\epsilon}_{z,j}\right)^2}} \tag{5}$$

The (unmodified) regressor corresponding to the highest absolute correlation (i.e. that best explains the currently unmodelled variance in the measurements) is selected for addition to the model. This regressor is only added if its contribution is significant enough, assessed through a F-statistic given by eq. (6). Here, F_0 denotes the test statistic and F_{IN} is the user-defined cutoff based on a significance level, α, of the test statistic (i.e. $F_{IN} = F(\alpha; 1, N-p-1)$). For the quadrotor, it is often the case that the number of data points is large ($N \gg 100$) and current number of model regressors small ($p < 10$), thus, $N \gg p$. Thus, F_{IN} is assumed to be constant and is commonly set to $F_{IN} = 4$ [2].

$$F_0 = \frac{SS_R(\hat{\boldsymbol{\Theta}}_{p+j}) - SS_R(\hat{\boldsymbol{\Theta}}_p)}{s^2} > F_{IN} \tag{6}$$

In eq. (6), $\hat{\boldsymbol{\Theta}}_p$ gives the model parameters without the added regressor, $\boldsymbol{\xi}_j$, while $\hat{\boldsymbol{\Theta}}_{p+j}$ represents the model parameters with the addition of $\boldsymbol{\xi}_j$. In both cases, the parameters can be estimated through OLS using eq. (2). $SS_R(\hat{\boldsymbol{\Theta}})$ denotes the regression sum of squares and is defined in eq. (7) where $\bar{z}$ the mean measurement value [2].

$$SS_R(\hat{\boldsymbol{\Theta}}) = \hat{\boldsymbol{\Theta}}^T \mathbf{A}^T \mathbf{z} - N\bar{z} \tag{7}$$

s^2 denotes the fit error variance and is defined by eq. (8) with p denoting the current number of regressors in the model and $\hat{y}_i$ the prediction of point i (refer to eq. (9)) [2]. If the condition $F_0 > F_{IN}$ is satisfied, then the regressor $\boldsymbol{\xi}_j$ is added to the model.

$$s^2 \triangleq \hat{\sigma}^2 = \frac{\sum_{i=1}^{N} (z_i - \hat{y}_i)^2}{N-p-1} \tag{8}$$

$$\hat{\boldsymbol{y}} = \mathbf{A}\hat{\boldsymbol{\Theta}} \tag{9}$$

Following the addition of a regressor $\boldsymbol{\xi}_j$, all model regressors are examined for redundancy [2]. It is important to note that, during this check, the regressors are not adjusted and therefore remain in their original form. For p model regressors, the quality of each regressor, $\boldsymbol{\xi}_k$, is evaluated through the test statistic defined in eq. (10) where $\hat{\boldsymbol{\Theta}}_{p-k}$ denotes the parameters corresponding to a model with all p regressors except $\boldsymbol{\xi}_k$ [2].

$$F_{0,k} = \frac{SS_R(\hat{\boldsymbol{\Theta}}_p) - SS_R(\hat{\boldsymbol{\Theta}}_{p-k})}{s^2} \tag{10}$$

Should any of the regressors satisfy the condition $F_{0,k} < F_{OUT}$ where F_{OUT} is another user-specified threshold, then regressor with the lowest $F_{0,k}$ is removed from the model. As with F_{IN}, typically $F_{OUT} = 4$ [2]. Should the removed regressor be the same as the just added regressors (i.e. $\boldsymbol{\xi}_k = \boldsymbol{\xi}_j$) then the algorithm terminates, otherwise it continues with the forward and backward steps.

In QuaSI, the termination condition is supplemented with the Predict Square Error (PSE), given by eq. (11), to minimize over-fitting by penalizing the addition of regressors [2]. In eq. (11), σ^2_{max} is the a constant defined by eq. (12). The second term in eq. (11) represents the penalty for a model with p regressor terms.

$$PSE \triangleq \frac{1}{N}(\mathbf{z} - \hat{\mathbf{y}})^T(\mathbf{z} - \hat{\mathbf{y}}) + \sigma^2_{max}\frac{p}{N} \tag{11}$$

$$\sigma^2_{max} = \frac{1}{N}\sum_{i=1}^{N}(z_i - \bar{z})^2 \tag{12}$$

Additionally, an upper limit may be imposed on the number of forward-backward steps taken in the algorithm (i.e. "regressor cap" in `idenfiticationConfig.json`).

Extensive documentation on the stepwise regression implementation and available functions can be found in the `README.md` file of the `sysidpipeline` repository.[19] and jupyter notebook files in the `sysidpipeline` repository.

6.3 Normalization (Not recommended)

As is common in the aerospace industry, users can elect to non-dimensionalize the quadrotor variables prior to model identification. Unfortunately, there is no standard normalization procedure for quadrotors and typical methods are often ill-equiped to handle the state variations of the quadrotor.

Consider, for example, the typical non-dimensionalization through the air speed for conventional aircraft. Such a non-dimensionalization would lead to singularities when the quadrotor is stationary, such as in hovering flight. Thus, this route is impractical. Alternatively, normalization by the propeller tip velocities (i.e. $V = \omega^2 R$)[20] may be used, as is done for helicopters. However, quadrotors vary thrust by changing the rotor RPMs directly whereas helicopters manipulate the angle of attack while maintaining a relatively constant rotor speed. Due to this, the non-dimensionalization through the rotor speeds leads to numerical issues stemming from an exploding range of 'non-dimensional' values due to the quadratic relation between rotor speed and linear tip velocity. Moreover, we still encounter a singularity as we approach low RPM values, causing the normalization to explode for, typically, small forces. These aspects often lead to numerical issues when considering the majority of the quadrotor's flight envelope.

Nonetheless, a modified variant of the helicopter normalization scheme is available in the identification scripts. These adjustments seek to address some of the numerical issues which plague the quadrotor normalization.

The main normalizing entity is the averaged rotor speed, ω_{avg}, given by

$$\omega_{avg} = \sqrt{\frac{1}{4}\sum_{i=1}^{4} \omega_i^2} \tag{13}$$

where ω_i is the rotor speed of an individual rotor. These rotor speeds are then normalized through ω_{avg} as $\bar{\omega}_i = \omega_i / \omega_{avg}$. These normalized rotor speeds are then used to derive the normalized control moments (i.e. differential thrust required to enact rolling, pitching, and yawing moments).

The rotational rates, $[p, q, r]$, may be normalized through

$$\bar{p} = \frac{pb}{\omega_{avg} R}, \quad \bar{q} = \frac{qb}{\omega_{avg} R}, \quad \bar{r} = \frac{rb}{\omega_{avg} R} \tag{14}$$

where R denotes the propeller radius and b is the characteristic length of the quadrotor (taken here as the diagonal distance between the rotor hub and the c.g.), both in meters. Likewise the velocities, $[u, v, w]$ may be normalized through

$$\mu_x = \frac{u}{\omega_{avg} R}, \quad \mu_y = \frac{v}{\omega_{avg} R}, \quad \mu_z = \frac{w}{\omega_{avg} R} \tag{15}$$

Note that the above equation actually describes the advance ratio of the propeller - the body velocity contextualized in terms of the rotor tip speed. Alternatively, the overall velocity magnitude can also be used as a normalization entity.

$$\bar{u} = \frac{u}{\sqrt{u^2+v^2+w^2}}, \quad \bar{v} = \frac{v}{\sqrt{u^2+v^2+w^2}}, \quad \bar{w} = \frac{w}{\sqrt{u^2+v^2+w^2}} \tag{16}$$

[19] Note that there is a general `README.md` file in the root directory and a `README-stepwise_regression.md` file in the `techniques` subfolder.

[20] Here, V is the linear velocity of the propeller tip, with radius R, and rotational rate ω

In QuaSI, this distinction between normalization schemes is made through the variable names. The advance ratios are given by column names `"mu_x"`, `"mu_y"`, and `"mu_z"` where as the (normalized) velocities are denoted by `"u"`, `"v"`, and `"w"`.

The aerodynamic body forces and moments along each axis, $[F_x, F_y, F_z, M_x, M_y, M_z]$, are normalized through

$$C_x = \frac{F_x}{N}, \quad C_y = \frac{F_y}{N}, \quad C_z = \frac{F_z}{N}, \quad C_l = \frac{M_x}{Nb}, \quad C_m = \frac{M_y}{Nb}, \quad C_n = \frac{M_z}{Nb} \tag{17}$$

where

$$N = \frac{1}{2}\rho(D\pi R^2)(R\frac{w_{max}^2}{(w_{max} + w_{avg})})^2 \tag{18}$$

Here, ρ is the air density, D describes the number of rotors (for the quadrotor $D = 4$), and w_{max} is the maximum (e)RPM of the quadrotor.

The inclusion of the maximum (e)RPM ratio term alleviates singularities when $w_{avg} \approx 0$ and constrains the normalization factor, N, to the interval $[\frac{1}{4}N_{cons}, N_{cons}]$ for $w_{avg} = w_{max}$ and $w_{avg} = 0$ respectively, with $N_{cons} = \frac{1}{2}\rho(D\pi R^2)(R\omega_{avg})^2$. However, the normalization is still dependent on the maximum (e)RPM of the quadrotor and numerical issues may nonetheless arise due to large values for ω_{max}^2.

6.4 Polynomial Candidates

In general, the candidate polynomial regressor pools can be quite extensive (in the order of thousands of regressors). QuaSI offers a systematic way to easily define these candidate regressor pools, along with fixed regressors, for each of the force and moment aerodynamic models. We here use a compact notation to describe candidate polynomials. An illustrative example of the relationship between this compact notation (in red) and the resultant candidate regressor pool (in teal) is given by eq. (19). Here, 1 denotes the (fixed) bias vector, b an arbitrary fixed regressor, $P^2\,(x, z)$ represents the polynomial basis, and $\{1, a\}$ the interacting factors.

$$\begin{aligned} \hat{Y} &= 1 + b + P^2\,(x, z)\,\{1, a\} \\ &= 1 + b + P^2\,(x, z) \cdot 1 + P^2\,(x, z) \cdot a \\ &= 1 + b + x^2 + 2xz + z^2 + ax^2 + 2axz + az^2 \end{aligned} \tag{19}$$

Notice that the polynomial basis does not include the bias vector (as this is already accounted for by 1 in the fixed regressors). The goal of the stepwise regression algorithm is to find appropriate coefficients for selected regressors through linear regression. That is, to find the set of coefficients C_i with $i = \{1, ..., 7\}$ which best approximate the target variable Y. If all the regressors in the candidate pool of eq. (19) are to be selected, this would yield:

$$\hat{Y} = C_0 + C_1 b + C_2 x^2 + C_3 xz + C_4 z^2 + C_5 ax^2 + C_6 axz + C_7 az^2 \tag{20}$$

In QuaSI, the fixed and candidate regressors are constructed from a polynomial specification `.json` file, for which a specific syntax applies. For example, eq. (19) may be specified through listing 6 or, equivalently, listing 7. The purpose of the second alternative syntax example (i.e. listing 7) is to demonstrate how different polynomials (which perhaps depend on different basis variables!) can be specified as a list of candidate constructors. Additionally, any overlap in regressors are removed from the candidate pool by QuaSI, such that each regressor in the total candidate pool is unique.

Note that, for both listing 6 and listing 7, the bias vector (i.e. 1 in eq. (19)) is absent from the fixed regressors. This is because it is added by default in QuaSI. Users may instead opt to remove the bias vector in identificationConfig.json. Note, however, that if the bias vector is removed then there needs to be at least one fixed regressor present to properly initialize the step-wise regression algorithm.

Moreover, each of the variables specified as a string (i.e. "x", "z", "a", and "b") should correspond to columns in the input `pandas.DataFrame`. Users may consult section 6.4.1 for an overview of the default variables made available by QuaSI for model identification. These variables (or indeed any others added to the `pandas.DataFrame`) may be combined through simple equations (i.e. multiplication, summation, subtraction, division, and powers). See section 9.2 for more information on what types of equations are permitted.

Listing 6: Syntax to construct fixed polynomial regressors and candidate regressor pools using the compact notation with comments for clarity preceeded by %

```
{
    % Name of model
    "Y":{
        % Candidate regressors in compact notation:
        %   P^(degree)(vars) * {sets}
        "candidates":[
            {
                "vars":["x", "z"],
                "degree":2,
                "sets":[1, "a"]
            }
        ],
        % Fixed regressors
        "fixed":["b"]
    },
}
```

Listing 7: Alternative syntax to construct fixed polynomial regressors and candidate regressor pools using the compact notation with comments for clarity preceeded by %.

```
{
    % Name of model
    "Y":{
        % Candidate regressors in compact notation:
        %   P^(degree)(vars) * {sets}
        "candidates":[
            {
                "vars":["x", "z"],
                "degree":2,
                "sets":[1]
            },
            {
                "vars":["x", "z"],
                "degree":2,
                "sets":["a"]
            }
        ],
        % Fixed regressors
        "fixed":["b"]
    },
}
```

In the polynomial specification file for quadrotor model identificaiton, each force (keys: "Fx", "Fy", "Fz") and moment (keys: "Mx", "My", "Mz") harbor their own construction recipe for the candidate regressors (under `candidates`) and fixed regressors (under `fixed`). Using the compact notation, polynomial candidates are constructed by specifying the basis variables under `vars`,

the degree of the polynomial under `degree`, and the interacting factors through `sets`. Users can list an arbitrary number of these polynomial compact forms to define the global candidate regressor pool. The entries in `vars` should correspond to columns in the data set `pandas.DataFrame`, or basic equations thereof (see section 9.2). Likewise, the valid entries for the `sets` are numbers, columns in the data set `pandas.DataFrame`, and basic equations thereof. Standard polynomial candidate files can be found in the `droneidentification/polynomialCandidates` subfolder. A commented example is shown in the appendix by listing 17.

6.4.1 (Default) variables available for quadrotor polynomial construction

Table 2: Default variables available for constructing polynomial candidates.

Variable name(s)	Description
`"w1", "w2", "w3", "w4"`	(True) Rotor speeds of rotors 1, 2, 3, and 4
`"d_w1", "d_w2", "d_w3", "d_w4"`	Difference between hovering thrust and the (true) rotor speeds of rotors 1, 2, 3, and 4
`"w2_1", "w2_2", "w2_3", "w2_4"`	Squared (true) rotor speeds of rotors 1, 2, 3, and 4
`"wtot"`	Sum of (true) rotor speeds
`"d_wtot"`	Difference between hovering thrust and the sum of (true) rotor speeds
`"wavg"`	Average of (true) rotor speeds (using eq. (13))
`"w1_CMD", "w2_CMD", "w3_CMD", "w4_CMD"`	(Commanded) Rotor speeds of rotors 1, 2, 3, and 4
`"u_p", "u_q", "u_r"`	Differential thrust to produce rolling (p), pitching (q), and yawing (r) moments. Derived using "w1", "w2", "w3", "w4".
`"|u_p|", "|u_q|", "|u_r|"`	Absolute value of the differential thrust to produce rolling (p), pitching (q), and yawing (r) moments. Derived using "w1", "w2", "w3", "w4".
`"U_p", "U_q", "U_r"`	Differential thrust to produce rolling (p), pitching (q), and yawing (r) moments. Derived using "w2_1", "w2_2", "w2_3", "w2_4".
`"|U_p|", "|U_q|", "|U_r|"`	Absolute value of the differential thrust to produce rolling (p), pitching (q), and yawing (r) moments. Derived using "w2_1", "w2_2", "w2_3", "w2_4".
`"p", "q", "r"`	Quadrotor body rotational rates
`"|p|", "|q|", "|r|"`	Absolute value of the quadrotor body rotational rates
`"roll", "pitch", "yaw"`	Quadrotor attitude

`"sin[roll]", "sin[pitch]", "sin[yaw]"`	Sine of the quadrotor attitude
`"cos[roll]", "cos[pitch]", "cos[yaw]"`	Cosine of the quadrotor attitude
`"x", "y", "z"`	Quadrotor position in the inertial frame (if GPS and/or OptiTrack information is available)
`"u", "v", "w"`	Quadrotor body velocity (if GPS and/or OptiTrack information is available)
`"|u|", "|v|", "|w|"`	Absolute value of the quadrotor body velocity (if GPS and/or OptiTrack information is available)
`"mu_x", "mu_y", "mu_z"`	Advance ratios, see eq. (15) (if GPS and/or OptiTrack information is available)
`"|mu_x|", "|mu_y|", "|mu_z|"`	Absolute value of the advance ratios (if GPS and/or OptiTrack information is available)

6.5 Example - Model identification

Listing 8 presents the `identificationConfig.json` file used in this example. Any changes to this base file will be highlighted where relevant. This is done to demonstrate the effect changing some of the configurable parameters.

Following from the data processing example (section 5.4), filtered data for rows 2, 3, 4, 5, 8 (contain manoeuvres) and 13 (mainly hovering flight) should be processed and available for identification. If not, please refer to section 5.4 to process the data for identification[21].

In the example `identificationConfig.json`, line 2 to line 8 indicate that the flights corresponding to rows 2, 3, 4, 5 and 13 are to be used for identification whereas row 8 is reserved for model validation. The data is to be loaded from the `data/processed/filtered/withGravity` folder (indicating that the filtered data contains the effect of gravity).

A few quick notes before proceeding with the identification scripts:

First, the identification scripts can be run through either `main.py` in the root directory, or through `droneidentification/buildDronePolyModel.py`. For users of the former, ensure that `doIdentification=True` in `main.py`[22] and work with the `identificationConfig.json` file in the root directory. Alternatively, for those working with `buildDronePolyModel.py` directly, modify the `droneIdentification/identificationConfig.json` file and make sure that any relative paths start from the `droneidentification` repository.

Second, as the subsequent examples may require the scripts to be re-run with different configurations, QuaSI may raise a `[ WARNING ]` related to an existing model ID:

```
[ INFO ] IDENTIFYING MODELS FOR:
[ INFO ] MDL-HDBeetle-NN-II-TEST
[ WARNING ] Model: MDL-HDBeetle-NN-II-TEST already seems to exist. Continuing
may override existing models.
[ WARNING ] Current sub-directories are:
    Fx
    taus.json
    trainAndTestIndices.pkl
Do you want to continue anyway (y/n)
```

This is a check to avoid over-writing models which have already been identified. For example, in the above, `Fx` is listed under "[WARNING] Current sub-directories are:", if `Fx` is to be identified in the configuration file, then the existing `Fx` will be replaced with the newly identified one should the identification script continue. Inputting 'n' will prompt users with an input for a new model name, such as to not overwrite existing models. Instead, if overwriting is intended, the scripts will proceed if users input any other key or 'y'.

Listing 8: Identification configuration parameters used in the model identification examples

```
1 {
2     "logging file": {
3         "directory": "data",
4         "filename": "HDBeetle_file_log",
5         "rows of flights to use (all)": ["2-5", 8, 13],
6         "rows of flights for validation": [8]
7     },
8     "filtered data save directory": "data/processed/filtered/
          withGravity",
```

[21] Or, if using `main.py`, set `doProcesing = True` with the appropriate rows specified in `processConfig.json`.

[22] If the filtered data is already available, then the processing step can be skipped by setting `doProcessing = False` in `main.py`. In this example, all flags aside from `doIdentification` may be set to `False`.

```
    "plotting": {
        "show trajectories": false,
        "number of drone stills": 5,
        "show animation": false,
        "animation speed up factor": 1
    },
    "data normalization": {
        "normalize data": false
    },
    "data partitioning": {
        "usable data ratio": 0.9,
        "random partition": true,
        "partition ratio":0.75
    },
    "manoeuvre excitations": {
        "isolate to regions of excitation": false,
        "show isolation results":true,
        "excitation threshold": 0.5,
        "spread": 0.02
    },
    "identification parameters": {
        "show actuator constant estimation": false,
        "polynomial specification file":"droneidentification/
           polynomialCandidates/
           polyCands_DiffSys_quadraticU_symmetric_SIMPLE_FM.json",
        "add bias term fx":true,
        "add bias term fy":true,
        "add bias term fz":true,
        "add bias term mx":true,
        "add bias term my":true,
        "add bias term mz":true,
        "identify fx": true,
        "identify fy": true,
        "identify fz": true,
        "identify mx": true,
        "identify my": true,
        "identify mz": true,
        "prediction interval confidence level": 0.95,
        "polynomial": {
            "regressor cap": 3
        }
    },
    "saving models": {
        "save identified models": true,
        "save directory": "models/HDBeetle",
        "model ID": "MDL-HDBeetle-EXAMPLE"
    }
}
```

6.5.1 Visualizing identification data

Before identifying models, it is always good to visualize the input data to verify that it is suitable for the desired model identification. To this end, QuaSI offers visualization utilities to plot and

animate the trajectories of selected flight logs.

Modify `identificationConfig.json` with:

```
    "plotting": {
        "show trajectories": true,
        "number of drone stills": 5,
        "show animation": true,
        "animation speed up factor": 1
    }
```

and

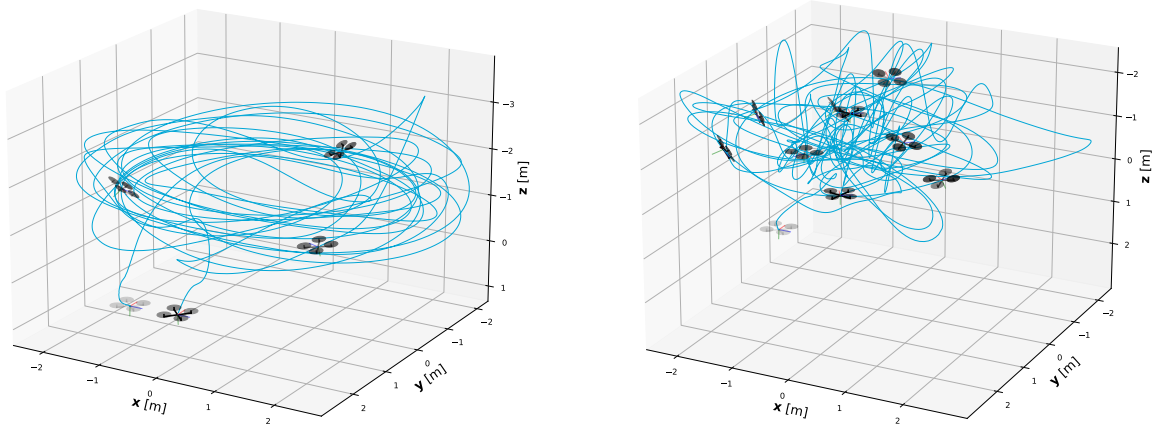
```
        "identify fx": false,
        "identify fy": false,
        "identify fz": false,
        "identify mx": false,
        "identify my": false,
        "identify mz": false,
```

and

```
    "saving models": {
        "save identified models": false,
        "save directory": "models/HDBeetle",
        "model ID": "TrajectoryExample"
    }
```

Run the relevant scripts (i.e. `main.py` with `doIdentification = True` in the root directory or `buildDronePolyModel.py` in the `droneidentification` directory). Trajectory plots, such as fig. 12, and animations will show up sequentially in the order of the "rows of flights to use (all)" configuration parameter. In this example, data corresponding to row 2 is visualized first. Through the "number of drone stills" parameter, users can configure the number of quadrotors which appear in the 3-D trajectory plot. This is useful to get an indication of the orientation of the quadrotor throughout the flight.

Furthermore, in the trajectory plots, time is encoded through the opacity of the quadrotor frame. The more opaque, the more recent. Likewise, for both the trajectory and animation



(a) Row 2 flight trajectory (5 stills)

(b) Row 4 flight trajectory (10 stills)

Figure 12: Example trajectory plots of flights of the HDBeetle quadrotor.

plots, the rotor speed information is embedded in the opacity of the individual rotors for which a fully opaque rotor corresponds to the maximum rotor speed. This is done to give an indication of what the quadrotor is doing at a given instant. For example, the leftmost quadrotor in fig. 12b appears to be yawing.

Note that the animation plots will not appear until after the trajectory plots are closed. Moreover, the animation plot may be closed at any time to continue with the script. All of the plots (trajecotry and animation) are interactive and thus users may change the perspective and zoom.

Revert any changes made back to the original `identificationConfig.json` before proceeding with the subsequent examples.

6.5.2 Nominal model identification

Using the example `identificationConfig.json` file, run the model identification scripts (i.e. `main.py` with `doIdentification = True` in the root directory or `buildDronePolyModel.py` in the directory of `droneidentification`).

The identification scripts first broadcast some of the input parameters as confirmation:

```
[ INFO ] Loading polynomial candidate file:  droneidentification/
 polynomialCandidates/polyCands_DiffSys_quadraticU_symmetric_SIMPLE_FM.json
[ INFO ] IDENTIFYING MODELS FOR:
[ INFO ]       MDL-HDBeetle-EXAMPLE
[ INFO ] Loading filtered data from:  data/processed/filtered/withGravity
```

In this example, all models are scheduled for identification, and will be identified sequentially. Note that, those that encounter `out of memory` errors may need to run the scripts multiple times to identify subsets of models at a time (e.g. first identify force models in run 1 and then moment models in run 2). This can be configured through the "identify <*mdl*>" entry in the `identificationConfig.json` file.

For each model scheduled for identification, the scripts will output information pertaining to the step-wise regression process. Below is an example for F_x:

```
[ INFO ] Identifying polynomial model for Fx...
[-------------------------]
[ INFO ] Initial values
[ INFO ] Selected regressor:  None
[ INFO ] Removed regressor:  None
[ INFO ] Predict square error:  0.06945343311495321
[ INFO ] Coefficient of Determination (R2):  0.815658526937801
[-------------------------]
[ INFO ] Current step:  1
[ INFO ] Bias:  [[-0.15957025]]
[ INFO ] Selected regressor:  (w_tot)*(u^(1.0))
[ INFO ] Removed regressor:  None
[ INFO ] Predict square error:  0.05778834859325183
[ INFO ] Coefficient of Determination (R2):  0.8466259801024811
```

First, the scripts broadcast the current model being identified, in this case F_x. Afterwards, the base model performance is shown. This is the best fit associated with the model composed of only the chosen fixed regressors. For each step in the regressor selection process, the scripts output, in order, the current step number, the bias value (if a bias exists), the regressor selected in the forward pass, any regressor removed (in the backward pass) due to redundancy, the predict square error, and the model fit through the coefficient of determination. Each step is delimited by `[------------------------]`. This type of output will persist until one of the termination conditions is met. In this example for F_x, the termination condition is:

```
[ INFO ] Current step has reached upper limit (k_lim = 3).  Terminating
selection process.
```

Meaning that the step-wise regression procedure has reached the maximum number of permissible steps as specified in `identificationConfig.json`. Following the termination of the model regressor selection process, the identification scripts broadcast its performance and probe for over-fitting. Continuing with the outputs of F_x as an example:

```
[ INFO ] Model performance:
[ INFO ]     RMSE w.r.t training data 0.22951179058002708
[ INFO ]     RMSE w.r.t test data 0.23453132737190033
[ INFO ]     RMSE w.r.t full data 0.23160358469122402
[ INFO ] Relative difference in RMSE between TEST and TRAIN (Fx):
   2.140241496997831 %
[ INFO ]     RMSE w.r.t VALIDATION data 0.23800901720224563
[ INFO ] Relative difference in RMSE between TEST and VALIDATION (Fx):
   -1.4828252879116093 %
[ INFO ] Saving figures...
```

The model performance - in terms of RMSE - is given with respect to the training data, test data, validation data and full data (including validation data). Relative differences between test & training and test & validation RMSEs are also shown. Significant differences between the training and test RMSE suggests that the model overfits the training data. Moreover, as the test RMSE is an estimate of the validation RMSE, these should ideally be the same. A significant difference between these may be indicative of over-fitting or that the validation set contains dynamics absent in the training data. Note that, if `"random partition":false` in `identificationConfig.json`, **then the test RMSE is the validation RMSE**; as there is no test data specified, QuaSI uses the validation data as the test data (but not for training!). By default, the scripts consider any difference in RMSE exceeding 10% to be significant and will output additional information if over-fitting is suspected (see section 6.5.3 for examples on model over-fitting).

In fact, depending on how the data is (randomly) partitioned, some users may encounter such an over-fitting warning for some of the models. In the present example, the scripts caution against the validity of the M_z model:

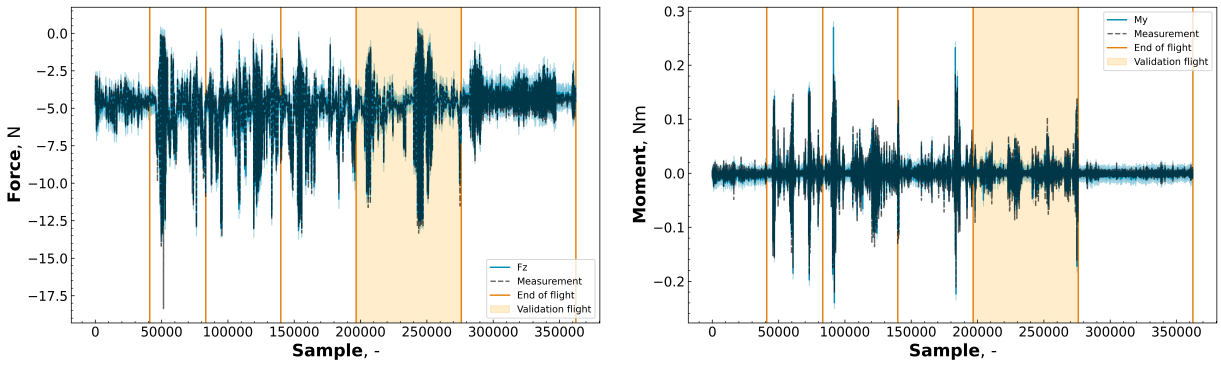
```
[ INFO ] Relative difference in RMSE between TEST and TRAIN (Mz):
  11.590178905332536%
[ WARNING ] The TEST RMSE is over 10% different than the TRAIN RMSE. The
Mz model may be OVERFITTING. Consider removing regressors.
```

Details on how to investigate this warning and interpret the additional results that arise are elaborated upon in another example dedicated to over-fitting (section 6.5.3). Procedures on how to mitigate over-fitting are given in the example of section 6.5.4. In short, however, the warning

raised for the M_z model here is more a symptom of a lack of useful information (i.e. excitations) in the underlying data. In fact, the largest discrepancy in training versus test RMSE emanates from the noisy near-zero measurements (i.e. the interquartile range of the M_z measurements).

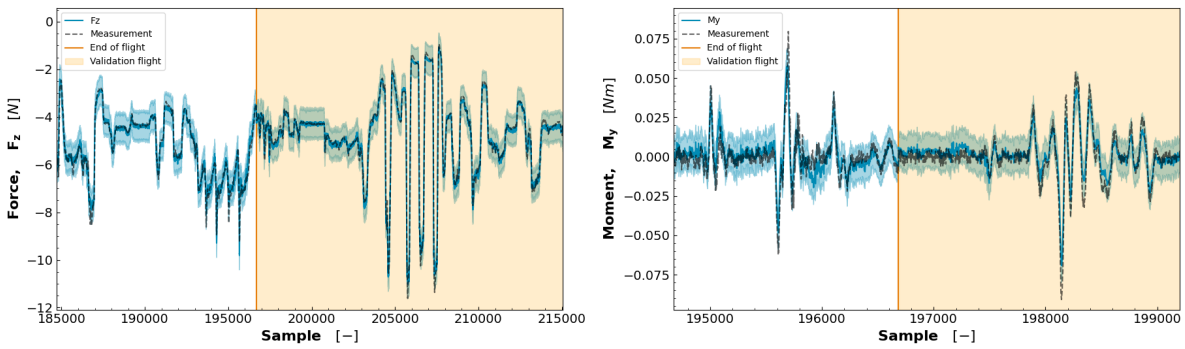
At the end of an identification run, the prediction interval (PI) results for each of the identified models are displayed in the terminal window. The prediction intervals are a measure of the confidence the model has in its predictions, subject to both model and (observed) measurement uncertainties, and thus relate to the reliability of a prediction. Moreover, the "Probability Coverage" should be near the specified confidence level, and if not, the identified model may be invalid (e.g. due to over-fitting). Note, however, that the polynomial PI probability coverage is governed predominantly by the measurement error term, which is highly subjective to the observed measurement data [3]. Hence, slight discrepancies observed (e.g. Coverage Probability = 94.5 < 95) may arise from differences in measurement noise variance between training and validation data, especially if the validation data is rich with large magnitude excitations. More details on interpreting these prediction intervals may be found in the example on over-fitting section 6.5.3.

Finally, the identification scripts generate figures for each of the identified models which compare their predictions to the target measurements. Also shown are the associated prediction intervals. Furthermore, in the plots, individual flights are delimited by vertical lines whereas flights used for validation are highlighted. Figure 13 depicts some examples of these plots, whereas fig. 14 illustrates zoomed-in variants to showcase the model predictive performance. Indeed, the identified models appear to fit the (training and validation) measurement data well.



(a) Force along z-axis F_z (b) Pitching moment, M_y

Figure 13: Comparison of the model predictions with the measurement data.



(a) Force along z-axis F_z (b) Pitching moment, M_y

Figure 14: Zoomed comparison of the model predictions with the measurement data.

6.5.3 Indicators of model overfitting

This example outlines what tools QuaSI provides to aid users in identifying potential over-fitting in their models. Make the following changes to the `identificationConfig.json` file:

```
    "logging file": {
        "directory": "data",
        "filename": "HDBeetle_file_log",
        "rows of flights to use (all)": [2, 8, 13],
        "rows of flights for validation": [8]
    },
```

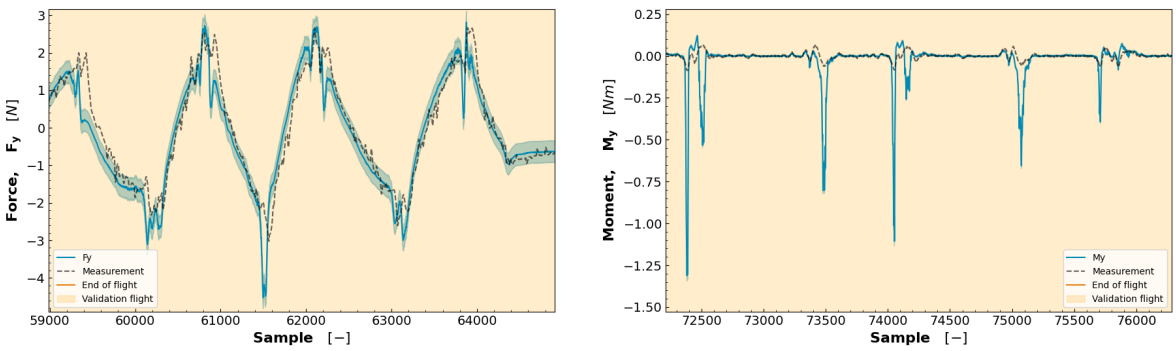
```
        "identify fx": false,
        "identify fy": true,
        "identify fz": false,
        "identify mx": false,
        "identify my": true,
        "identify mz": false,
```

```
    "saving models": {
        "save identified models": true,
        "save directory": "models/HDBeetle",
        "model ID": "MDL-HDBeetle-Overfit"
    }
```

Run the identification scripts (i.e. `main.py` with `doIdentification = True` in the root directory or `buildDronePolyModel.py` in the `droneidentification` directory) and wait for the figures to appear. Note that there will likely be a lot of output warning messages broadcast to the terminal, these will be revisited further on in this example.

From these model predictive performance plots, it is clear that the identified models for F_y and M_y are over-fitting the training data. Figure 15 depicts regions in the validation measurement data where this model fit is particularly poor.

Indeed, the majority of the training data is comprised of hovering flight data which, for F_y and M_y, are mostly just noise. Moreover, the useful information (i.e. excitations in F_y and M_y) stem from an FPV (first person view) flight while the validation flight is LOS (line of sight). These all contribute to the over-fitting problem. While it is difficult to determine if over-fitting occurs in more nuanced cases compared to the example studied here, QuaSI nonetheless computes several metrics to aid users in this endeavour.



(a) Force along y-axis F_y

(b) Pitching moment, M_y

Figure 15: Zoomed comparison of the model predictions with the validation measurement data showcasing over-fitting.

Using model performance metrics between data sets:
The first way to check for over-fitting is through the differences in training, test, and validation RMSE (note that differences in test and validation RMSE only emerge if users have `"random partition": true` in `identificationConfig.json`, otherwise the test and validation RMSE will always be the same!). By default, QuaSI considers relative differences greater than 10% to be significant and will automatically run further analyses. QuaSI conducts further analyses only in situations where the test RMSE is over 10% **greater** than the training RMSE as these are more indicative of over-fitting. Negative ratios mostly arise from differences in data set magnitudes rather than over-fitting. Note that the *opposite* is true for the validation versus test RMSE comparisons; here QuaSI is concerned with negative differences as they correspond to situations where the validation RMSE is greater than the test RMSE.

In this example, the F_y model test RMSE is over 40% larger than the training RMSE, indicating that the model poorly fits the test data and is likely over-fitting:

```
[ INFO ] Relative difference in RMSE between TEST and TRAIN (Fy):
  43.65993608856851 %
[ WARNING ] The TEST RMSE is over 10% different than the TRAIN RMSE. The
Fy model may be OVERFITTING. Consider removing regressors.
```

When the script identifies a potential issue of over-fitting, it runs a further analysis breaking down the RMSE into quartiles for the data. It also generates a figure that presents the distribution of the training, test, and validation data among the quartiles. For F_y, fig. 16 illustrates this breakdown in which the highlighted regions represent the quartiles; 50% of the measurement data is contained in the so-called 'middle interquartile' and the remainder of the data is split equally into the 'Upper' and 'Lower' quartiles (25% respectively). In the leftmost plot, the measurements and model predictions are shown whereas the rightmost plot depicts the distributions of the training, test, and validation data.

The differences between distributions in the right plot of fig. 16 suggest that there exist biases in the training data, visible from the 'humps' on either side of the mean. These actually emerge from the FPV flights. The larger negative hump suggests that this region is more feature rich.

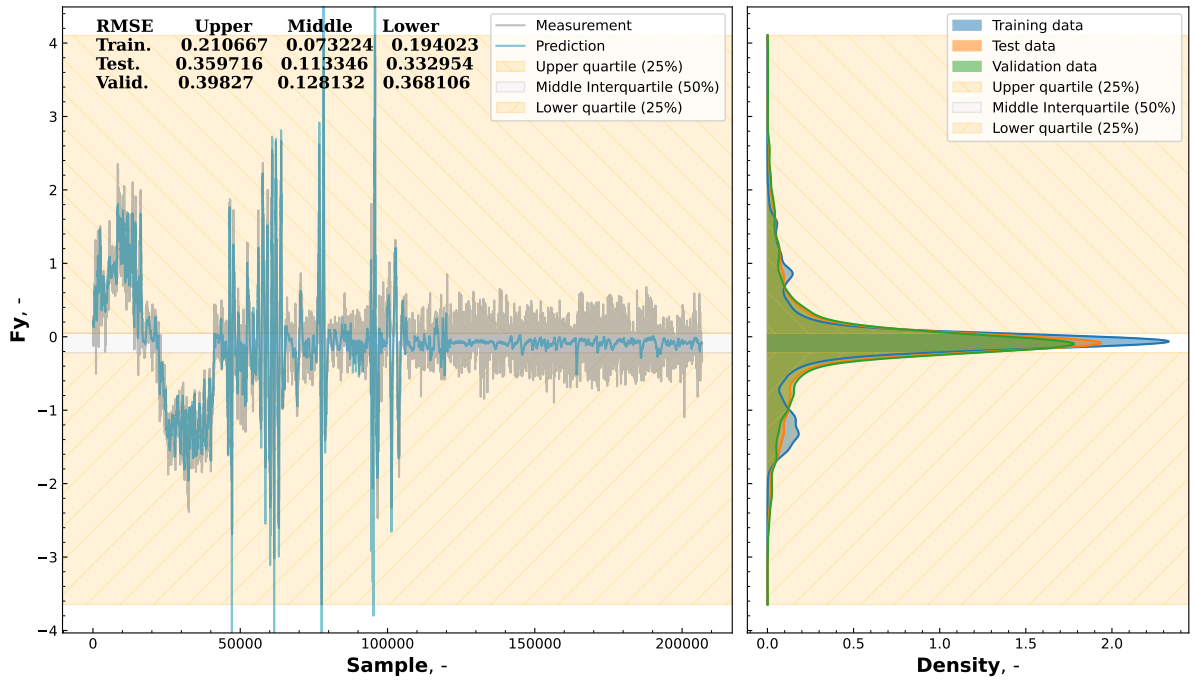


Figure 16: Performance metric (RMSE) quartile breakdown for the F_y data from the example (section 6.5.3) to investigate issues of over-fitting.

Indeed, the RMSEs (tabulated at the top of the left plot of fig. 16) of the 'Lower' are slightly more accurate for all data subsets than their 'Upper' counterparts. Nevertheless, the performance is relatively symmetric as these RMSEs are similar.

Moreover, these RMSE breakdown results are broadcast to the terminal window, including ratios between the various RMSE values:

```
[ INFO ] Splitting RMSE by quartile:
[ INFO ]   Upper 25% of Fy measured range:
[ INFO ]     Training (X) : 0.210667
[ INFO ]     Test (T) : 0.359716
[ INFO ]     Validation (V) : 0.39827
[ INFO ]     Ratio (X/T) : 0.5856491843207541
[ INFO ]     Ratio (V/T) : 1.107180372529933
[ INFO ]   Interquartile Fy measured range:
[ INFO ]     Training (X) : 0.073224
[ INFO ]     Test (T) : 0.113346
[ INFO ]     Validation (V) : 0.128132
[ INFO ]     Ratio (X/T) : 0.6460227079866838
[ INFO ]     Ratio (V/T) : 1.1304470835848932
[ INFO ]   Lower 25% of Fy measured range:
[ INFO ]     Training (X) : 0.194023
[ INFO ]     Test (T) : 0.332954
[ INFO ]     Validation (V) : 0.368106
[ INFO ]     Ratio (X/T) : 0.5827327474527111
[ INFO ]     Ratio (V/T) : 1.1055776871367198
```

From these ratios, particularly (X/T), it can be seen that while the poor RMSE exists throughout the measurement range of F_y, the poor performance is more pronounced for larger magnitudes. Note, however, that the absolute value of the RMSE is dependent on the absolute magnitude of the underlying data and thus may also contribute to the difference between quartile RMSEs seen here. Furthermore, the ratio between test and validation RMSE (i.e. V/T in the terminal output) is only around 10% (as opposed to the 40% with respect to training) indicating that the poor model fit with respect to the validation data is not (predominantly) due to the presence of unobserved dynamics in the validation data (i.e. dynamics not seen in training). In aggregate, then, these discrepancies between train RMSE and test/validation RMSE suggest that the model over-fits the training data throughout the measurement range.

Contrarily, the RMSE results of the pitching moment model M_y reveal a strong asymmetry between the test and validation RMSE for the 'Upper' and 'Lower' quartiles. From the RMSE table in fig. 17, the M_y model performs significantly (almost 300%) worse for negative pitching moments than positive. Indeed, this negative asymmetry is clearly visible in the M_y model predictions in fig. 15b. Moreover, this behaviour can already be anticipated through the final model structure. Through the `SysID.Model.summary()` method, the final model structure and coefficients may be outputted to the terminal (see listing 9).

From the model structure, the U_q^4 term stands out as a problematic regressor. Indeed, this term overpowers U_q^2 for large values of U_q (which is likely to occur in nominal flight since U_q is composed of the square of the rotor speeds!). The question, then, is why does the model elect this model structure?

From the distribution plots of fig. 17, the test and validation data harbor longer and wider tails than the training data and therefore contain a larger proportion of high magnitude M_y data

points (i.e. excitations). The training data which is, in a way, over-exposed to low M_y values from hovering data (it over-fits the low M_y values). Indeed, when the pitching control moment, U_q, is near zero, the M_y asymmetry is negligible. However, for larger values of U_q (as is more pertinent in the test and especially the validation data), the U_q^4 term dominates the opposing U_q^2 regressor and thus the negative overshoot behaviour materializes. Indeed, this model structure is a symptom of a poor selection of training flights. Most importantly, though, is that the RMSE checks in place correctly point to this being an issue.

Listing 9: Terminal line output of the `SysID.Model.summary()` method for the identification example given in section 6.5.3

```
##################################################################
                          Model Summary
##################################################################
Technique:                                        stepwise_regression
Model State:                                                  Trained

------------------------------------------------------------------
----------------------Chosen Regressors-------------------------
Regressor                                                     Value
------------------------------------------------------------------
bias                                                      4.276e-03
q                                                         1.907e-03
U_q                                                       2.422e-06
(q^(3.0))                                                -4.480e-04
(U_q^(4.0))                                              -2.991e-18
(U_q^(2.0))                                               2.867e-10

------------------------------------------------------------------
Total number of terms                                             6

------------------------------------------------------------------
##################################################################
```

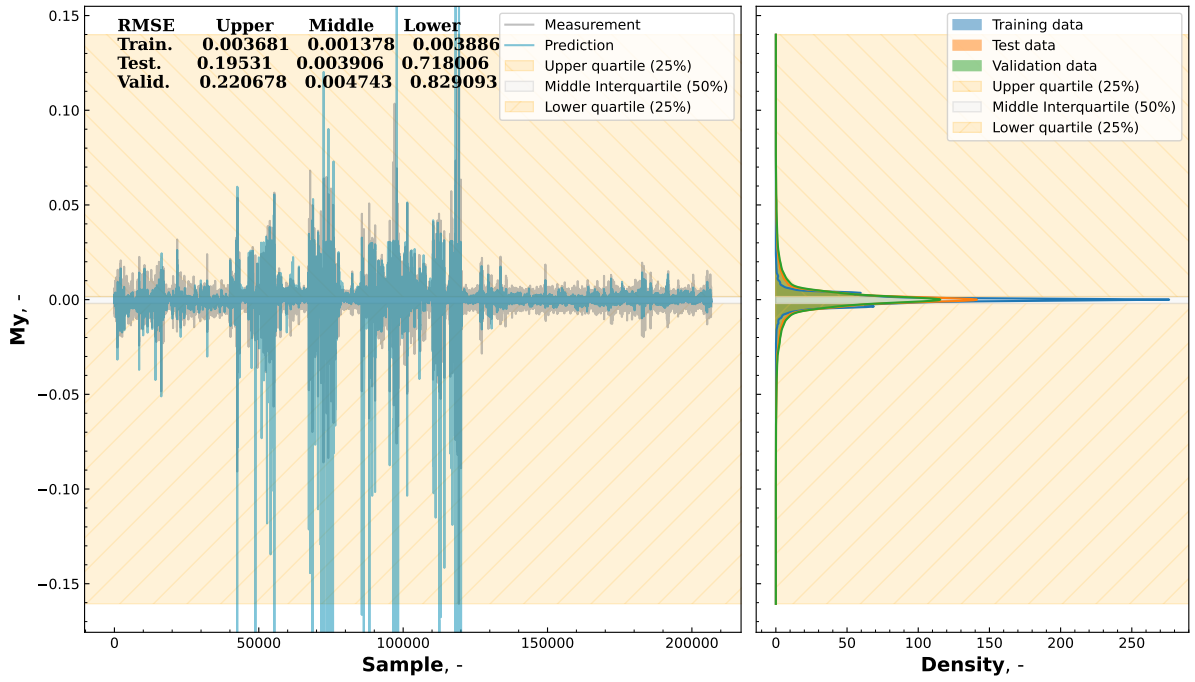


Figure 17: Performance metric (RMSE) quartile breakdown for the M_y data from the example (section 6.5.3) to investigate issues of over-fitting.

Using the prediction intervals:
In addition to the RMSE comparisons, the second validation check made available by QuaSI is to look at the prediction interval results of the models. These are outputted at the end of each identification phase for all of the force and moment models scheduled for identification for the current model ID (here: `MDL-HDBeetle-Overfit`).

In this example, the prediction intervals are built for a confidence level of 95% (`"prediction interval confidence level": 0.95` in `identificationConfig.json`), which means that at least 95% of the data (both training and test, but in the ideal case also validation) should be contained within these bounds. By default, QuaSI computes and outputs this probability coverage for the full data set including training, test, and validation data. If this coverage probability is not met sufficiently (by default, the specified confidence level minus 5%), then it will output additional results to help users isolate the issue.

Something similar to the following should be shown in the terminal:

```
[ INFO ] Polynomial Fy - Prediction Interval results (Confidence level =
  95.0)
[ INFO ] Probability Coverage:  89.8285404652243
[ INFO ] Normalized mean PI width:  0.5820998676810208
[ WARNING ]   Probability Coverage for Fy <= 90.0, model may be overfitting
                or validation data contains unobserved dynamics.
[ INFO ]     Probability Coverage (Train):  93.1345876148421
[ INFO ]     Probability Coverage (Test):  86.9855124628813
[ WARNING ]   Given difference in probability coverage between training
                and test data, model overfits or test data ill-constructed

[ INFO ] Polynomial My - Prediction Interval results (Confidence level =
  95.0)
[ INFO ] Probability Coverage:  86.03601424313968
[ INFO ] Normalized mean PI width:  0.01104154554011039
[ WARNING ]   Probability Coverage for My <= 90.0, model may be overfitting
                or validation data contains unobserved dynamics.
[ INFO ]   Probability Coverage (Train):  95.07460812422828
[ INFO ]   Probability Coverage (Test):  78.26329523980922
[ WARNING ]   Given difference in probability coverage between training
                and test data, model overfits or test data ill-constructed
```

The indicator of the total prediction interval coverage is denoted by `Probability Coverage` in the terminal output whereas `Normalized mean PI width` describes the average width of these prediction intervals, normalized with respect to the range of the measurement data. Indeed, the (total) probability coverage values are significantly lower than the specified confidence level of 95% suggesting that the identified models are invalid across the full data set. At the very least, the identified models cannot be used for the validation data or similar data sets. While, in general, it may be the case that the validation data contain dynamics unobserved during training which explain the reduction in probability coverage, the test data probability coverage results here are also much lower than the corresponding training probability coverage values. This indicates that the model is over-fitting, or that the data is partitioned poorly[23]. Indeed, the coverage probability of the test data should be similar to the training data for a valid model as

[23] In some unlucky cases where the test data is composed of a small portion of the overall data, then some partitions may allocate a disproportionate amount of excitation data in the test subset. That is, the test data is sometimes not representative of the training data.

they are derived, randomly, from the same data series. The additional `[ WARNING ]` messages in the terminal output summarize this conclusion. In the event that the training and test coverage probabilities are similar enough, then the `[ WARNING ]` messages will indicate that perhaps the validation data contains unobserved dynamics.

A reasonable conclusion from these prediction interval results is that the identified models are only valid for the training data envelope. In other words, they over-fit the training data significantly. Taken in tandem with the RMSE validation analysis, it is clear that this model over-fits the training data.

Mitigating over-fitting issues due poorly conditioned training data:
The underlying issue for the over-fitting in this example is that the training data is not constructed well; there is an over-abundance of hovering data and insufficient excitations. While an obvious solution is to simply gather more data, the partitioning of the available data may also be manipulated to improve model performance. Intuitively, one way to do this is to isolate the data set to regions of relevant excitation. An example of this is given in section 6.5.4. However, perhaps a simpler approach to achieve this is to change the flights used for training and validation. In fact, in the example used here, swapping the validation flight to row '2' resolves the over-fitting issue. To check this, make the following changes to the `identificationConfig.json` **used in this example**:

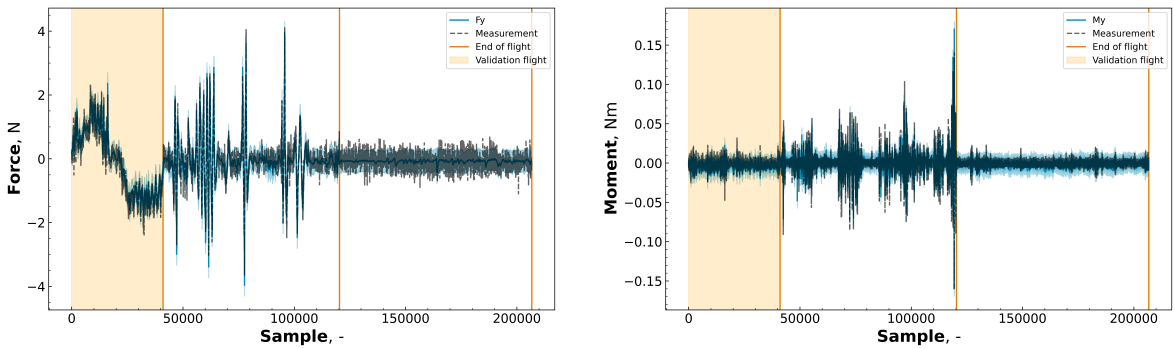
```
2     "logging file": {
3         "directory": "data",
4         "filename": "HDBeetle_file_log",
5         "rows of flights to use (all)": [2, 8, 13],
6         "rows of flights for validation": [2]
7     },
```

and

```
49    "saving models": {
50        "save identified models": true,
51        "save directory": "models/HDBeetle",
52        "model ID": "MDL-HDBeetle-Overfit-FIX"
53    }
```

Below are the relevant validation check messages outputted to the terminal with the over-fitting strategy applied. Indeed, the issue of over-fitting is largely resolved and the RMSE and prediction interval checks do not indicate significant over-fitting of the data. Figure 18 depicts the resultant, and much improved, performance plots (especially for M_y).

```
[ INFO ] Relative difference in RMSE between TEST and TRAIN (Fy):
  -0.010123687316877295 %
[ INFO ] Relative difference in RMSE between TEST and VALIDATION (Fy):
  -0.8419950453026246 %
[ INFO ] Relative difference in RMSE between TEST and TRAIN (My):
  -1.8448207535930976 %
[ INFO ] Relative difference in RMSE between TEST and VALIDATION (My):
  2.394408658198415 %
[ INFO ] Polynomial Fy - Prediction Interval results (Confidence level =
  95.0)
[ INFO ] Probability Coverage:  93.52237101830708
[ INFO ] Polynomial My - Prediction Interval results (Confidence level =
  95.0)
[ INFO ] Probability Coverage:  95.22245229709331
```



(a) Force along y-axis F_y

(b) Pitching moment, M_y

Figure 18: Zoomed comparison of the model predictions with the validation measurement data with over-fitting data partitioning fix applied.

6.5.4 Identification with manoeuvre isolation

This example explores the use of the built-in manoeuvre isolation methods to focus the training data on informative regions for identification. The employed isolation algorithms are described in more detail in section 9.4.1. This example will treat cases where manoeuvre isolation is beneficial to model identification and cases where it may in fact be detrimental.

Beneficial manoeuvre isolation

Depending on the random partition of training and test data in the nominal identification example (i.e. section 6.5.2), QuaSI likely raises a `[ WARNING ]` message for M_z model pertaining to the possibility that the M_z model may be over-fitting. It is notoriously difficult to obtain good moment models for the quadrotor, especially in terms of the yawing moment M_z, given that relevant excitations are often short and sparse. Thus, this example applies manoeuvre isolation in order to improve the M_z model performance and validity.

To this end, modify the standard example `identificationConfig.json` file with:

```
23      "manoeuvre excitations": {
24          "isolate to regions of excitation": true,
25          "show isolation results":true,
26          "excitation threshold": 0.15,
27          "spread": 0.02
28      },
```

```
38          "identify fx": false,
39          "identify fy": false,
40          "identify fz": false,
41          "identify mx": false,
42          "identify my": false,
43          "identify mz": true,
```

```
49      "saving models": {
50          "save identified models": true,
51          "save directory": "models/HDBeetle",
52          "model ID": "MDL-HDBeetle-EXAMPLE-Mz"
53      }
```

Then, run the identification scripts (i.e. `main.py` with `doIdentification = True` in the root directory or `buildDronePolyModel.py` in the `droneidentification` directory).

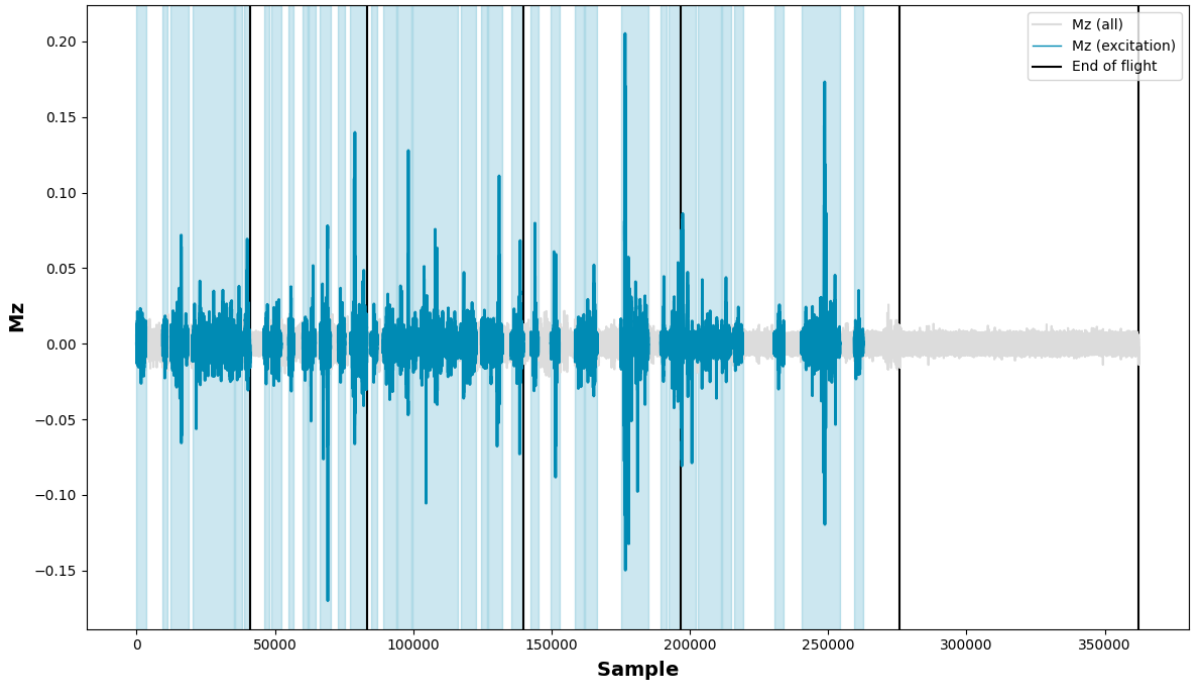


Figure 19: Manoeuvre isolation algorithm results for the yawing moment model M_z in example section 6.5.4.

When running the identification routine with manoeuvre isolation active and `"show isolation results": true`, figures of the isolation algorithm results will be shown. Figure 19 depicts such a plot for the current example where the highlighted regions denote the isolated manoeuvres. Overall, the isolation algorithm produces satisfactory results (the algorithm behaviour is configurable through the hyper-parameters `"excitation threshold"` and `"spread"` in the `"manoeuvre excitations"` configuration parameters). Note that the entire hovering flight (last flight in fig. 19) is discarded whereas the majority of the other flights are kept due to overlap between the pre- and post- manoeuvre `"spread"`. While perhaps reasonable, it has implications on the model performance in this region which manifests in poor local performance. Ideally, there is a healthy balance between regions of excitation and hovering data to ensure that the identified models to not over-fit either.

To contextualize the identification results, table 3 summarizes the performance metrics between the M_z identified with and without manoeuvre isolation, subject to identical training and test data partitions. Indeed, the M_z model with manoeuvre isolation active performs *marginally* better overall and over-fits the training data to a lesser degree (through better validation performance). This performance advantage is also visible in the prediction comparison plots during manoeuvres in fig. 20. The price, however, is poorer hover performance, which materializes in the worse test RMSE.

It is up to users to determine if this trade-off is worthwhile. Note, however, that while hovering flights may not be useful for moment model identification, they are essential for force models, especially (hover) thrust in F_z.

Model	$\mathbf{M_z}$ **with manoeuvre isolation**	$\mathbf{M_z}$ **without manoeuvre isolation**
Final R2	0.293	0.217
Validation RMSE	8.55e-03	8.72e-03
Test RMSE	8.03e-03	7.99e-03

Table 3: Performance summary of the identified yawing moment models subject to identical training, testing, and validation data sets.

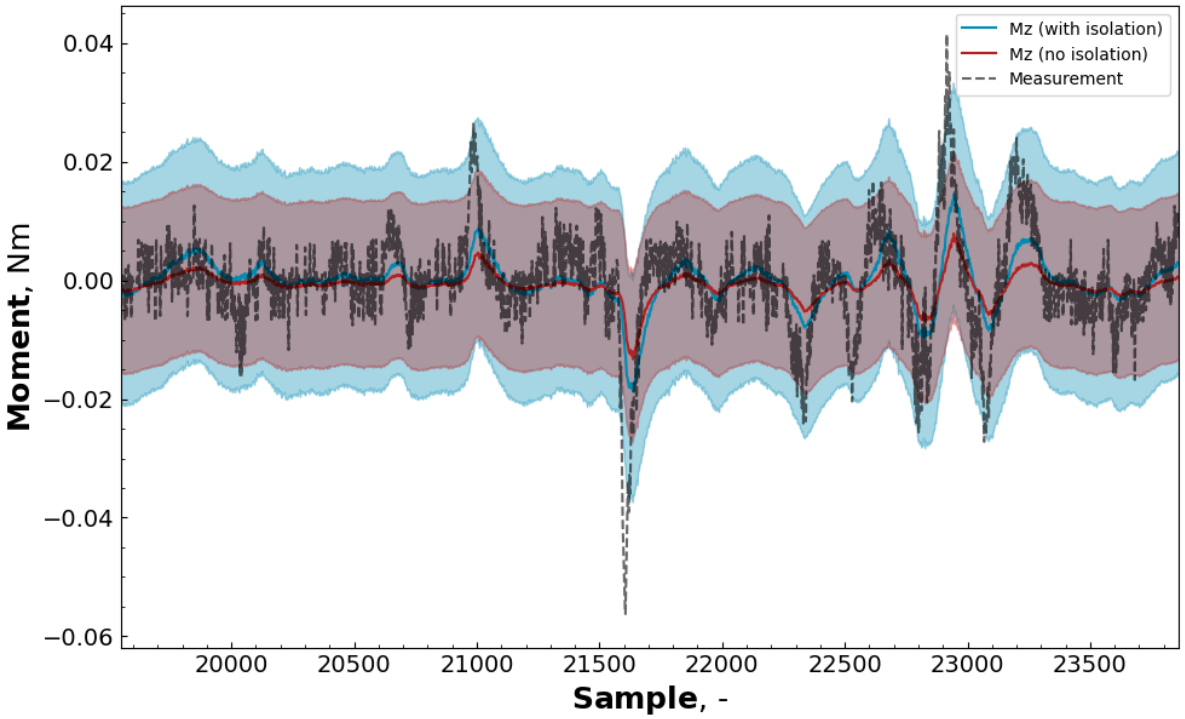


Figure 20: Yawing moment M_z performance comparison between the M_z model identified with (—) and without (—) manoeuvre isolation, centered around a yawing manoeuvre, for the example section 6.5.4.

Detrimental manoeuvre isolation

For cases where the flight data may be feature rich, isolation algorithms are likely unnecessary and may actually deteriorate model performance. As aforementioned, a good balance between regions of excitation and hovering data is essential to ensure that models do not over-fit either region. Indeed, the quadrotor data made available in the `QuaSI` repository is relatively feature rich, especially in terms of the thrust force, F_z and subsequent models (often) do not benefit from manoeuvre isolation.

To illustrate this, make the following changes to the example `identificationConfig.json`:

```
"logging file": {
"directory": "data",
"filename": "HDBeetle_file_log",
"rows of flights to use (all)": [2, 3, 8, 13],
"rows of flights for validation": [2]
},
```

```
    "manoeuvre excitations": {
        "isolate to regions of excitation": true,
        "show isolation results":true,
        "excitation threshold": 0.6,
        "spread": 0.02
    },
```

```
        "identify fx": false,
        "identify fy": false,
        "identify fz": true,
        "identify mx": false,
        "identify my": false,
```

```
        "identify mz": false,,
```

```
    "saving models": {
        "save identified models": true,
        "save directory": "models/HDBeetle",
        "model ID": "MDL-HDBeetle-Iso-FzBad"
    }
```

Run the identification scripts (i.e. `main.py` with `doIdentification = True` in the root directory or `buildDronePolyModel.py` in the `droneidentification` directory).

Figure 21 previews the manoeuvre isolation results for the F_z model used in this example. Again, the isolation results appear reasonable. Note that the isolation here is more strict than for the M_z as the `"excitation threshold"` has been increased. If using manoeuvre isolation, users are encouraged to experiment with this parameter to promote the best results for their applications.

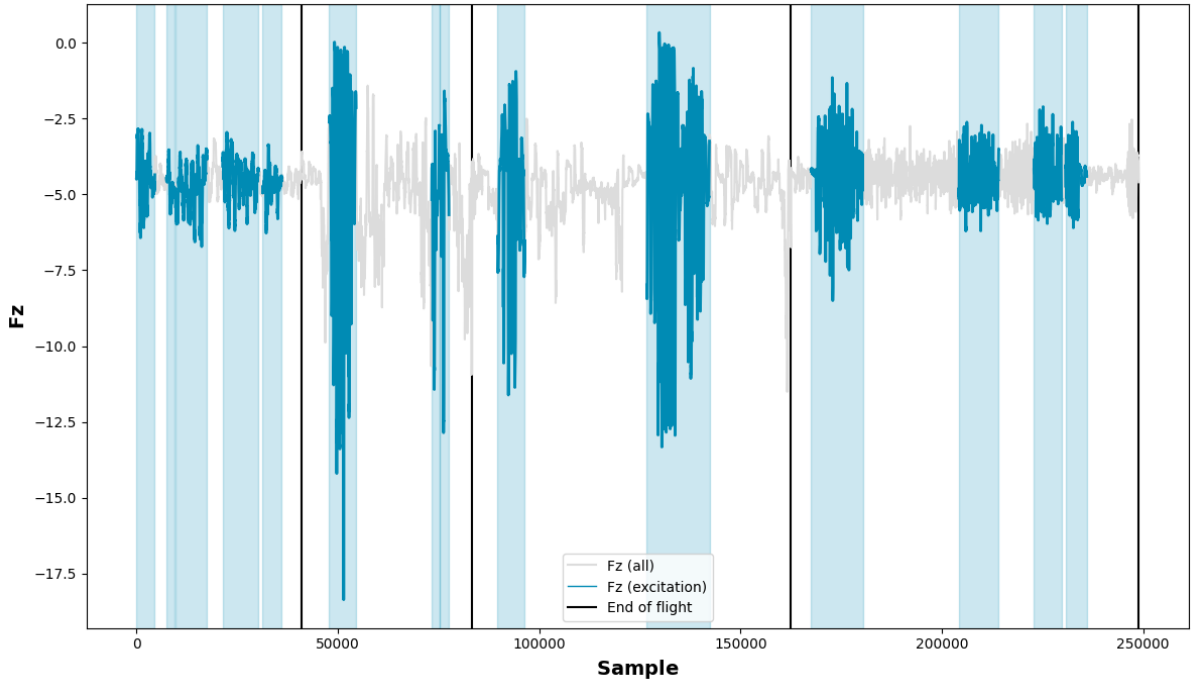


Figure 21: Manoeuvre isolation algorithm results for the force along the quadrotor z-axis F_z in example section 6.5.4.

As a means of comparison, the F_z model is also identified on an identical data set (i.e. equivalent training, test, and validation partitioning) but without the maneouvre isolation active. The performance results are summarized in table 4 and visualized through fig. 22. In the figure, there is a negligible performance different between the identified models for the training data, but a clear deterioration in performance for the F_z model identified with manoeuvre isolation for the validation flight. These results are confirmed through the associated RMSE values in table 4; the F_z identified with maneouvre isolation performs 20% and 40% worse on the test and validation data sets respectively. While the training final R2 (i.e. model fit) is better, these results in aggregate are suggestive of over-fitting.

Note that, when using manoeuvre isolation, the training RMSE typically grows in magnitude as the model error residuals, in absolute terms, grow in tandem. For such scenarios, it is perhaps more useful for users to check the validation versus test RMSE to probe for over-fitting.

Model	$\mathbf{F_z}$ with manoeuvre isolation	$\mathbf{F_z}$ without manoeuvre isolation
Final R2 (Training)	0.964	0.961
Validation RMSE	0.359	0.255
Test RMSE	0.354	0.296

Table 4: Performance summary of the identified force F_z models subject to identical training, testing, and validation data sets.

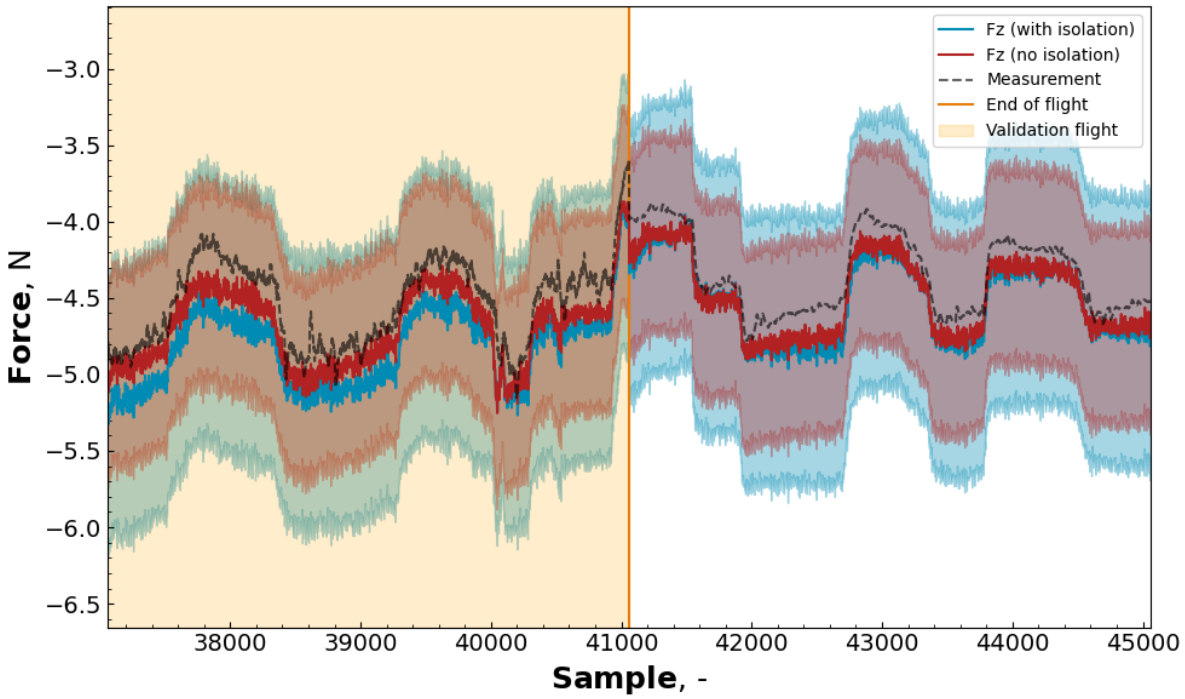


Figure 22: Force F_z performance comparison between the F_z model identified with (—) and without (—) manoeuvre isolation, for both validation and training data for the example section 6.5.4.

6.6 Standalone Models

The `SysID.Model` objects contain a lot of information and add some overhead which may be unnecessary when using the models in practice (e.g. in simulations). Moreover, the `SysID.Model` objects are not directly portable; they are dependent on the `sysidpipeline` repository.

Thus, in an effort to make them more portable, and to retain only the essential features of the model (e.g. polynomial coefficients and regressors), QuaSI contains some utility scripts to make so-called standalone models. These standalone models also handle all the data processing necessary to build the model regressors from the state vector (velocity, body roational rates, and attitude) and rotor speeds. In this way, interacting with the model object is standardized and straight forward: users need only call `Model.predict(stateVector, rotorSpeeds)`. While useful in theory, in practice the standalone models can be finicky (hence the strict versioning for QuaSI). To use these standalone models, the environments need to be (mostly) consistent such that the `.pkl` files can be unpickled. In any case, there should be no issue if deploying these standalone models within your own workflow (where environments are likely consistent). Regardless, it is recommended to use these standalone models as they are more efficient at making predictions than the `SysID.Model` objects from which they are derived.

Standlone models are created via the `droneidentification/makeStandalonePolyModel.py` which reads the `standaloneConfig.json` file.

A note on making modifications to QuaSI:

Those that wish to modify/create their own processing steps (e.g. adding variables for model identification beyond those in section 6.4.1, or extra identification processing steps beyond those currently in QuaSI) in the model identification scripts: these changes need to also be made correspondingly in the standalone scripts (i.e. `makeStandalonePolyModel.py`). Any modifications will likely need to be made in the `DronePolynomialModel` class, which may be found in `makeStandalonePolyModel.py`. Users are directed to check the variables in `self.columns` and `self.ncolumns` in the `DronePolynomialModel.__init__()` method to make modifications to the base variables available for polynomial construction. All users making modifications may also need to make modifications in the `DronePolynomialModel.droneGetModelInput()` method. This function essentially just transforms the state vector into the polynomial basis variables. The underlying `PolynomialModel` class (from which `DronePolynomialModel` inherits) sources from this database and allocates the appropriate variables to the polynomial regressors, so this class can mostly be left untouched.

6.6.1 Example - Creating standalone models

Using the polynomial model from the nominal model idenitification example (i.e. 'MDL-HDBeetle-EXAMPLE'), this example outlines how a standalone model can be created from an identified quadrotor polynomial model. If this model has not been identified, then follow the nominal model identificaiton example to create it. Alternatively, for users that wish to port one of their own models, simply make the appropriate changes to the `standaloneConfig.json` file. This example continues with the following `standaloneConfig.json` configuration file:

Listing 10: Standalone model configuration parameters used in the example (section 6.6.1)

```
{
    "model path": "models/HDBeetle",
    "model ID": "MDL-HDBeetle-EXAMPLE",
    "droneConfig path": "data",
    "droneConfig name": "HDBeetleConfig.json",
    "make moment DiffSys": true,
    "DiffSys u": ["U_p","U_q","U_r"]
}
```

Run the standlone script (i.e. `main.py` with `makePortableModel = True` in the root directory or `makeStandalonePolyModel.py` in the `droneidentification` directory). The script will output various messages to the terminal throughout the model porting process. In this example, the first message indicates that actuator dynamics (i.e. rotor rate constant) information has been identified in the underlying model and it will add these to the portable version:

```
[ INFO ] Found actuator dynamics information, adding to standalone model.
```

Subsequently, the standalone scripts load each of the force and moment models, for example, for F_x

```
[ INFO ] [makeStandAlonePolyModel.py] Loading Fx (normalized:  False,
  gravity:  True)
[ INFO ] Flag for uses induced velocity could not be found in metadata,
  will confirm with model regressors.
```

For each model, the script broadcasts whether it is normalized and whether the effect of gravity has been removed (as a verifcation check for users). Moreover, here, the scripts also provide two additional messages pertaining to the "induced velocity". Essentially, in the identified models, it is unclear whether the induced velocity is used or not. Hence, the standalone script scans all the model regressors and determines if terms associated with the induced velocity are present. If not, then it updates the standalone model information appropriately and skips the induced velocity calculation steps (as these can add quite some overhead) for efficiency:

```
[ INFO ] Searched polynomials and updated uses induced velocity to False
```

Finally, if no errors are thrown, then the standalone model has been successfully created. To verifty this, check your model folder (here `models/HDBeetle/MDL-HDBeetle-EXAMPLE`) for a subfolder called `standalone` or `standaloneDiff` (depending on whether `"make moment DiffSys"` is true or not). This subfolder should contain various `".pkl" files`.

For users of the `main.py` file, there are some additional fail-safes in place. If the specified `"model ID"` cannot be found in the `"model path"`, then the `main.py` will broadcast:

```
[ INFO ] Failed to make portable model with current standaloneConfig.json.
Attempting to reconstruct from identificationConfig...
```

Meaning that the script will attempt to make a portable model of the `"model ID"` in the `droneIdentificationConfig.json` file, under the assumption that this model is complete and has been identified. This is done such that those seeking to run QuaSI in one go need not modify the scripts with the correct `"model ID"` everywhere, it may instead be inferred from previous steps in QuaSI.

7 Simulation

(Identified) quadrotor models may be deployed in a Python-based simulation environment to develop controllers, (numerically) validate models, or run other analyses (e.g. forward reachable set computation). In principle, the base simulation (`sim`) class is not restricted to quadrotor simulations and may be used to simulate other systems as well. However, there are several quadrotor specific utilities available for convenience (such as the quadrotor animation and included controllers).

The base quadrotor simulation can be managed from the `droneSimConfig.json` file, the contents of which are described in section 7.1. Those that wish to have more control over the simulation environment should familiarize themselves with the simulation architecture. To this end, a brief overview of the simulation architecture is presented in section 7.2. Useful simulation utilities are also highlighted in section 7.3. Finally, examples on how to use the simulation - including how to add custom modules - are presented in section 7.4.

7.1 Configuration files - droneSimConfig.json

The Python simulation environment can be (mostly) configured through the `droneSimConfig.json` file. Listing 11 summarizes these configurable parameters, supplemented with comments.

Listing 11: Configurable entries of the droneSimConfig.json file with comments for clarity preceeded by %

```
{
    % Model configuration parameters
    "model":{
        % Whether an indentified model should be used. If False
            then the built-in SimpleModel will be used.
        "use identified model":true,
        % Directory of the model folder
        "model path":"models",
        % Name of the model
        "model ID":"MDL-HDBeetle-NN-II-TEST-01",
        % Whether the bias term should be removed (artificially) in
            the simulation. This effectively removes asymmetries in
            the moment and force models.
        "correct for moment asymmetries":true
    },
    % Noise configuration parameters
    "noise":{
        % Whether noise should be added to the simulation. If true,
            then noise is added
        "add":true,
        % RNG seed to use
        "seed":737400
    },
    % Controller configuration parameters
    "controller":{
        % Name of the controller to use for the simulation, INDI
            based on [4]
        "controller":"drone_INDI"
    },
    % Actuator configuration parameters
    "actuator":{
        % Name of the actuator model to use for the simulation
```

```
        "actuator":"rotorDynamicsV2"
    },
    % Simulation time configuration parameters
    "simulation":{
        % Time step of the simulation
        "time step":0.004,
        % Total simulation duration
        "duration":10
    },
    % (Basic) Reference configuration
    "references":{
        % Whether a (pre-defined) linear position tracking
           reference should be followed
        "do position track":false,
        % Whether a (pre-defined) circular position tracking
           reference should be followed
        "do circle track":true
    },
    % (Basic) Initial conditions
    "initial state conditions":{
        % Initial conditions for attitude, in radians
        "attitude":[0, 0, 0],
        % Initial conditions for position, in meters
        "position":[0, 0, 0]
    },
    % (Basic) Rotor fault configuration parameters
    "faults":{
        % Whether a (partial actuator) fault should be injected
           during the simulation
        "inject failure":false,
        % Time at which the fault occurs, in seconds
        "time of failure":1,
        % The ratios of the each rotor health after the failure. 0
           corresponds to complete loss of a rotor whereas 1
           indicates a nominally functioning rotor.
        "rotor health after failure":[0, 1, 0, 1]
    },
    % Animation parameters
    "animation":{
        % Whether the quadrotor animation should be shown during
           the simulation
        "show":true,
        % Whether the simulation should be saved, requires ffmpeg
        "save":false
    }
}
```

For more control over the simulation environment, users are directed to the documentation of the `dronesim` repository and the `droneSim.py` file.

7.2 Simulation architecture

In general, a simulator instance is initialized by passing the relevant modules to the main simulator object through the `sim.sim(model, EOM, controller, actuator)` method. For the quadro-

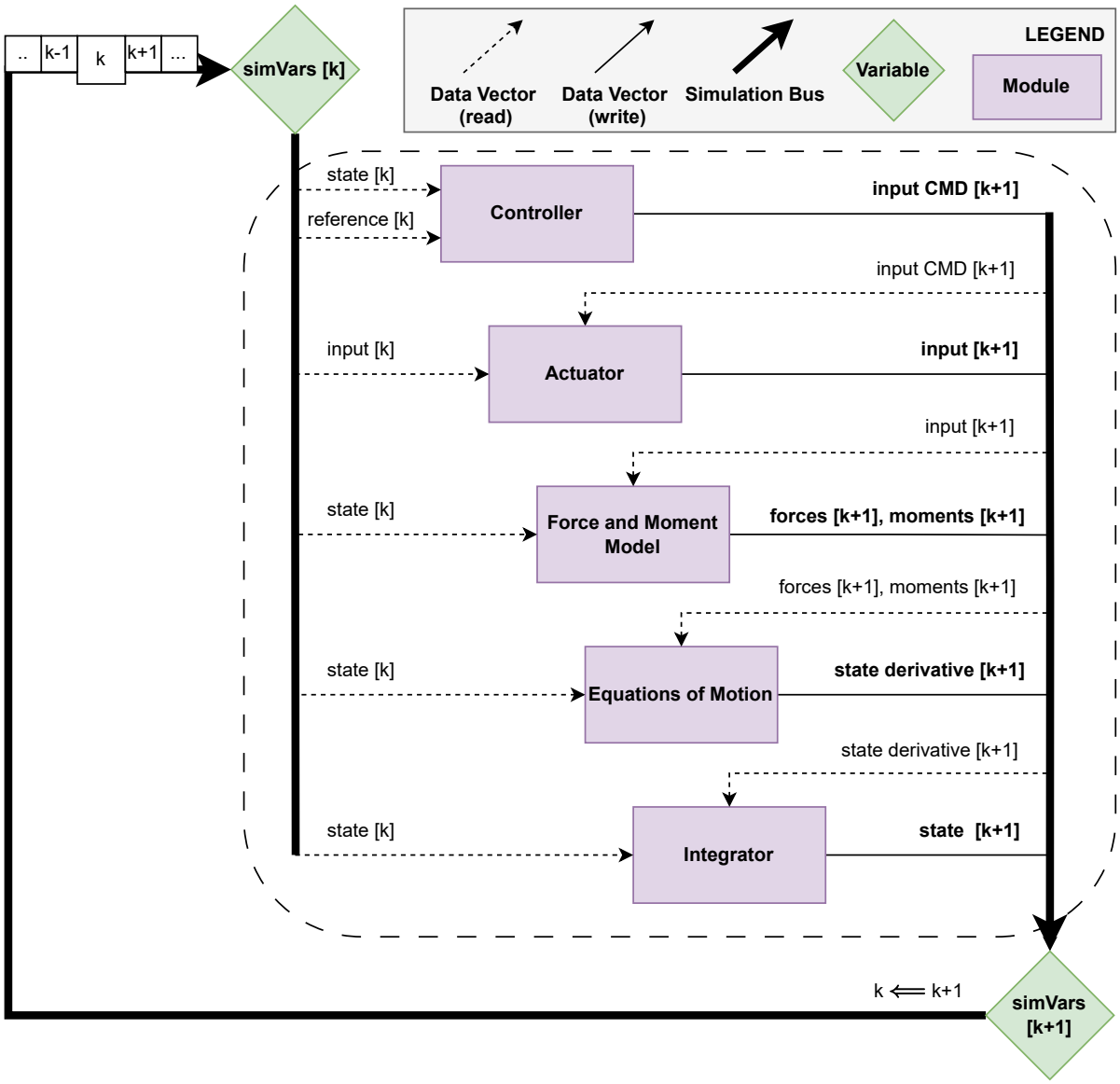


Figure 23: Schematic overview of the Python `droneSim` simulation flow at each step, k

tor simulations here, two keyword arguments are additionally passed for the visualization of the simulation: `animator = animator, showAnimation = True`. Moreover, before simulations can be run, the initial state of the simulator should be specified. This is done through the `sim.setInitial(time, state, inputs, reference)` method. If the simulator does not have an initial state, it cannot be run.

The core of the simulation environment is the `sim.run()` method of `sim.py`. This method hosts the main simulation loop and manages the interactions between the core simulation modules: controllers, actuators, models, integrators, and animators. The main simulation variables, such as the state vector, are stored in the `sim.simVars` dictionary. In this way, each simulation module (even custom user modules) can interact with other modules in the simulation environment. It is important to highlight that the modules do not necessarily interact with each other directly, but rather, through the `sim.simVars` data bus. A schematic overview of the simulation flow, at each step k, is illustrated by fig. 23 in which this read (- -) and write (—) behaviour is emphasized. This architecture also facilitates the injection of additional steps at any point in the simulator (see example in section 7.4.3). These steps are also summarized in the following:

1. The **controller** makes a desired input and writes this to the commanded inputs of the next step $k+1$: `simVars["inputs_CMD"][k+1]`. Typically, these are based on observations of the state at the current step, k, (i.e. `simVars["state"][k]`) and any other information

necessary for the controller available at step k or before.

2. These commanded inputs are then passed to the **actuator** which translates these commands to a true input for step $k+1$, based on some model of the actuator. That is, the actuator writes to `simVars["inputs"][k+1]`.

3. The current state, `simVars["state"][k]`, and desired input, `simVars["inputs"][k+1]`, are then passed on to the **model** to compute the forces and moments of step $k+1$: `simVars["forces"][k+1]` and `simVars["moments"][k+1]` respectively.

4. These forces and moments are then transformed into the state derivative of step $k+1$ through the **equations of motion** (EOM). `simVars["stateDerivative"][k+1]` hosts these state derivatives.

5. Finally, the state of step $k+1$ (i.e. `simVars["state"][k+1]`) is updated using the selected **integrator** and the state derivative information.

7.2.1 Main simulation bus - simVars

Most of the data produced during a simulation is stored in the `sim.simVars` data bus. The `simVars` variable itself is a dictionary which hosts various data streams and modules. These entries are summarized in table 5. Users may retrieve their desired data object through the key, for example, the state can be obtained through `state = sim.simVars['state']`. Note that various simulator modules are also made available in `sim.simVars` (for example, the model can be accessed through `sim.simVars['model']`). There are essentially no requirements imposed on these modules aside from some basic interfacing for compatibility with the `sim.run()` main loop. For more information on the basic interfacing, users can run the `buildMySim.py` script in the `dronesim` repository. This script sets up a documented skeleton of a general (i.e. not quadrotor specific!) simulation environment. This will create files with the class methods and function outputs expected by the simulator (`sim.sim`). See section 7.5 for a more detailed example.

Table 5: Variables inside the `sim.simVars` data bus

`sim.simVar`[**Variable Name**]	**Description**
`time`	Simulation time vector (`numpy.array` of shape [N[24], 1])
`T`	Simulation period (`None` if simulation has no pre-defined end period, else `float`)
`dt`	Simulation time step (`float`)
`currentTimeStep_index`	Simulation step, used for indexing (`int`)
`state`	State vector (`numpy.array` of shape [N, 1, X[25]])
`state_noisy`	Noise contaminated state vector, if noise is active in simulation (`numpy.array` of shape [N, 1, X])
`stateDerivative`	Derivative of the state vector (`numpy.array` of shape [N, 1, X])
`stateDerivative_noisy`	Noise contaminated derivative of the state vector, if noise active in simulation (`numpy.array` of shape [N, 1, X])

[24] N is the total number of data points
[25] X is the number of states

quat	Quaternion representation[26] of orientation (None if no orientation information, else numpy.array of shape [N, 1, 4])
quatDot	Derivative of quaternion vector (None if no orientation information, else numpy.array of shape [N, 1, 4])
reference	State reference vector (numpy.array of shape [N, 1, X])
inputs	System input vector, after actuator dynamics (numpy.array of shape [N, 1, U[27]])
inputs_CMD	Desired system input vector, output of controller (numpy.array of shape [N, 1, U])
forces	Body force vector (numpy.array of shape [N, 1, 3])
Moments	Body moment vector (numpy.array of shape [N, 1, 3])
model	Simulator model module (models/model object. For quadrotors these are: droneModel.model or droneModel.SimpleModel objects)
controller	Simulator controller module (controllers/controller object. E.g. droneNDI.controller)
EOM	Simulator equations of motion function (object or function)
actuator	Simulator actuator module (actuators/actuator object. E.g. droneRotors.rotorDynamicsSaturation)
integrator	Simulator integration scheme (function. Available integration schemes can be found in funcs/integrators.py. Default is funcs/integrators.droneIntegrator_Euler)
integrator_properties	Simulator integration scheme configurable parameters, if any (dict)

7.3 Simulation utilities

7.3.1 Quaternion functions

Internally, the simulation uses quaternions to keep track of the quadrotor's (or, indeed, any other body's) attitude. These are initialized from the inital euler angles. There are various utility scripts made available to those that would like to make use of quaternions. These can be found in dronesim/funcs/angleFuncs.py. A few common functions are described here.

- angleFuncs.**Eul2Quat**(theta *numpy.array*) - Converts euler angles (theta = [roll, pitch, yaw] $\in$ N x 3 array) to quaternions. Returns N x 4 array of the quaternion representation

[26] In Hamilton form: (w, x, y, z)
[27] U is the number of inputs to the system

in Hamilton (i.e. [w, x, y, z]) form.

- `angleFuncs`.**Quat2Eul**(quat *numpy.array*) - Converts quaternions (`quat = [w, x, y, z]` $\in$ N x 4 array) to euler angles. Returns N x 3 of the euler angles with `[roll, pitch, yaw]`.
- `angleFuncs`.**QuatRot**(q *numpy.array*, x *numpy.array*, rot = 'B2E' *string*) - Apply quaternion rotation, q, to vector, x. The order of rotations is specified through the `rot` parameter where 'B2E' indicates body-to-inertial (earth) and 'E2B' is the opposite.
- `angleFuncs`.**QuatMul**(Q1 *numpy.array*, Q2 *numpy.array*) - Multiply quaternion, `Q1`, with quaternion, `Q2`. Here, the rotation `Q2` is applied first and `Q1` is applied to the rotated intermediate frame.

Note that there are also JPL representations (i.e. [x, y, z, w]) available for quaternions for those that prefer it (see **Eul2Quat_JPL**(theta) and **Quat2Eul_JPL**(quat)). Regardless of the choice (Hamilton or JPL), it should be kept consistent throughout the simulation modules or converted where necessary. For example, **QuatRot** and **QuatMul** expect quaternions in the Hamilton form.

7.3.2 Drone visualization

Although closely related to the `droneviz` library, the `dronesim` animation module is actually independent of `droneviz`. The `droneSim` animation module is tailor made for the simulation environment. However, some useful functions from `droneViz` have been ported over as well.

The core of the animation module is the **animate.animation** class. This class handles the `matplotlib.pyplot.figure` objects as well as various animation properties such as the update frequency. An `animation` object may be initialized through **anim = animate.animation()**.

Before any animations can be visualized, the animation objects (called `actors`) need to be defined. These actors define how the animation object should be drawn; how the position and orientation information should be plotted in the figure. For convenience, a drone visualization class (`dronesim/animation/drone.py`) is made available to users of QuaSI. After a drone `actor` is created, it may be added to an `animate.animation` object. The animation is now ready to be deployed in the simulation, below is a code snippet of how this process works in practice:

```
6   from animation import animate, drone

241 # Define drone visualization object (actor)
242 droneViz = drone.body(model, origin=state[0, 0, 9:12], rpy=state[0,
        0, :3])
243 # Initialize animation
244 animator = animate.animation()
245 # Add drone to animation with same name as loaded model
246 animator.addActor(droneViz, model.modelID)
247
248 # Pass on all simulation objects to runSim.sim() to define the
        simulator.
249 simulator = sim.sim(model, EOM, controller, actuator, animator=
        animator, showAnimation=showAnimation)
```

Here, `model` is the (identified) `droneModel` object and `state` is the state vector. The initial position and orientation of the quadrotor should correspond to the initial conditions of the simulation object, otherwise the visualization will not render correctly. When added to an `animation` object, each `actor` needs to be assigned a name (in case there are multiple simulation objects). A practical choice is the ID corresponding to the quadrotor model.

The `drone.body` object supports some basic cosmetic configuration. Below are some of these functions (assuming the `drone.body` object is called `droneViz`, as in the code snippet above).

- `droneViz._`**setColor**(color *string*) - Sets both the rotor and body color of the quadrotor, default is black. Colors need to be compatible with `matplotlib.pyplot` colors.
- `droneViz._`**setFrameColor**(color *string*) - Set the color of the quadrotor frame, default is black. Colors need to be compatible with `matplotlib.pyplot` colors.
- `droneViz._`**setFrameThickness**(thickness *float*) - Sets the line thickness of the quadrotor frame. Default is 2.
- `droneViz._`**setHistoryColor**(color *string*) - Sets the color of the quadrotor position history line, default is turquoise blue. Colors need to be compatible with `matplotlib.pyplot` colors.
- `droneViz._`**setRotorColor**(color *string*) - Set the color of the quadrotor rotors, default is black. Colors need to be compatible with `matplotlib.pyplot` colors.

Note that these cosmetic properties of the drone can be modified at any point throughout the simulation (e.g. to represent changes in quadrotor state, such as rotor faults).

While the bulk of the animation is handled (and saved) by the `sim` object, there are various utility functions present to create additional animations or 3-D trajectory figures. These are summarized below (assuming `anim = animate.animation()`):

- `anim.`**animate**(time *numpy.array*, objectsPose *dict*, figure = None *matplotlib.pyplot.figure or None*, wrapper = None *function or None*, wrapperKwargs = {} *dict*, axisLims = {'x':None, 'y':None, 'z':None} *dict*) - Create animation of the simulation where `time` is the time vector and `objectsPose` is a dictionary of the actors names (e.g. `model.modelID`) to animate. Unlike other functions, here `objectsPose` is formatted in the following general way:

```
objectsPose = {
    model.modelID:{ # Actor name
        'time':simulator.time, # Time vector
        'position':simulator.state[:, 0, 9:12], # Position vector
        'rotation_q':simulator.simVars['quat'], # Quaternion orientation
        'inputs':simulator.inputs # Rotor speed information
    }
}
```

  This function permits the addition of the animation to an existing `figure`, wherein it will be added to the last active axes (this can be done explicitly by setting `anim.ax` to the desired index in `figure.axes`). Additional animations or data to plot can be completed in an (external) `wrapper` function of the form `wrapper(frames, objectPoses, wrapperKwargs)`[28]. Any arguments for the custom plots must be stored in the `wrapperKwargs` dict. `axisLims` allows users to specify (potentially time varying[29]) limits for each of the axes.

- `anim.`**asImage**(objectsPose *dict*, times *list*, parentFig = None *matplotlib.pyplot.figure or None*, figsize = (10, 10) *tuple*, uniformAxis = False *boolean*, encodeRotorSpeeds = True *boolean*, axisLims = {'x':None, 'y':None, 'z':None} *dict*) - Plot the trajectories of actors in `objectsPose` at times specified in `times` in an existing figure `parentFig` or as a standalone figure. Here, `objectsPose` is a dictionary of the actors names (e.g. `model.modelID`) to animate formatted in the following general way:

[28] The current index can be extracted through `idx = int(frames * wrapperKwargs["AnimationUpdateFactor"])`, note that "AnimationUpdateFactor" is added automatically to `wrapperKwargs`

[29] Modify `anim.axisLims` in the `wrapper` function to do so.

```
    objectsPose = {
        model.modelID:{ # Actor name
            'time':simulator.time, # Time vector
            'position':simulator.state[:, 0, 9:12], # Position
                vector
            'rotation_q':simulator.simVars['quat'], # Quaternion
                orientation
            'inputs':simulator.inputs # Rotor speed information
        }
    }
```

The boolean `encodeRotorSpeeds` specifies whether the thrust level of a given rotor should be encoded in its opacity. Set `uniformAxes = True` to ensure that all axes share the same limits, use `axisLims` to specify limits for each of the axes, or use default axis limits per axis.

- `anim.`**posteriorAnimation**(objectsPose *dict*) - Creates an (isolated[30]) animation of the simulation where `objectsPose` is a dictionary of the actors names (e.g. `model.modelID`) to animate, along with their corresponding `sim.simVars`.
  Example: `objectsPose = {model.modelID:simualtor.simVars}`.

- `anim.`**saveAnimation**(filename *string*, fps = None *int or None*, dpi = 300 int) - Saves the (most recent) animation to `filename`. Use `fps` to control the animation speed. Requires that ffmpeg is installed.

- `anim.`**snapShot**(objectsPose *dict*, times *list*, figsize = (10, 10) *tuple*, uniformAxis = False *boolean*) - Create a 3-D image of the simulation trajectory for each actor in `objectsPose` at the times specified in `times`. At each time, the actor object will be plotted along with its trajectory history. Set `uniformAxes = True` to enforce the same axis limits for x, y, and z-axes. `objectsPose` is a dictionary of the actors names (e.g. `model.modelID`) to animate, along with their corresponding `sim.simVars`.
  Example: `objectsPose = {model.modelID:simualtor.simVars}`.

Distinction between anim.animate and anim.liveAnimation:
The animation used during the simulation of the quadrotor at runtime is `anim.`**liveAnimation**, which has no relation to `anim.`**animate**. `anim.`**liveAnimation** is optimized to run much faster than `anim.`**animate**, but, as a consequence, does not support saving of animations. Thus, animations can only be saved by first calling the `anim.`**animate** function. `anim.`**liveAnimation** is purely for visualization of the simulation at runtime.

7.4 Example - quadrotor simulation

In this section, various examples are given to illustrate some of the functionality of the `dronesim` module. All of these examples are based on the `droneSimConfig.json` file shown in listing 12. Any changes to this base configuration will be highlighted where relevant in the subsequent examples. Note that these examples assume that the standalone model `MDL-HDBeetle-EXAMPLE` is available (and located in `dronesim/models`)[31]. If not, please see the (nominal) model identification (section 6.5) and standalone (section 6.6.1) examples to create the relevant model. Furthermore, those using `main.py` in the root directory should set `doSimulation = True` (all others modules may be set to `False` to only run the simulation) and modify `droneSimConfig.json` in the root directory. Those instead using the module scripts directly (i.e. `dronesim/droneSim.py`) should modify the local `dronesim/droneSimConfig.json` file.

[30] Unlike some other functions in the `animate.animation` class, the resultant animation cannot be added to other figures

[31] Of course, those that already have a simulation model may instead use that in place of `MDL-HDBeetle-EXAMPLE` in `droneSimConfig.json`

Another important note for those using `dronesim/droneSim.py` directly is that any identified (standalone/portable) models need to be copied over to the `dronesim/models` folder. To do so, navigate to the identified model directory and open the `standalone` (sometimes called `standaloneDiff`) subfolder. Copy all the files in the subfolder to `dronesim/models/<model ID>` where <*model ID*> is the model ID (i.e. `MDL-HDBeetle-EXAMPLE` in droneSimConfig.json). This <*model ID*> **must** match the start of the force and moment `.pkl` files from the `standalone` subfolder (e.g. in the example droneSimConfig.json, `.pkl` files start with `MDL-HDBeetle-EXAMPLE`).

This copying process is done automatically in `main.py`, so users of `main.py` do not need to worry about this copying.

Listing 12: Python simulation configuration parameters used in the simulation examples

```
{
    "model": {
        "use identified model": true,
        "model path": "models",
        "model ID": "MDL-HDBeetle-EXAMPLE",
        "correct for moment asymmetries": true
    },
    "noise": {
        "add": true,
        "seed": 737400
    },
    "controller": {
        "controller": "drone_NDI"
    },
    "actuator": {
        "actuator": "rotorDynamicsV2"
    },
    "simulation": {
        "time step": 0.004,
        "duration": 10
    },
    "references": {
        "do position track": false,
        "do circle track": true
    },
    "initial state conditions": {
        "attitude": [0, 0, 0],
        "position": [0, 0, 0]
    },
    "faults": {
        "inject failure": false,
        "time of failure": 2,
        "rotor health after failure": [0.5, 1, 0.75, 1]
    },
    "animation": {
        "show": true,
        "save": false
    }
}
```

7.4.1 Nominal simulation

Run the quadrotor simulation scripts[32] with the example `droneSimConfig.json`. At first, the simulation scripts will output various parameters to the terminal, such as:

```
[ INFO ] Found model-specific actuator time constant.  Using these over
default.
[ INFO ] SIMULATION PARAMETERS
        Controller:                 drone_NDI
        Add noise:                  True
        Drone model:                MDL-HDBeetle-EXAMPLE
        c.g.  offset correction:    True
        Actuators:                  rotorDynamics_V2
```

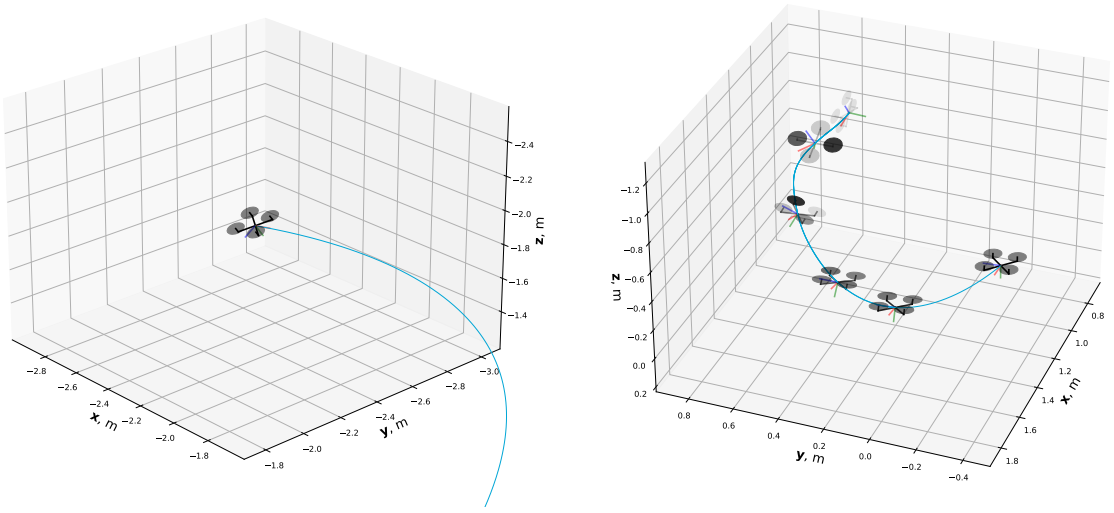
The first `[ INFO ]` message indicates that the chosen simulation model contains information on the rotor (motor) rate constants and it will use these values over the defaults for the actuator dynamics. The `[ INFO ] SIMULATION PARAMETERS` summarizes various parameters selected for the simulation, mainly shown as a verification check for users.

A visualization window should also pop-up depicting a 3-D view of the quadrotor. Initially hovering, at three seconds, the quadrotor begins to track a circular trajectory (as `"do circle track": True` under `"references"` in `droneSimConfig.json`). Figure 24a depicts a snapshot of this manoeuvre.

Through the `"initial state conditions"` parameters, the initial attitude and position of the quadrotor are specifiable. Make the following modifications to `droneSimConfig.json`:

```
26      "initial state conditions": {
27          "attitude": [0.5, -1, 0.5],
28          "position": [1, 0.5, -1]
29      },
```

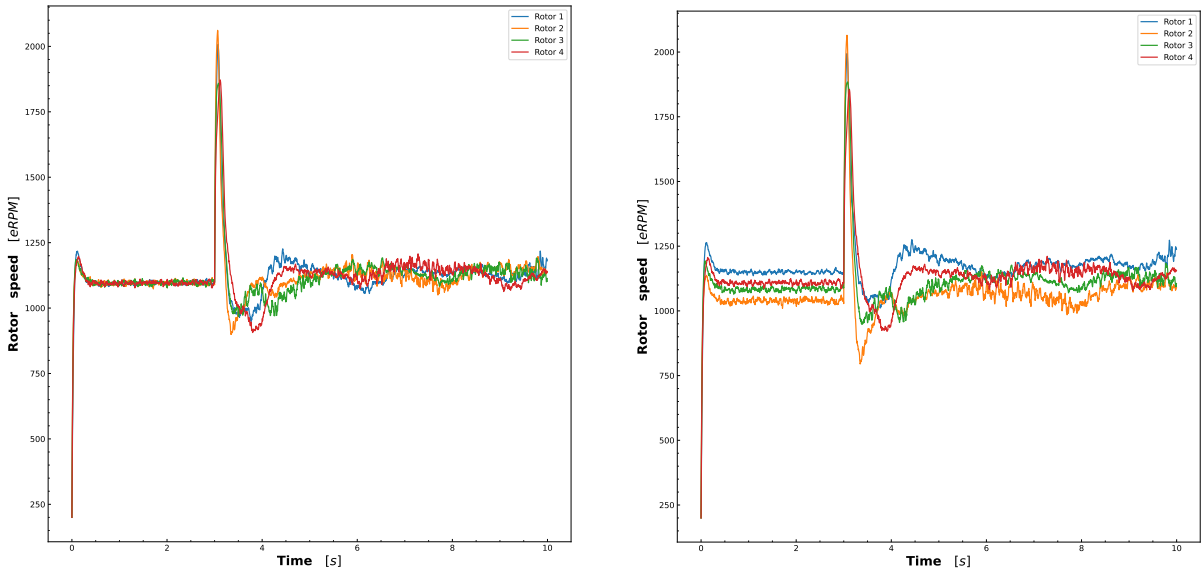
[32] `main.py` in the root directory or `dronesim/droneSim.py`



(a) Circular trajectory tracking

(b) Initial conditions response

Figure 24: Example animation snapshots of (nominal) `dronesim` simulations.



(a) `"correct for moment asymmetries":true` (b) `"correct for moment asymmetries":false`

Figure 25: Rotor speed plots from the nominal simulations.

which will initialize the quadrotor with an attitude of roll = 0.5, pitch = -1, and yaw = 0.5 radians at the position of x = 1, y = 0.5, and z = -1 meters. Note that, per the NED axis frame definition, the z-axis is positive downwards thus z = -1 corresponds to an altitude of +1 meters. Running the simulation scripts shows the quadrotor start at this initial condition and quickly return to the [0, 0, 0] conditions (as the references remain at zero until the circular trajectory starts at 3 seconds). Snapshots of this initial condition recovery response are shown in fig. 24b.

Similar to this initial condition 'disturbance' are the (typically) non-zero bias terms present in the identified moment models. Physically, these correspond to offsets between the center of gravity and center of rotation in the physical quadrotor. Moreover, any motor and propeller differences (e.g. in efficiency or degradation) also contribute to these moment biases. These biases are sometimes undesirable in simulations and thus the simulator can remove these when needed. This is done through the `"correct for moment asymmetries"` term under `"model"` in `droneSimConfig.json`. Set this value to `false` to observe the effect of these moment asymmetries:

```
6     "correct for moment asymmetries": false
```

Indeed, the quadrotor initially drifts away from the origin as the rotor speeds compensate for this 'initial' moment disturbance. Another interpretation is that the quadrotor is initially not correctly trimmed for hovering flight. Observe also the resultant rotor speed plots (depicted in fig. 25), for which there is now a clear difference in (hovering) rotor speeds when this moment bias correction is not applied.

Note that the rotor speed plots, along with other logged information (e.g. states, inputs, forces etc.), are saved to the `dronesim/simResults` sub-folder by the simulator. If users opt to save animations (i.e. `"save":true` under `"animation"` in `droneSimConfig.json`), then these will also be saved in this sub-folder. Note that ffmpeg is needed to save animations.

7.4.2 Rotor fault simulation

Partial rotor faults

For the partial rotor fault scenario, modify `droneSimConfig.json` with:

```
    "simulation":{
        "time step":0.004,
        "duration":6
    },
    "references":{
        "do position track":false,
        "do circle track":false
    },
```

```
    "faults":{
        "inject failure":true,
        "time of failure":2,
        "rotor health after failure":[0.5, 1, 0.75, 1]
    },
```

Run the simulation scripts and observe how, as soon as the fault occurs at two seconds, the quadrotor begins to drift from the origin. This is because one of the rotors is limited to below the minimum thrust required to maintain hover. Eventually, the controller loses control and the quadrotor spirals. This full trajectory is illustrated in fig. 26a. For some users, the simulation may crash before the full time has elapsed during the spiraling phase. This occurs because of numerical instabilities which arise from the rapidly changing dynamics. Depending on the exact model, this occurs faster or slower than others. Experiment with the time horizon (i.e. `"duration"` under `"simulation"` in `droneSimConfig.json`) to observe the full trajectory. The effect of the partial rotor faults in the resultant rotor speed plots of the simulation are visible through the blue (—) and green (—) lines in fig. 27a.

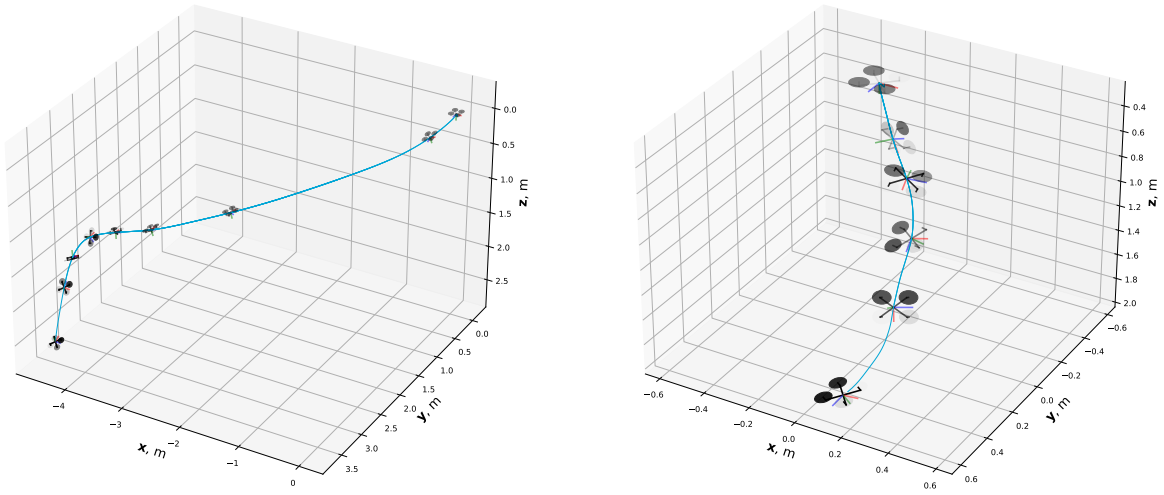
In this case, loss-of-control occurs because the controller is unaware of the fault and believes maintains nominal control effectiveness. As the tracking error grows, the controller is unable to reduce the error, becoming more aggressive as it grows, eventually leading to loss-of-control. It is important to highlight that the main reason the quadrotor spirals is due to the fact that the controller attempts to return to the origin while maintaining yaw control when it cannot. Forgiving some control authority/reference tracking (e.g. yaw axis) may facilitate recovery.

Full rotor faults

To demonstrate full rotor loss, change the following parameters in `droneSimConfig.json`:

```
    "simulation":{
        "time step":0.004,
        "duration":3
    },
    "references":{
        "do position track":false,
        "do circle track":false
    },
```

```
    "faults":{
        "inject failure":true,
        "time of failure":2,
        "rotor health after failure":[1, 1, 1, 0]
    },
```

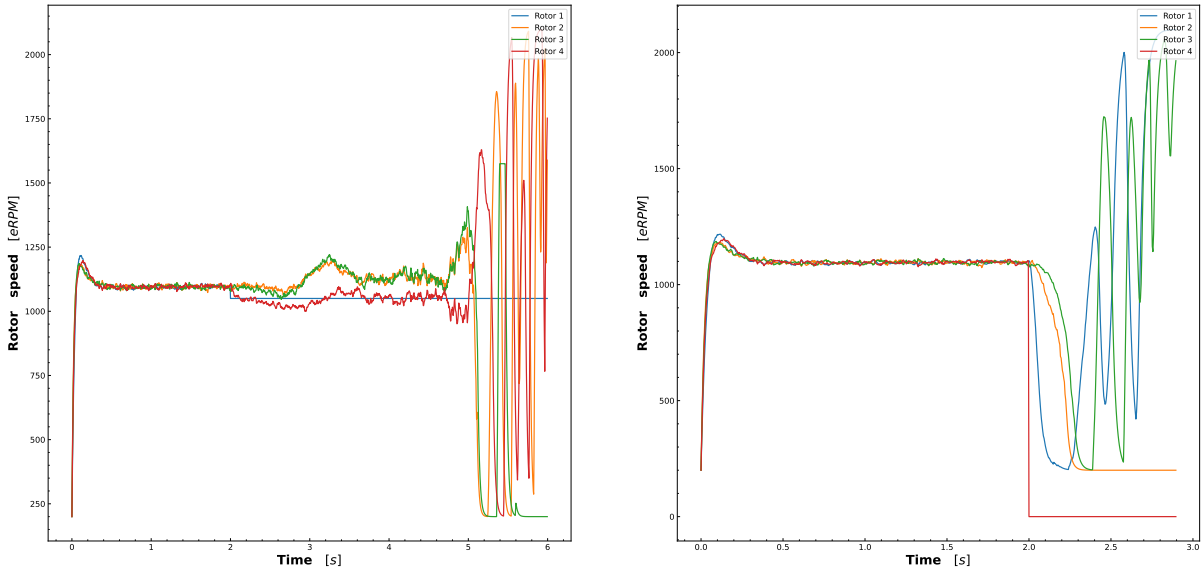


(a) Partial fault with [0.5, 1, 0.75, 1]

(b) Complete loss with [1, 1, 1, 0]

Figure 26: Snapshots of rotor fault `dronesim` simulations.

Run the simulation scripts and observe that rotor four "disappears" after the full rotor failure injection at two seconds. The quadrotor quickly loses control and spirals, as depicted in fig. 26b. Take a look at the resultant rotor speed plot, in fig. 27b, for which it is clear that the fourth rotor (—line) has zero (e)RPM after failure.



(a) Partial fault [0.5 (—), 1(—), 0.75 (—), 1 (—)]

(b) Full fault [1(—), 1(—), 1 (—), 0 (—)]

Figure 27: Rotor speed plots from the rotor fault simulations.

7.4.3 Custom simulation

General customization procedure

The simulator is designed such that modifications at any step of the main simulation loop (see fig. 23) can be made *without* having to modify the loop itself in `sim.run()`. In fact, this is precisely what is done to inject (partial) rotor faults for the examples in section 7.4.2. Take a look at the relevant code in `dronesim/droneSim.py`, shown below supplemented with additional comments for convenience:

Listing 13: Python code to inject (partial) rotor faults in droneSim

```
256 '''
257 Rotor failures
258 '''
259 # Modify actuator to simulate reduction in motor efficiency
260 rotorIndices = [0, 1, 2, 3]
261 rotorLimits = [f*model.droneParams['max RPM'] for f in simConfig['
       faults']['rotor health after failure']]
262 timeFailure = simConfig['faults']['time of failure']
263
264 # Limit the relevant rotor speeds
265 def capRotorsAt(trueInput, affectedRotors, limits):
266     # Get actual input and restrict relevant indices
267     cappedInput = trueInput.copy()
268     for i, rotorIdx in enumerate(affectedRotors):
269         cappedInput[:, rotorIdx] = np.nanmin([trueInput[:, rotorIdx
              ][0], limits[i]])
270     return cappedInput
271
272 # Define a "modified" actuator to cap rotor outputs between Actuator
       and Force and Moment Model in fig. 23
273 def modifiedActuator(simVars):
274     # Run "nominal" actuator
275     trueInput = simulator.actuator.actuate(simVars)
276     # If past failure injection time, limit rotor speeds
277     if simVars['currentTimeStep_index'] >= int(timeFailure/simVars['
          dt']):
278         modifiedInput = capRotorsAt(trueInput, rotorIndices,
              rotorLimits)
279     # Otherwise, keep nominal input
280     else:
281         modifiedInput = trueInput.copy()
282     return modifiedInput
283
284 if simConfig['faults']['inject failure']:
285     # Assign modified actuator to actuation step in simulator
286     simulator.doActuation = modifiedActuator
```

Notice that nothing fancy is going on here, all that is done is that the actuation step of the simulator is re-assigned to the `modifiedActuator`. The `modifiedActuator` function itself is essentially just filters the nominal actuator step (`simulator.actuator.actuate`) by limiting (or saturating) the relevant rotors before returning the final 'actuator' output. In this way, the controller is completely unaware of the modifications made to the actuator output.

Nonetheless, there is a specific syntax which needs to be followed in order to successfully modify the simulation steps:

First, the modified function (in this case `modifiedActuator`) takes only the `simVars` simulation data bus as input. To interface properly with the `sim.run()`, modified functions **must take** `simVars` as the the only input argument, whether it is used or not. While this may seem restrictive, recall that anything may be added to `simVars`, including custom signals, modules, functions etc. Hence, most of the necessary information can be made available through `simVars`. Note that, in listing 13, the nominal actuator input is obtained through `simulator.actuator.actuate` where `simulator` is defined globally. As the actuator module is directly available in `simVars`, this is equivalent to calling `simVars['actuator'].actuate`. The output of the `modifiedActuator` should be of the same shape and type as the nominal actuator output.

Second, the modified steps need to be assigned to the appropriate attribute in the `sim` object. In the example of listing 13, the `modifiedActuator` is assigned to the `sim.doActuation` attribute. Below are the attributes in the `sim` object available for re-assignment:

- `sim.doControl` - Step corresponding to the controller, returns the commanded input.
- `sim.doActuation` - Step corresponding to the actuator dynamics, returns the true input.
- `sim.doForcesMoments` - Step corresponding to the force and moment models, returns the forces and moments
- `sim.doEOM` - Step corresponding to the equations of motion, returns the state derivative

Note that it is possible to modify multiple steps in custom simulation architectures.

Another way to view these modifications is that the customizations are made by 'hooking on to' step(s) of the nominal simulation (to modify them) as necessary. As an exercise to experiment with this customization architecture, consider how the simulation may be modified to add, potentially time-varying, lateral wind.

Custom simulation - Adding keyboard control

Following this same logic of 'hooking on to' a main loop step, this example modifies the simulation to accept user inputs to control the quadrotor directly. This is achieved by reassigning the `sim.doControl` method to a modified variant which adjusts the reference signal with the user keyboard inputs before passing it onto the (nominal) controller module. The relevant code can be found in listing 14 (for convenience, users can copy-and-paste or download from here). The code should be placed just before the `'''MAIN SIMULATION LOOP'''` in `dronesim/droneSim.py`.

Listing 14: Python code to control the quadrotor using the keyboard in droneSim. Code can be copied or downloaded from here.

```
'''
Make quadrotor position controllable through the keyboard.
'''
# First clear any references:
simulator.reference = state.copy()
simulator.simVars['reference'] = simulator.reference

from funcs import keyboard_utils
# Create get character object
userinput = keyboard_utils._getch_wasd()
# Start thread (i.e. listening)
userinput.start()
# Add get character thread to simulator
simulator.addSimVar('userinput', userinput)

# Here, we define our modified controller
def modifiedController(simVars):
    # Get current input step
```

```
    k = simulator.simVars['currentTimeStep_index']
    delta = 0.5

    userinput = simVars['userinput'].getKey()

    # Control quadrotor position through w a s d
    if userinput == 'w': # Forward
        simVars['reference'][k+1:, :, 10] = simVars['reference'][k,
            :, 10] - delta
    elif userinput == 's': # Backward
        simVars['reference'][k+1:, :, 10] = simVars['reference'][k,
            :, 10] + delta
    elif userinput == 'a': # Left
        simVars['reference'][k+1:, :, 9] = simVars['reference'][k, :,
            9] - delta
    elif userinput == 'd': # Right
        simVars['reference'][k+1:, :, 9] = simVars['reference'][k, :,
            9] + delta

    # Stop reading user input near end of simulation
    if k == len(simulator.simVars['time']) - 10:
        simVars['userinput'].doKill()

    # Do nominal control with modified reference
    return simVars['controller'].control(simVars)

# Assign modified controller to simulator control method.
simulator.doControl = modifiedController
```

For convenience, this example makes use of the `dronesim/keyboard_utils` module, which handles the key captures. Make the following modifications to the example `droneSimConfig.json` file[33]:

```
    "references":{
        "do position track":false,
        "do circle track":false
    },
```

Now run the `dronesim/droneSim.py` script. Note that, to give keyboard commands to the simulator, the command terminal needs to be the active window (not the `matplotlib` visualization). Experiment with the "w", "a", "s", and "d" keyboard inputs and observe the behaviour of the quadrotor. Note that too aggressive commands may result in loss-of-control.

Custom simulation - Endless keyboard control

Up until now, defined time horizons have been specified for the simulator. While this is convenient for repeated and defined simulations, it may be also be useful for some users to run simulations without pre-defined end times. In the `dronesim` module this can be configured through the simulator's `run_mode`, which is specified upon initialization of the `sim` object. By default, `run_mode = 'finite'`, which indicates that there is a defined end time for the simulation. Passing `run_mode = 'infinite'` allows the simulation to run indefinitely. For example, in `droneSim.py`:

```
simulator = sim.sim(model, EOM, controller, actuator, animator=
    animator, showAnimation=showAnimation, run_mode='infinite')
```

[33] Users need to modify the `dronesim/droneSimConfig.json` file.

Set the simulator `run_mode = 'infinite'` during the `sim` object initialization (as is done in the code snippet above). Run the user keyboard input simulation of the previous example again. Observe that the progress bar no longer appears in the terminal, as before, interact with the simulation through the "w", "a", "s", and "d" keys. To exit the simulation, simply close the animation window.

Note that, when `run_mode = 'infinite'`, a termination condition **must** be set. If an animator is passed to the simulator, then the default termination condition is tied to the closure of the animation window (i.e. simulation stops when the animation stops, as seen in the previous example). However, if an animator is not passed, then users must specify a termination condition (otherwise, an error will be thrown when `sim.run` is called). This is done through the `simulator.setTerminationCondition` method. Users must pass a callable function which returns `True` or `False`, where `False` indicates that the simulation should be stopped. For flexibility, a dictionary of keyword arguments may be passed to `simulator.setTerminationCondition`, which will then be supplied to the user-defined termination function. An example of how such a custom termination condition is given below:

```
def terminationFunc(simVars={}):
    k = simVars['currentTimeStep_index']
    return not simVars['state'][k, 0, 9] > 10

#Syntax: simulator.setTerminationCondition(func, funcKwargs = {})
simulator.setTerminationCondition(terminationFunc, funcKwargs={'
    simVars':simulator.simVars})
```

Here, the a custom termination condition is created wherein the simulation will stop once the x-position of the quadrotor exceeds 10 meters. In order to assess this, the current state of the quadrotor needs to be known. This is accessible through the `sim.simVars` data bus, and is therefore used as a keyword argument. At each simulation step, the simulator supplies the `terminationFunc` with `funcKwargs['simVars']`, which it uses to evaluate the position condition.

7.5 Example - Constructing arbitrary simulation

This example covers how a simulation environment may be constructed for arbitrary (potentially non-quadrotor related) simulations. The `buildMySim.py` script creates templates of the core modules of a simulation environment and manages the basic interfacing between them. It is up to the users to complete these pre-filled templates to create their desired simulation environment. In this example, a mass-spring-damper system is used. First, the framework of a simulation environment can be created through the `dronesim/buildMySim.py` file. In the `dronesim` directory, run

```
python buildMySim.py
```

The following prompt asking for a simulation environment name should appear

```
Please indicate a name for the SIMULATION (e.g. mySim):
```

In this example, the name `massSpringDamper` is used. Subsequently, the script will ask if default names should be used for the simulation modules:

```
Would you like to use default names for the simulation components? (y/n)
```

If users type `n`, then they will be asked to provide names for *each* of the simulation modules (controllers, actuators, etc.). Typing `y` will prepend the chosen simulation environment name, here `massSpringDamper`, with the relevant module (e.g. `massSpringDamper_controller` for the controller). This option is chosen in the present example. Subsequently, the script provides a summary of the created files, their directories, and where the main simulation file (here called `massSpringDamper.py`) can be found. Various files should have been created for the different simulator modules:

- `massSpringDamper_actuator.py` under the `dronesim/actuator` subfolder
- `massSpringDamper_controller.py` under the `dronesim/controller` subfolder
- `massSpringDamper_EOM.py` under the `dronesim/funcs` subfolder
- `massSpringDamper_Model.py` under the `dronesim/models` subfolder
- `massSpringDamper.py` under the root `dronesim` folder

Specifiying model - massSpringDamper_model.py:

Open the `massSpringDamper.py` file and observe within the first six lines that the relevant simulation modules are already imported into the script. The first module that is initialized is the mass-spring-damper model, on lines 10-11:

```
10 # Load and initialize model. Add (keyword) arguments as necessary
11 model = massSpringDamper_model.massSpringDamper_model()
```

Indeed, the `buildMySim.py` file has no knowledge of what a mass spring damper system is, and thus, the actual dynamics need to be implemented by the user. To this end, open the `models/massSpringDamper_model.py` file. In the initialization method (i.e. `__init__()`), only the name of the module is present. Typically, the system properties are also specified in this step. For the mass-spring-damper system, these are the mass (m), damping constant (c), and spring constant (k). Below, the `__init__()` method is extended to accept these as (keyword) arguments and stores them as model attributes.

```
5  class massSpringDamper_model:
6      def __init__(self, m = 1, c = 1.5, k = 0.8):
7          self.name = "massSpringDamper_model"
8          self.m = m
9          self.c = c
10         self.k = k
11         return None
```

The core of the model object is the `getForcesMoments(self, simVars)` function, which takes the simulation data bus as input. Note that a few (interfacing) steps are already pre-filled by the `python buildMySim.py` script. These are the current simulator step, which is retrieved from the data bus through `step = simVars["currentTimeStep_index"]`, along with the current state and most recent (true) input. All that is left is to actually implement the force and moment dynamics. Assuming zero moments, the mass-spring-damper system may be represented by

$$\begin{aligned} F_x &= m\ddot{x} \\ &= m\left(u - 2\zeta\omega_n\dot{x} - \omega_n^2 x\right) \end{aligned}$$

where F_x is the force along x, u is the true input to the system, $\zeta = \frac{c}{2m\omega_n}$ is the damping factor and $\omega_n = \sqrt{\frac{k}{m}}$ the natural frequency of the system. Let $x_1 = x$ and $x_2 = \dot{x}$. This model may be implemented by (with `import numpy as np` specified at the top of the file):

```
16     def getForcesMoments(self, simVars):
17         # Extract the current step and state
18         step = simVars["currentTimeStep_index"]
19         state = simVars["state"][step]
20         # Since the control action has just been updated, take the
               next step
21         trueInput = simVars["inputs"][step + 1]
22         # Calculate forces and moments here. Create and call relevant
               functions if necessary
```

```
23          # vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv Added dynamics
24          wn = np.sqrt(self.k/self.m)
25          damping = self.c/(2*self.m*wn)
26          Fx = np.array(trueInput - self.m*(2*damping*wn*state[:, 0] +
                wn**2 * state[:, 1])).reshape(-1)
27          F = np.hstack([Fx, 0, 0]) # Only force along x, but sim
                expects also y and z forces
28          M = np.zeros((1, 3)) # No moments
29          # ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Added dynamics
30          return F, M
```

Specifying initial conditions - massSpringDamper.py

The initial conditions are specified on lines 13-18 in `massSpringDamper.py`. These need to be modified to fit our current system of two states. Make the following changes to the initial conditions:

```
13 # Initial condition and simulation parameters
14 dt = 0.01 # Time step
15 time = np.arange(0, 10, dt) # Create time array with time step dt
16 state = np.zeros((len(time), 2)).reshape(-1, 1, 2) # Define state vector
17 action = np.zeros(time.shape).reshape(-1, 1, 1) # Define action (
       input) vector
18 reference = np.zeros(time.shape).reshape(-1, 1, 1) # Define reference
       , here we only control the position (x1)
```

Note that the time step, `dt`, has been reduced from 0.1 to 0.01. This is because the controller actually becomes unstable with the larger time step in the presence of actuator dynamics (i.e. the delay induced by the actuator dynamics causes the controller to become unstable when an action can only be made every 0.1 seconds).

Specifying controller - massSpringDamper_controller.py

In the simulation architecture, (sensor) noise is added in the controller step seeing as this is what the controller 'observes' of the state. To this end, the simulation scripts support the addition of Gaussian white noise to each of the states. The noise magnitude is specified per state. Take a look at lines 20-35 of `massSpringDamper.py`, which outline how the noise statistics are specified and how the noise 'block' is created. This `noiseBlock` is required to initialize the controller, even for noise-free simulations.

Modify the noise statistics as follows:

```
20 # Initialize noise block
21 addNoise = True # Boolean to toggle noise in system
22 stateMapper = {"x1":0, "x2":1} # Map states to their indices in state
       vector
23 noiseBlock = sensorNoise.noiseBlock(stateMapper)
24 # Noise seed
25 noiseSeed = 123456
26 # Define noise levels for each state
27 noiseMapping = {
28     'x1':0.1,
29     'x2':0.01}
30 if not addNoise:
31          # To turn noise off, set mapping of added noise to 0 for each
               state
```

```
        noiseBlock.mapStateNoiseScales(np.array(list(stateMapper.
            values())), np.array(list(stateMapper.values()))*0)
else:
    for k, v in noiseMapping.items():
        noiseBlock._setStateNoiseScaling(k, v)
```

Now, it is time to create the controller. Here, use of the built-in `PID` class is made. Open the `controllers/massSpringDamper_controller.py` file. First, at the top of the script, import the `PID` class and `numpy` through

```
from controllers import PID
import numpy as np
```

Next, in the controller `__init__()` method, allow for the specification of the P, I, and D gains and define the PID controller as below[34]

```
class massSpringDamper_controller:
    def __init__(self, P = 1, I = 0, D = 0, noiseBlock = None):
        self.PID = PID.PID(np.array(P).reshape(1, 1), np.array(I).
            reshape(1, 1), np.array(D).reshape(1, 1))
        self.name = "massSpringDamper_controller"
        if noiseBlock is None:
            raise ValueError("Need to specify noiseBlock")
        self.noiseBlock = noiseBlock
        return None
```

Complete the `control` method by passing the tracking error and time step to the `PID.control` method:

```
    def control(self, simVars):
        # Extract current step and state at this step
        step = simVars["currentTimeStep_index"]
        # Use noiseBlock to simulate sensor noise (only perceived by
            controller, hence state_noisy)
        self.noiseBlock.addStateNoise(simVars) # Updates simVars["
            state_noisy"] with state[step] + noise
        state = simVars["state_noisy"][step]
        # Do control here. Create and call relevant functions as
            necessary
        # vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv Added control action
        dt = simVars['dt'] # Needed by PID
        reference = simVars['reference'][step]
        controlInput = self.PID.control(reference[:, 0] - state[:,
            0], dt)
        # ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Added control action
        return controlInput
```

Return to `massSpringDamper.py` and adjust the initialization of the controller to be compatible with the changes made:

```
controller = massSpringDamper_controller.massSpringDamper_controller(
    P = 3, I = 8, D = 2, noiseBlock = noiseBlock)
```

Specifying actuator dynamics - massSpringDamper_actuator.py

[34] Note that the built-in PID class expects the gains to be given as a 2-D `numpy.array` object. The rows correspond to the "input" whereas the the columns are the "output" of the PID controller. For example, a rate to rotor-speed PID will take 3 inputs (roll, pitch, and yaw rates) and output 4 rotor speeds (for a quadrotor).

In this example, the actuator is assumed to have first order dynamics without saturation. For this, rate constants needs to be specified. In `actuators/massSpringDamper_actuator.py` modify the `__init__()` method to allow for this, and store it as an attribute:

```
5  class massSpringDamper_actuator:
6      def __init__(self, rateConstant = 1/10):
7          self.tau = rateConstant
8          self.name = "massSpringDamper_actuator"
9          return None
```

Before the actuator dynamics can be properly implemented, the initial actuator action needs to be specified. To this end, create a new function in the `massSpringDamper_actuator` class called `setInitial` as below

```
11     def setInitial(self, intialAction):
12         self.previousCommand = intialAction
```

Now, the actuator dynamics can be implemented in the `actuate` method:

```
16     def actuate(self, simVars):
17         step = simVars["currentTimeStep_index"]
18         commandedInput = simVars["inputs"][step + 1]
19         # Place actuator dynamics here. Create and call relevant
               functions if necessary
20         # vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv Added actuator dynamics
21         dt = simVars["dt"]
22         trueInput = self.previousCommand + dt*(1/self.tau)*(
               commandedInput - self.previousCommand)
23         self.previousCommand = trueInput
24         # ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Added actuator dynamics
25         return trueInput
```

Again, the initialization of the actuator module needs to be modified in `massSpringDamper.py` following the modifications made:

```
32 actuator = massSpringDamper_actuator.massSpringDamper_actuator(
       rateConstant=1/15)
33 actuator.setInitial(action[0])
```

Specifying dynamics - massSpringDamper_EOM.py

Before constructing the equations of motion, it is perhaps beneficial to re-write the system in state-space form:

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ -k/m & -c/m \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{bmatrix} 0 \\ 1/m \end{bmatrix} u$$

Open the `funcs/massSpringDamper_EOM.py` file and modify the `EOM` function with these dynamics.

```
4  import numpy as np
5  def EOM(simVars):
6      # Extract current step and state at this step
7      step = simVars["currentTimeStep_index"]
8      state = simVars["state"][step]
9      # Since the forces and moments have just been updated in the
           simulation loop, we need to extract the forces and moments of
           the next step
10     forces = simVars["forces"][step + 1]
11     moments = simVars["moments"][step + 1]
```

```
    # vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv Added state equation
    model = simVars['model']
    A = np.matrix([[0, 1], [-model.k/model.m, -model.c/model.m]])
    B = np.matrix([[0], [1/model.m]])
    x_dot = np.matmul(A, state.T) + np.matmul(B, forces[:, 0].reshape
        (1, 1))
    # ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Added state equation
    return x_dot.T
```

Running simulator - massSpringDamper.py

Now the simulation environment is effectively set up. To observe the tracking performance of the simple controller, a (non-zero) reference should be supplied for tracking. In `massSpringDamper.py`, before the `sim.setInitial` method, define a step reference tracking input. For example:

```
# Set up step reference
reference[int(1/dt):] = 1

# Construct simulator
# NOTE: If you would like to use quaternions, then specify the [roll,
    pitch, yaw] euler angle indices in the state vector under the
    keyword argument "angleIndices=[roll_index, pitch_index, yaw_index
    ]"
simulator = sim.sim(model, EOM, controller, actuator, angleIndices =
    [])
simulator.setInitial(time, state, action, reference) # Set initial
    conditions
```

Moreover, modify the plotting steps to also show the reference:

```
# Plot some results
fig = plt.figure()
ax = fig.add_subplot()
ax.plot(simulator.time, state[:, 0, 0], color = 'tab:blue', label = '
    Position')
ax.plot(simulator.time, state[:, 0, 1], color = 'tab:orange', label =
    'Velocity')
ax.plot(simulator.time, reference[:, 0, 0], color = 'tab:blue', label
    = 'Position Reference', linestyle = '--')
ax.set_xlabel("Time, s")
ax.set_ylabel("State(s)")
plt.legend()
plt.show()
```

Run the `massSpringDamper.py` script to simulate the controlled system. A plot similar to fig. 28 should appear once the simulation completes. Indeed, the controller is able to force the system position to the desired reference. Experiment with the controller tuning to improve the tracking performance.

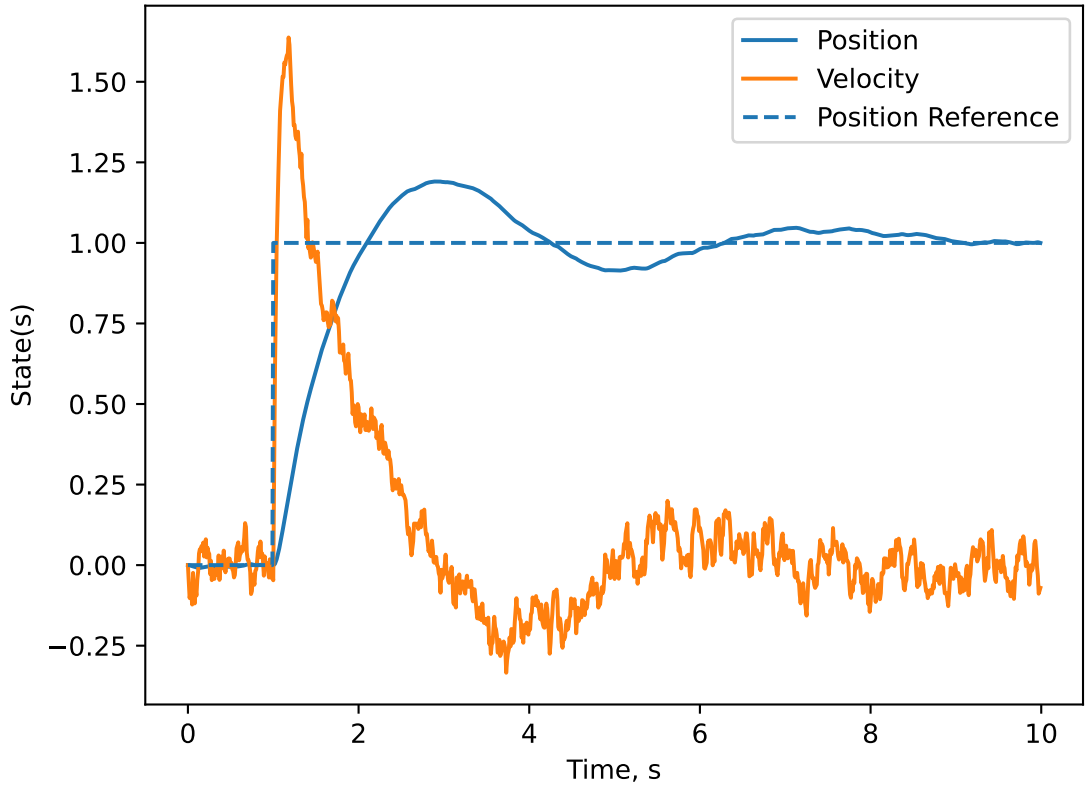


Figure 28: Position (—) tracking performance of the controller mass spring damper system. Also shown is the velocity (—).

8 MATLAB Port

A MATLAB (R2021b) port of the identified models and simulation environment is made available by QuaSI. This allows users to run the identification pipeline to create models in Python, but deploy them in MATLAB for further analysis or use. This also facilitates the use of various MATLAB utilities that do not (yet) exist in Python, such as many functionalities of the robust control toolbox. Moreover, using Simulink to C++ or Px4 toolboxes, identified models and subsequent controllers may be easily flashed to hardware.

After the model identification is completed, users can port the identified models to MATLAB through the `toMATLAB.py` script. The script parameters (e.g. model ID to port) are configurable through the `MATLABConfig.json` file, summarized in section 8.1. Usage of these MATLAB models is summarized in section 8.2 while section 8.3 details their deployment in the MATLAB simulation environment.

8.1 Configuration files - MATLABConfig.json

Reminiscent of the `standaloneConfig.json`, the `MATLABConfig.json` has mostly the same configuration parameters. This is because the `toMATLAB.py` calls the standalone scripts in order to generate a portable model from which it creates the MATLAB model. Note that this file is called `modelConfig.json` in the `droneidentification/matlab_compatibility` folder, they are actually the same file. `main.py` overrides the contents of `modelConfig.json` with the contents of `MATLABConfig.json` in the root `QuaSI` repository.

Listing 15: Configurable entries of the MATLABConfig.json file with comments for clarity preceeded by %

```
{
    % Directory in which the model folder is located
    "model path": "models/HDBeetle",
    % Name of the model (folder)
    "model ID": "MDL-HDBeetle-EXAMPLE",
    % Directory of the quadrotor configuration file (section 4.2)
    "droneConfig path": "data",
    % Name of the quadrotor configuration file
    "droneConfig name": "HDBeetleConfig.json",
    % Whether a (moment model) differential form of the polynomial
       model should be created and added to the standalone model.
       Currently, only differential forms of the moment model are
       supported.
    "make moment DiffSys": true,
    % The differential form input vector
    "DiffSys u": ["U_p","U_q","U_r"]
}
```

8.2 Quadrotor Model Port

Within the `droneidentification` repository, there is a subfolder called `matlab_compatibility`, which hosts all of the MATLAB (R2021b) files.

Identified quadrotor polynomial models can be transferred to MATLAB (R2021b) through the `toMATLAB.py` script in the `matlab_compatibility` folder. This script first creates a portable version of the identified models (mainly, to make the differential form model also available in MATLAB), from which it writes the relevant data to MATLAB (`.m` and `.mat`) files. Ported models are saved to the `matlab_compatibility/models` subfolder, with the same `"model ID"` used during model identification.

Each model folder contains various `.m` scripts and `.mat` files created by `toMATLAB.py`. The `droneParams.mat` file hosts information pertaining to the properties of the quadrotor (i.e. information contained in the Quadrotor Configuration File along with additional information derived in the identification process, such as the rotor rate constants). Similarly, the `modelParams.mat` file harbors details on the model identification parameters, such as whether the model is normalized.

The `.m` scripts are used to obtain the aerodynamic model forces and moments from the state and input vectors. For example, `get_Fx.m` computes the force along the quadrotor x-axis. These scripts are all dynamically created from the identified models, and may therefore have a different number of input arguments per model. Thus, to standardize the force and moment computations, users can simply call the `getFM.m` file, which takes the state and input vector (along with data from `droneParams.mat` and `modelParams.mat`) and returns a tuple of the force and moment vectors. Internally, the `getFM.m` calls upon the `getDroneInputs.m` file in the `matlab_compatibility/models` subfolder. This script translates the input arguments of `getFM.m` to the polynomial regressors.

In fact, many of the useful utility scripts from the Python identification pipeline have counterparts in this `matlab_compatibility/models` subfolder. For instance, some of the quaternion utilities (see section 7.3.1) are also ported for use in the MATLAB simulation, such as `Eul2Quat.m`. Note that, users that make modifications to the sister scripts in the Python identification pipeline need to also mirror those changes in the relevant `.m` file. For example, changes made to `processing/normalization.py` need to also be made in the normalization steps of the `getDroneInputs.m` file.

The `getDroneInputs.m` file is also needed for those that want to make use of the differential form moment model. This model, if identified in the Python pipeline, can be found in the `get_PQRDiffSys.m` file in the ported model folder. As the only input argument, this script takes the output of `getDroneInputs.m` and returns the differential form A and B matrices at a given state and input.

8.2.1 Identified Model Port Example

This example demonstrates how a MATLAB port of an identified, Python-based, quadrotor polynomial model can be created. The following `MATLABConfig.json` file is used:

Listing 16: MATLAB model port configuration parameters used in the example

```
{
    "model path": "models/HDBeetle",
    "model ID": "MDL-HDBeetle-EXAMPLE",
    "droneConfig path": "data",
    "droneConfig name": "HDBeetleConfig.json",
    "make moment DiffSys": true,
    "DiffSys u": ["U_p","U_q","U_r"]
}
```

Users of the `main.py` file in the root directory of `QuaSI` should ensure that `doMATLAB = True` in `main.py`. All other flags may be set to `False`. Note, however, that this example assumes that the model `"MDL-HDBeetle-EXAMPLE"` has been fully identified (i.e. all forces and moments have been identified). Of course, this `"model ID"` may be replaced by other identified models[35] if users prefer; this example will continue with `"MDL-HDBeetle-EXAMPLE"`. If no model has been identified, consult the model identification examples.

Alternatively, users of the `toMATLAB.py` file directly need to modify the `modelConfig.json` file in the `matlab_compatibility` folder with the contents of `MATLABConfig.json`. Care should be taken to ensure that the relative paths point to the correct directories. That is, both `"model`

[35] Modify the paths as necessary

path" and "droneConfig path" should be be pre-pended with "../../" in the standard folder layout of QuaSI. This is because the matlab_compatibility is located two directories deeper than the data folder whereas main.py is in the same directory.

Run the MATLAB port scripts (i.e. main.py or toMATLAB.py). Upon a successful model port, the following output (or similar) should be shown in the terminal after the standard standalone script terminal outputs

```
[ INFO ] Converting model...
[ INFO ] Model conversion successful!
[ INFO ] Option make moment DiffSys selected.
[ INFO ] DiffSys conversion successful!
```

To verify this, navigate to the droneidentification/matlab_compatibility/models folder. There should be a folder called MDL-HDBeetle-EXAMPLE, which contains the .m and .mat files described in section 8.2.

8.3 MATLAB Simulation Environment

The main.m script, located in the matlab_compatibility folder, demonstrates how ported models can be deployed in MATLAB. In summary, the relevant ported model folder should be added to the MATLAB path and the model needs to be initialized through [modelParams, droneParams] = model_init(modelID). See the example in section 8.3.1 for more details on how to achieve this.

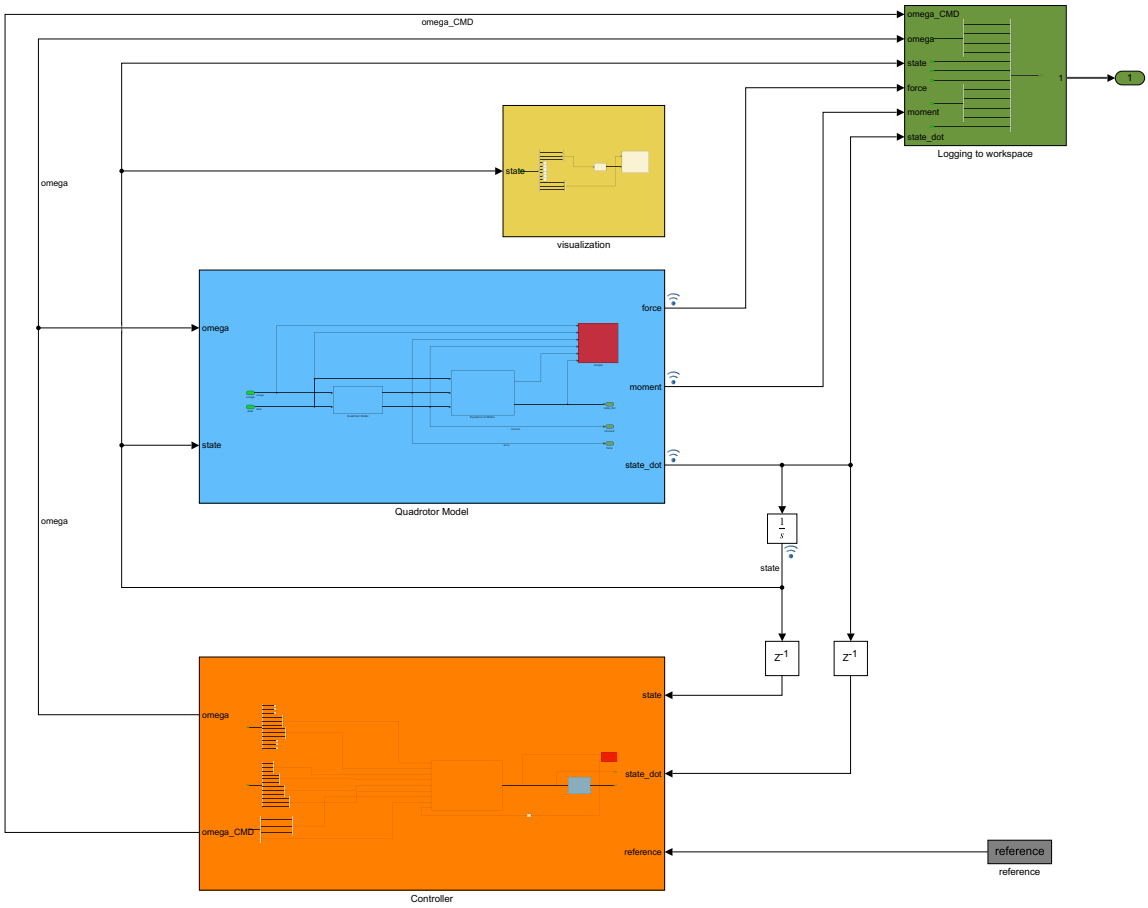


Figure 29: Schematic block diagram of the MATLAB simulation environment of simpleModelSim.slx

The heart of the MATLAB simulation environment is the `simpleModelSim.slx` Simulink file, the block diagram of which is depicted in fig. 29. Unfortunately, the MATLAB simulation environment is not as extensive as the Python variant. Currently, only simulations with pre-defined end times can be run, unlike the indefinite simulations of the Python Simulation environment. Moreover, only an INDI controller is implemented for users (see orange block in fig. 29). Nonetheless, users can replace the default controller with their own, or modify the inner-loops of the current INDI implementation, shown in fig. 30.

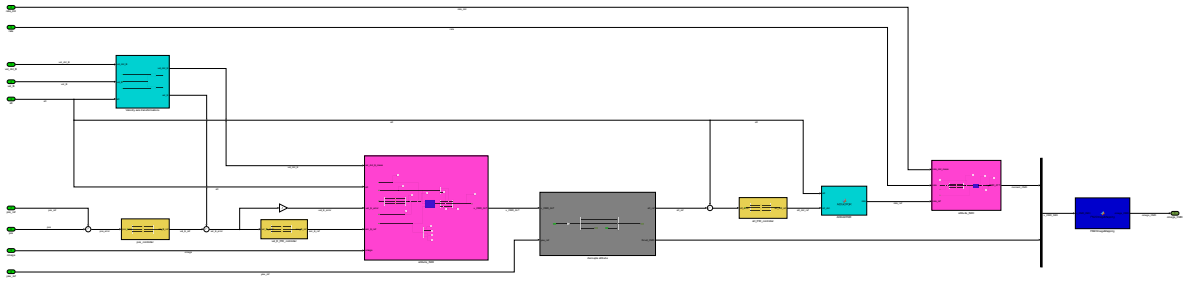


Figure 30: Schematic block diagram of the implemented controller in `simpleModelSim.slx`. Here, yellow blocks denote PID controllers, pink blocks are INDI controllers, turquoise blocks denote axis transformations, and dark blue blocks host dynamics.

Those that instead prefer to only use the model block (blue block in fig. 29) may be interested in the `baseModel.slx` Simulink file, which just contains the model itself. Alternatively, a copy of the "Quadrotor Model" block in fig. 29 (illustrated in fig. 31) can be made.

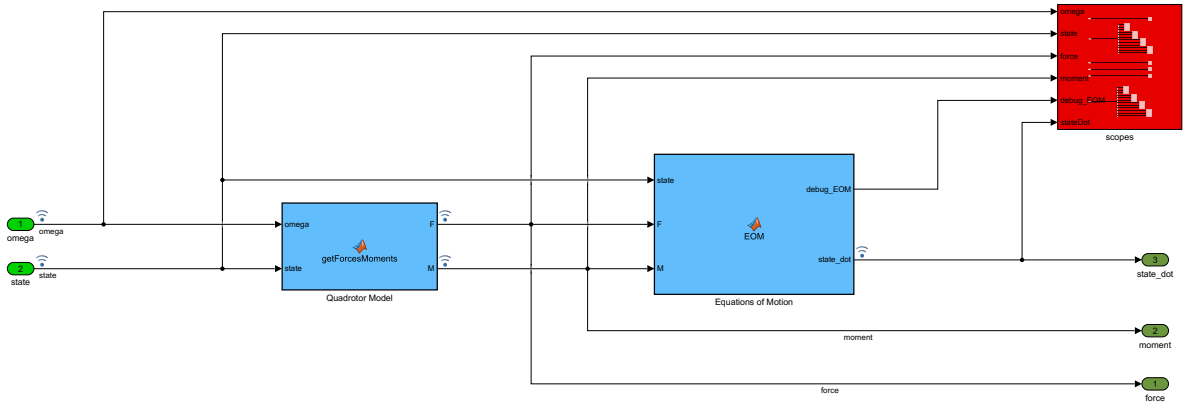


Figure 31: Schematic block diagram of the Quadrotor Model block in `simpleModelSim.slx`

The `simpleModelSim.slx` simulation environment also sports a visualization of the quadrotor, using an off-the-shelf UAV visualization toolbox. The visualization blocks are all contained within the yellow block in fig. 29.

Finally, the `simpleModelSim.slx` outputs all logged variables (see green block in fig. 29) to the workspace for further analysis. Additional logged variables can be added as necessary.

Currently, the state vector, derivative thereof, commanded input vector, true input vector and forces and moments are logged. To aid with debugging, the `simpleModelSim.slx` scopes these variables throughout the block diagram. These scopes are all collected in the red blocks.

8.3.1 MATLAB Simulation Example

Indeed, the identified models may be used directly in users' own MATLAB pipelines. In this example, the environment constructed in the `droneidentification/matlab_compatibility` folder is used to demonstrate how the models can be accessed and implemented.

In the `matlab_compatibility` folder, open the `main.m` file. Here, the model to load is defined through `modelID`. To make it visible to the MATLAB scripts (in particular, `simpleModelSim.slx`), the relevant model folder is added to the MATLAB path[36]. Subsequently, the model is initialized. Continuing with the example of section 8.2.1, modify `main.m` with

```
% Clear all and reset to default paths, we add necessary paths upon
    init
close all; clear all; path(pathdef); clc;

% Define underlying model and add necessary paths
modelID = 'MDL-HDBeetle-EXAMPLE';
addpath("models");
addpath(fullfile("models", modelID));

% Initialize model
[modelParams, droneParams] = model_init(modelID);
```

This is the minimum necessary to properly load the identified models. From here, use `getFM(state, inputs, modelParams, droneParams)` to get the forces and moments. In fact, this is what is done in the `Quadrotor Model` block of `simpleModelSim.slx`.

Continuing with the example, the next step needed to construct the simulation is to set up the simulation parameters and reference signal. To this end, in the `main.m` script, make the following changes

```
%% Simulation (simpleModelSim.slx)
% Time
Tmax = 15;
dt = 0.004;
time = (0:dt:Tmax);

% Define initial values
% State = [attitude, velocity, rates, position]
%   - attitude = [roll, pitch, yaw] in rad
%   - velocity = [u, v, w] in m/s along body x, y, z respectively
%   - rates = [p, q, r] in rad/s about body x, y, z respectively
%   - position = [x, y, z] in m in the earth-frame! (All other vars
    in body frame)
% Omega = [w1, w2, w3, w4], rotor speeds in eRPM
initialState = [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]';
initialOmega = [droneParams.minRPM, droneParams.minRPM, ...
                droneParams.minRPM, droneParams.minRPM]';
```

[36] `simpleModelSim.slx` uses the generic `getFM.m` files such that only the `main.m` file needs to be modified to change models, and the Simulink files can be left untouched. To get this to work, only the relevant model can be present in the path.

```
29
30 % Define reference: [x, y, z, yaw]
31 reference_ = zeros(length(time), 4);
32 reference_(:, 1) = 0;
33 reference_(:, 2) = 5*sin(2*pi*time/5);
34 reference_(:, 3) = -2;
35 reference = struct;
36 reference.time = time;
37 reference.signals.values = reference_;
38 reference.signals.dimensions = 4;
```

To visualize the quadrotor trajectory, before running `main.m`, ensure that the animation is active. If not, open the `simpleModelSim.slx` file, open the `visualization` block (yellow), then open the `UAV Animation` block and click `Show Animation`. Run the `main.m` script. The quadrotor should rise to 2 meters in altitude, and begin tracking a sinusoidal y-position reference signal while keeping x at zero. This trajectory is shown in fig. 32.

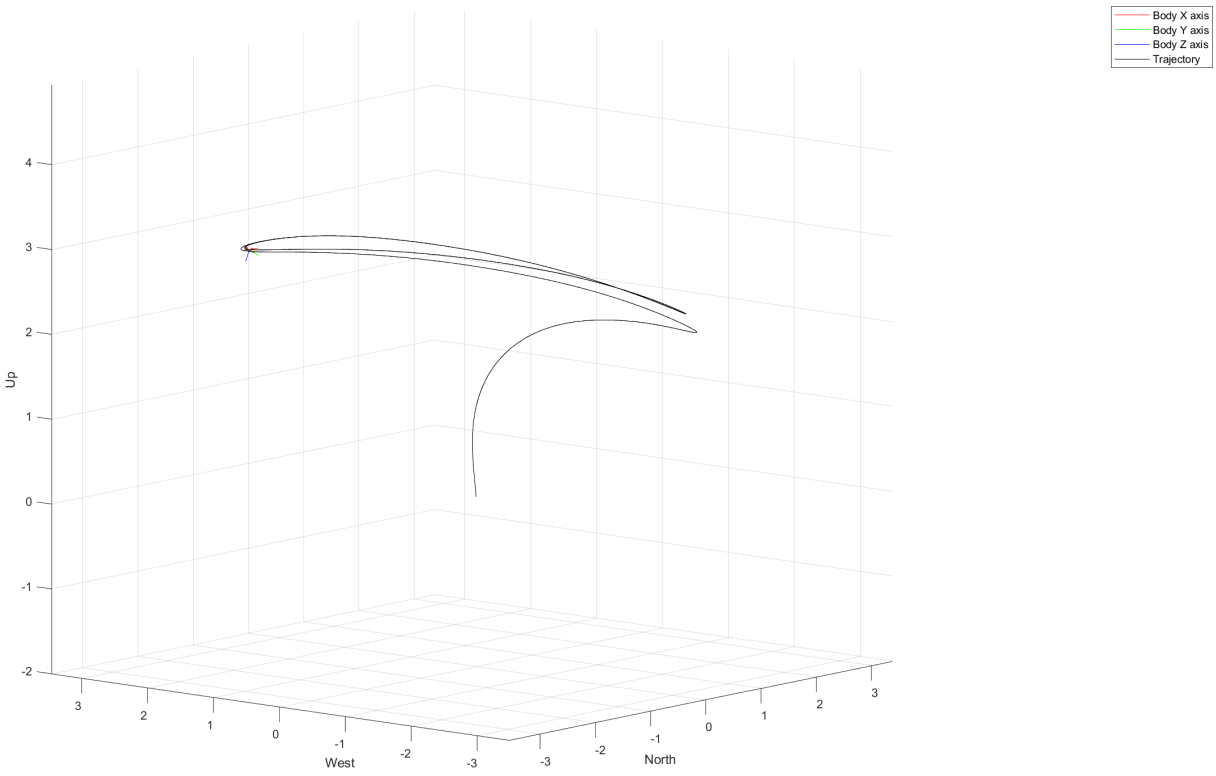


Figure 32: Example animation trajectory of the MATLAB simulation environment

Upon the completion of the simulation, much of the simulation data is saved to the `simdata` variable. These are accessible for further analysis or use. For instance, in the Command Window, type

```
plot(simdata.force)
```

to view the forces throughout the simulation.

9 Additional information and documentation

9.1 Config.json files - Row selection syntax

Within the configuration files, rows may be selected in various ways. The most simple method is to simply list all the rows, as integers, that we would like to use. For example:

```
"rows" : [2, 3, 4, 5, 6, 8, 9, 10]
```

However, this may be inconvenient if there are many rows to import. Alternatively, a string range can be given. Using the same series as above:

```
"rows" : ["2-6", "8-10"]
```

Notice the absence of row 7 in the original series. If "2-10" were used, then row 7 would be included. Instead, formulating two ranges as above would exclude the desired row and retain the original series.

We can also combine the two methods for writing rows. For example, if we would also like to include row 45 to the series, we may write:

```
"rows" : ["2-6", "8-10", 45]
```

9.2 DataFrame column parser

The pipeline supports some basic equation functionality to facilitate the aggregation of regressors without having to define new `pandas.DataFrame` columns all the time. The equation parser assumes that all elements of a given equation are either numbers, operators, (standard) brackets, or variables (columns) in the inputted `pandas.DataFrame`. The equation parser can handle the following operators:

- **Addition** using: `"+"`
- **Subtraction** using: `"-"`
- **Multiplication** using: `"*"`
- **Division** using: `"/"`
- **Powers** using: `"^"`

Moreover, the order of operations can be controlled through standard brackets: `"(·)"`. When working with powers, it is strongly recommended to use these brackets to denote what variables or numbers should be raised as a power. Furthermore, as the standard brackets are reserved for denoting operation order, they **cannot** be used in the columns of the inputted `pandas.DataFrame`, use square brackets instead: `"[·]"`. For example, if users want to specify a variable as "F(x)", please use "F[x]" instead. Otherwise the parser will look for variables "F" and "x", not the column "F(x)". This is also the case for any of the operators listed above (i.e. columns using any of the operators will not be parsed correctly).

Below are examples of the types of equations that can be constructed (assuming the variables are "a", "b", "c", and "d" in the inputted `pandas.DataFrame`):

- $1 + \frac{40}{2}$ → `"1 + 40/(2)"`
- $\frac{1}{a-1} + b - (c + \frac{d}{b})$ → `"1/(a-1) + b - (c + d/(b))"`
- $(a + b)^{1.234}$ → `"(a + b)^(1.234)"`
- $c(a + b + 3.14)^{\frac{d}{c-a}b^2}$ → `"c*(a + b + 3.14)^(d/(c-a)*b^(2))"`

Indeed, these equations can become quite complex. Note that, if a constructed equation is used often, then is recommended that it is added to the `pandas.DataFrame` for efficiency. Therefore, these equations are most useful for the fixed regressor construction. For example, a simple thrust model of the quadrotor can be defined by

$$T = -\kappa \sum_{i=1}^{4} \omega_i^2$$

which a useful (fixed) regressor to have in the F_z models. This regressor can be constructed from the base rotor speed columns ("w1", "w2", "w3", "w4") through:

```
"(w1^(2) + w2^(2) + w3^(2) + w4^(2)"
```

The equation parser is used when reading the polynomial specification file, thus, arbitrary equations can be made for both the candidate regressors and fixed regressors. For example:

```
"Fx":{
    "candidates":[
        {
            "vars":["(u + v - w)", "u", "w1 - w2 + w3 - w4", "sin[
               roll]"],
            "degree":4,
            "sets":[1, "w_tot", "q^(10 - p*r)", "U_q + 3.47"]
        },
        {
            "vars":["q/(p - r)", "q*q"],
            "degree":3,
            "sets":[1, "w_tot*((u + v)^(w-2))"]
        }
    ],
    "fixed":["u*(sin[pitch]+u)*v + w^(u/(v) + p*r + q)"]
}
```

is a permissible candidate regressor formulation (assuming the quadrotor polynomial variables are available in the provided `pandas.DataFrame`).

9.3 Example polynomial regressor specification File

Listing 17 gives an example of a polynomial regressor specification file for quadrotor aerodynamic model identification. More details on the syntax of entries can be found in section 6.4.

Listing 17: Example of a polynomial fixed and candidate regressor specification file

```
{
    "Fx":{
        "candidates":[
            {
                "vars":["u"],
                "degree":4,
                "sets":[1, "w_tot", "q", "U_q"]
            },
            {
                "vars":["q"],
                "degree":3,
                "sets":[1, "w_tot"]
            }
```

```
        ],
        "fixed":["u"]
    },

    "Fy":{
        "candidates":[
            {
                "vars":["v"],
                "degree":4,
                "sets":[1, "w_tot", "p", "U_p"]
            },
            {
                "vars":["p"],
                "degree":3,
                "sets":[1, "w_tot"]
            }
        ],
        "fixed":["v"]
    },

    "Fz":{
        "candidates":[
            {
                "vars":["|u|", "|v|", "w"],
                "degree":4,
                "sets":[1, "d_w_tot", "|p|", "|q|", "|r|", "|U_p|",
                    "|U_q|", "|U_r|"]
            },
            {
                "vars":["|p|", "|q|", "|r|"],
                "degree":3,
                "sets":[1, "d_w_tot"]
            },
            {
                "vars":["d_w_tot"],
                "degree":4,
                "sets":[1]
            }
        ],
        "fixed":["w", "(w2_1 + w2_2 + w2_3 + w2_4)"]
    },

    "Mx":{
        "candidates":[
            {
                "vars":["p"],
                "degree":4,
                "sets":[1]
            },
            {
                "vars":["U_p"],
                "degree":4,
                "sets":[1]
```

```
            }
        ],
        "fixed":["p", "U_p"]
    },

    "My":{
        "candidates":[
            {
                "vars":["q"],
                "degree":4,
                "sets":[1]
            },
            {
                "vars":["U_q"],
                "degree":4,
                "sets":[1]
            }
        ],
        "fixed":["q", "U_q"]
    },

    "Mz":{
        "candidates":[
            {
                "vars":["r"],
                "degree":4,
                "sets":[1]
            },
            {
                "vars":["U_r"],
                "degree":4,
                "sets":[1]
            }
        ],
        "fixed":["r", "U_r"]
    }
}
```

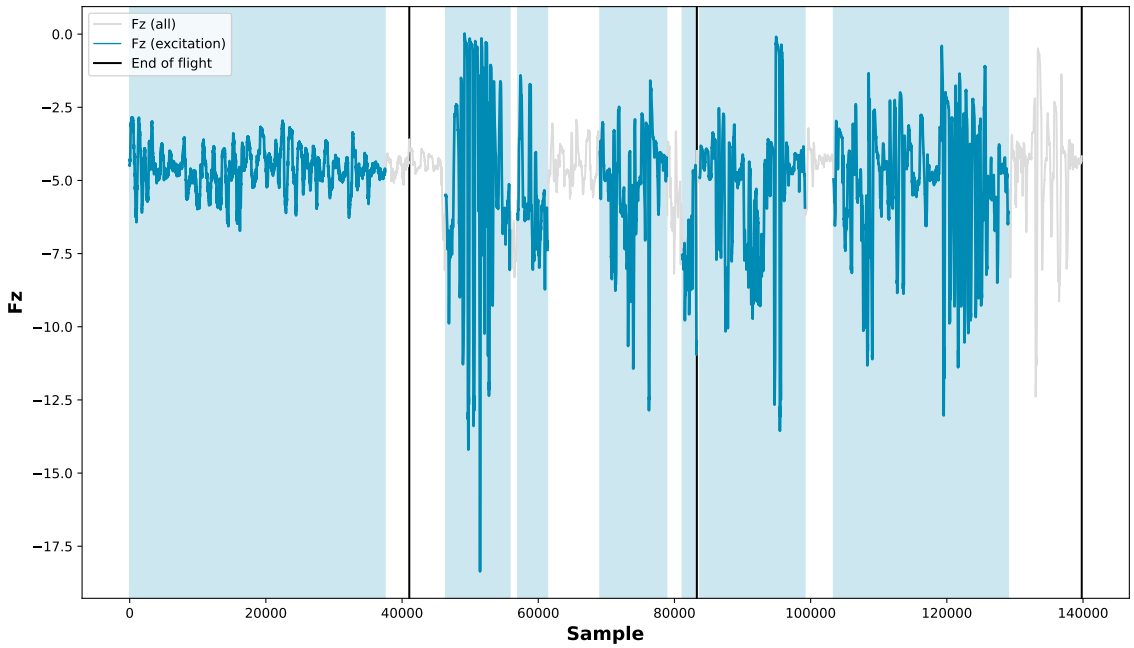
9.4 Model identification extras

9.4.1 Manoeuvre isolation

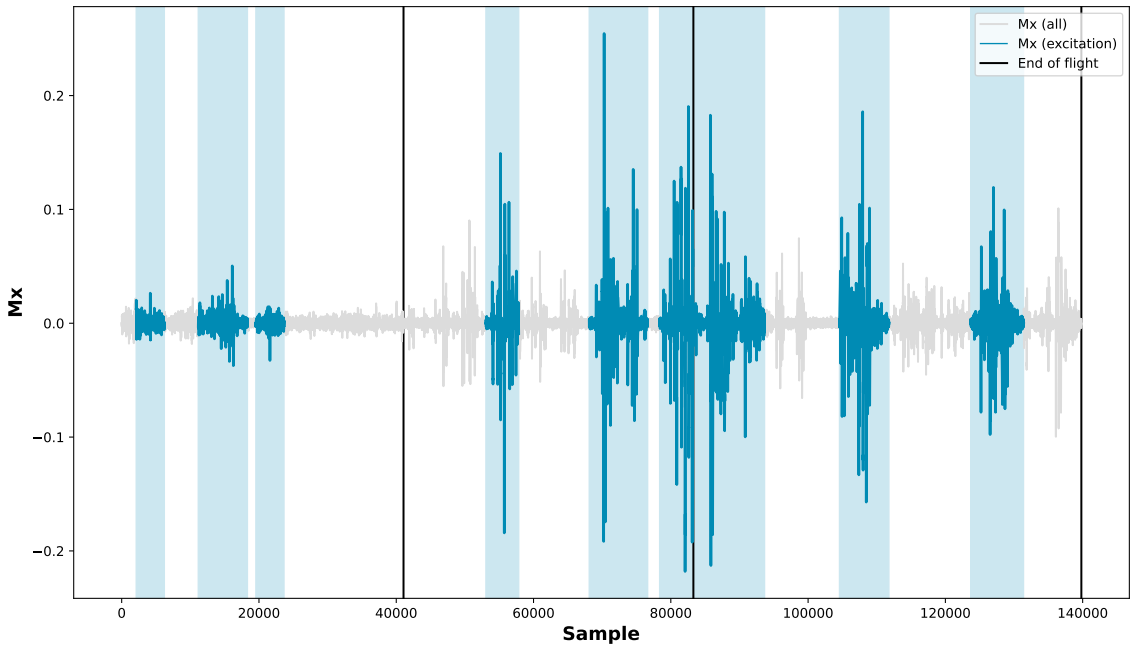
In order to identify suitable models, there should be sufficient excitation in the flight data such that the input-output behaviour can be mapped. Indeed, there may be regions in the flight data logs where excitations are absent and potentially for modelling.

The pipeline provides some utility to constrain the training data to relevant regions of excitation. Figure 33 illustrates the results of the manoeuvre isolation algorithms for arbitrary flights of the HDBeetle quadrotor. Note that the different flight logs are delimited by the black vertical line.

However, the underlying algorithms are not perfect and thus may not (initially) produce the desired results. For example, in fig. 33a, it can be argued that the algorithm should include the manoeuvres at the end of the third flight (i.e. samples 130000+) over those of the first flight (i.e. samples 0 to around 40000) since the former manoeuvres are larger in magnitude. The reason



(a) F_z



(b) M_x

Figure 33: Example result of the manoeuvre isolation algorithms.

for this discrepancy is that the algorithm considers the relative magnitude of the manoeuvre within a given flight and not over the entire data set. Indeed, a global model should be valid at both low and high magnitudes of force.

Nonetheless, users can alter the algorithm hyper-parameters to produce the desired results. It is recommended that you consult the resultant manoeuvre isolation figures to confirm the algorithm results.

References

[1] S. F. Armanini, M. Karásek, G. C. H. E. de Croon, and C. C. de Visser, “Onboard/offboard sensor fusion for high-fidelity flapping-wing robot flight data,” *Journal of Guidance, Control, and Dynamics*, vol. 40, no. 8, pp. 2121–2132, 2017.

[2] V. Klein and E. A. Morelli, *Aircraft System Identification: Theory and Practice*. 2006.

[3] J. J. van Beers and C. C. de Visser, *Peaking into the Black-box: Prediction Intervals Give Insight into Data-driven Quadrotor Model Reliability*.

[4] E. Smeur, G. de Croon, and Q. Chu, “Cascaded incremental nonlinear dynamic inversion for mav disturbance rejection,” *Control Engineering Practice*, vol. 73, pp. 79–90, 2018. [Accessed 25-05-2023].