
casiopeia

Release 0.1

January 13, 2017

1	Contents	3
1.1	Get and install casiopeia	3
1.2	Defining a system	8
1.3	System simulation	8
1.4	Parameter estimation	9
1.5	Optimum experimental design	12
1.6	Sample applications	15
	Python Module Index	21



`casiopeia` holds a user-friendly environment for optimum experimental design and parameter estimation and identification applications. It does so by providing Python classes that can be initialized with the problem specifications, while the computations can then easily be performed using the available class functions.

`casiopeia` uses the optimization framework `CasADi` to solve the resulting optimization problems.

Note: `casiopeia` is still in it's testing state, and does not yet contain all the features it will provide in future versions. Therefore, you should check for updates on a regular basis.

In the following sections, you will receive the information necessary to obtain, install and use `casiopeia`. If you encounter any problems using this software, please feel free to submit your errors with a description of how they occurred to adrian.buerger@hs-karlsruhe.de.

New: [try casiopeia live in your browser](#) ¹

¹ This service is at the moment limited to one user at a time, due to restricted resources. If your computations do not start immediately, there's probably another user testing `casiopeia` at the moment.

Contents

1.1 Get and install casiopeia

Within the next sections, you will get to know how to obtain and install casiopeia, and what prerequisites have to be met to get casiopeia working correctly.

1.1.1 Installation on Ubuntu 16.04

The following instructions show the installation on Ubuntu 16.04. If you are planning to install casiopeia on Linux systems different from Ubuntu 16.04, these commands need to be adapted accordingly.

Prerequisites

Python

In order to use casiopeia, please make sure that [Python](#) (currently supported version is Python 2.7) as well as [Python Numpy](#) (≥ 1.8), [PyLab](#) and [Python Setuptools](#) are installed on your system. This can easily be ensured by running

```
sudo apt-get update
sudo apt-get install python python-numpy python-scipy python-matplotlib python-setuptools --install-
```

If you want to install casiopeia using [pip](#), which is the recommended and easiest way, you also need to install pip by running

```
sudo apt-get install python-pip
```

Also, you might want to install the [Spyder IDE](#) for working with Python. You can install it by running

```
sudo apt-get install spyder
```

Note: These commands require root privileges. In case you do not have root privileges on your system, consider using [Miniconda](#) to install Python and the necessary modules into a user-writeable directory.

CasADi

For casiopeia to work correctly, you need [CasADi](#) version ≥ 3.1 to be installed on your system. Installation instructions for CasADi can be found [here](#).

Note: Some plugins for CasADi require extra prerequisites to work on Linux. For a list of the required libraries and installation instructions, see [the corresponding section in the CasADi installation guide](#). If something goes wrong with executing CasADi and/or casiopeia, missing one or more of these libraries might be the reason.

In addition to unpacking the archive you just obtained, please make sure that the unpacked folder that contains CasADi can be found by Python permanently. As mentioned in the CasADi installation instructions, this can e. g. be ensured by adding the CasADi directory to the `PYTHONPATH` variable permanently. Just open the file `~/.bashrc` on your system with your favorite text editor, and add the line

```
export PYTHONPATH=$PYTHONPATH:/<path>/<to>/<casadi>/<folder>
```

while `/<path>/<to>/<casadi>/<folder>` needs to be adapted to the path of your unpacked archive. Afterwards, save these changes and close all open terminals. Now open a new terminal, and have a look at the value of `PYTHONPATH` by typing

```
echo $PYTHONPATH
```

It should now contain at least the path you just inserted. If everything went well, you should be able to open a Python console, and execute the following commands

```
>>> from casadi import *
>>> x = MX.sym("x")
>>> print jacobian(sin(x), x)
```

without receiving error messages.

Option 1: Get casiopeia using pip (recommended)

Installation

casiopeia is listed on the [Python Package Index](#). You can obtain it from there by running

```
sudo pip install casiopeia
```

If this command fails with a message that CasADi cannot be found on your system, and you installed CasADi by appending its directory to `PYTHONPATH` via `~/.bashrc`, it's most likely that your users `PYTHONPATH` variable is not available when using `sudo`. In this case, try

```
sudo env PYTHONPATH=$PYTHONPATH pip install casiopeia
```

Note: These commands require root privileges. In case you do not have root privileges on your system, consider [Option 2: Get casiopeia from GitHub](#).

Upgrades

Upgrades to new releases of casiopeia can simply be obtained by running

```
sudo pip install casiopeia --upgrade
```

or


```
sudo env PYTHONPATH=$PYTHONPATH pip install casiopeia --upgrade
```

respectively.

Note: These commands require root privileges.

Option 2: Get casiopeia from GitHub

Installation

You can also obtain the casiopeia module [directly from its GitHub repository](#). You can either clone the repository, or download the contained files within a compressed archive. To just obtain an archive, you do not need to have [git](#) installed, but cloning the repository provides an easy way to receive updates on casiopeia by pulling from the repository.

You can install git by running

```
sudo apt-get update
sudo apt-get install git
```

Note: These commands require root privileges. In case you do not have root privileges and git is not installed on your system, consider downloading the archive from the [GitHub page](#) using your favorite web browser instead of cloning the git repository.

Afterwards, you can clone the repository using the following commands

```
git clone git@github.com:adbuerger/casiopeia.git
```

and install casiopeia by running

```
sudo python setup.py install
```

from within the casiopeia directory. If this command fails with a message that CasADi cannot be found on your system, and you installed CasADi by appending its directory to PYTHONPATH via `~/.bashrc`, it's most likely that your users PYTHONPATH variable is not available when using `sudo`. In this case, try

```
sudo env PYTHONPATH=$PYTHONPATH python setup.py install
```

Note: These commands require root privileges. In case you do not have root privileges, consider adding the casiopeia directory to PYTHONPATH, as described above for CasADi.

Upgrades

If you received casiopeia by cloning the git repository, you can update the contents of your local copy by running

```
git pull
```

from within the casiopeia directory. In case you did not clone the repository, you would again need to download a compressed archive.

Afterwards, you need to install the recent version again by running

```
sudo python setup.py install
```

or

```
sudo env PYTHONPATH=$PYTHONPATH python setup.py install
```

respectively.

Note: These commands require root privileges.

Warning: If you installed casiopeia by adding the directory to `PYTHONPATH`, just place the newly obtained files in the previously defined path to upgrade to a new version of casiopeia. You do not need to add the directory again to `PYTHONPATH` then. Also, make sure not to add multiple versions of casiopeia to `PYTHONPATH`, since this might lead to conflicts.

1.1.2 Installation on Windows

The following instructions have been tested on Windows 7 64 bit.

Note: You need to have administrator rights on your system to be able to follow the instructions below.

Prerequisites

Python

The easiest way to meet the prerequisites for casiopeia and CasADi on a Windows system might be to install a recent version of [Python\(x,y\)](#), which is also the procedure recommended by the CasADi developers. It is recommended to do a “Full” installation. In the following, the instructions also assume that you are installing Python(x,y) and all components with their default paths.

CasADi

For casiopeia to work correctly, you need [CasADi](#) version ≥ 3.1 to be installed on your system. Installation instructions for CasADi can be found [here](#).

After unpacking the archive, go to `My Computer > Properties > Advanced System Settings > Environment Variables`. If a variable `PYTHONPATH` already exists, apply the full path to the CasADi folder to the end of the variable value, and separate this new path from the ones already contained by `;`. If `PYTHONPATH` does not yet exist on the system, create a new environmental variable with this name, and fill in the path to the unpacked CasADi folder.

Option 1: Get casiopeia using pip (recommended)

Installation

casiopeia is listed on the [Python Package Index](#). Since you installed [pip](#) with Python(x,y), you can obtain casiopeia by opening a command line and running

```
pip install casiopeia
```

Note: If you have problems obtaining casiopeia with pip (which can e. g. be caused by a company's proxy server) consider *Option 2: Get casiopeia from GitHub*.

Upgrades

Upgrades to new releases of casiopeia can simply be obtained by running

```
pip install casiopeia --upgrade
```

Option 2: Get casiopeia from GitHub

Installation

You can also obtain the casiopeia module [directly from its GitHub repository](#). Since installing git is more time-consuming on Windows than it is on most Linux systems, it is recommended (at least for less experienced users) to just download the contained files for casiopeia within a compressed archive.

Afterwards, unpack the archive, and install casiopeia by running

```
python setup.py install
```

from the command line, within the unzipped folder.

Note: If this procedure is for some reason not applicable for you, you can consider adding the casiopeia directory to PYTHONPATH instead, as described above for CasADi.

Upgrades

For upgrading casiopeia, you would again need to download a compressed archive.

Afterwards, you need to install the recent version by again running

```
python setup.py install
```

Warning: If you installed casiopeia by adding the directory to PYTHONPATH, just place the newly obtained files in the previously defined path to upgrade to a new version of casiopeia. You do not need to add the directory again to PYTHONPATH then. Also, make sure not to add multiple versions of casiopeia to PYTHONPATH, since this might lead to conflicts.

1.1.3 Recommendations

To speed up computations in casiopeia, it is recommended to install [HSL for IPOPT](#). On how to install the solvers and for further information, see the page [Obtaining HSL](#) in the CasADi wiki.

1.2 Defining a system

Since *casiopia* uses CasADi, the user first has to define the considered system using CasADi symbolic variables (of type MX). Afterwards, the symbolic variables which define states, controls, parameters, etc. of the system can be brought into connection by creating a `casiopia.System` object.

```
class casiopia.system.System(u='MX(u)', q='MX(q)', p=None, x='MX(x)', eps_u='MX(eps_u)',
                             phi=None, f='MX(f)', g='MX(g)')
```

The class `System` is used to define non-dynamic, explicit ODE- or fully implicit DAE-systems within *casiopia*.

Raises `TypeError`, `NotImplementedError`

Parameters

- **u** (`casadi.casadi.MX`) – time-varying controls $u \in \mathbb{R}^{n_u}$ that are applied piece-wise-constant for each control intervals, and therefor can change from on interval to another, e. g. motor dutycycles, temperatures, massflows (optional)
- **q** (`casadi.casadi.MX`) – time-constant controls $q \in \mathbb{R}^{n_q}$ that are constant over time, e. g. initial mass concentrations of reactants, elevation angles (optional)
- **p** (`casadi.casadi.MX`) – unknown parameters $p \in \mathbb{R}^{n_p}$
- **x** (`casadi.casadi.MX`) – differential states $x \in \mathbb{R}^{n_x}$ (optional)
- **eps_u** (`casadi.casadi.MX`) – input errors $\epsilon_u \in \mathbb{R}^{n_{\epsilon_u}}$ (optional)
- **phi** (`casadi.casadi.MX`) – output function $\phi(u, q, x, p) = y \in \mathbb{R}^{n_y}$
- **f** (`casadi.casadi.MX`) – explicit system of ODEs $f(u, q, x, p, \epsilon_u) = \dot{x} \in \mathbb{R}^{n_x}$ (optional)
- **g** (`casadi.casadi.MX`) – equality constraints $g(u, q, p) = 0 \in \mathbb{R}^{n_g}$ (optional)

Depending on the inputs the user provides, the `System` is interpreted as follows:

Non-dynamic system ($x = \text{None}$):

$$\begin{aligned} y &= \phi(u, q, p) \\ 0 &= g(u, q, p). \end{aligned}$$

Explicit ODE system ($x \neq \text{None}$):

$$\begin{aligned} y &= \\ \phi(u, q, x, p) \\ \dot{x} &= \\ f(u, q, x, p, \epsilon_u). \end{aligned}$$

This system object can now be used within the *casiopia* simulation, parameter estimation and optimum experimental design classes.

1.3 System simulation

The module `casiopia.sim` contains the class used for system simulation.

class `casiopelia.sim.Simulation` (*system*, *pdata*, *qdata=None*)

The class `casiopelia.sim.Simulation` is used to simulate dynamic systems defined with the `casiopelia.system.System` class. It is supposed that the system contains a number of time-constant parameters p .

Parameters

- **system** (`casiopelia.system.System`) – system considered for simulation, specified using the `casiopelia.system.System` class
- **pdata** (`numpy.ndarray`, `casadi.DMatrix`) – values of the time-constant parameters $p \in \mathbb{R}^{n_p}$
- **qdata** (`numpy.ndarray`, `casadi.DMatrix`) – optional, values of the time-constant controls $q \in \mathbb{R}^{n_q}$; if no values are given, 0 will be used

run_system_simulation (*x0*, *time_points*, *udata=None*, *integrator_options={}*, *print_status=True*)

Parameters

- **x0** (`numpy.ndarray`, `casadi.DMatrix`, *list*) – state values $x_0 \in \mathbb{R}^{n_x}$ at the first time point t_0
- **time_points** (`numpy.ndarray`, `casadi.DMatrix`, *list*) – switching time points for the controls $t_N \in \mathbb{R}^N$
- **udata** (`numpy.ndarray`, `casadi.DMatrix`) – optional, values for the time-varying controls at the first $N - 1$ switching time points $u_N \in \mathbb{R}^{n_u \times N-1}$; if no values are given, 0 will be used
- **integrator_options** (*dict*) – optional, options to be passed to the CasADi integrator (see the CasADi documentation for a list of all possible options)
- **print_status** (*bool*) – optional, set to `True` (default) or `False` to enable or disable console printing.

This function will run a system simulation for the specified initial state values and control data from t_0 to t_N .

If you receive integrator-related error messages during the simulation, please check the corresponding parts of the CasADi documentation.

After the simulation has finished, the simulation results x_N can be accessed via the class attribute `Simulation.simulation_results`.

1.4 Parameter estimation

The module `casiopelia.pe` contains the classes for parameter estimation. For now, only least squares parameter estimation problems are covered.

1.4.1 Parameter estimation from single experiments

class `casiopelia.pe.LSq` (*system*, *time_points*, *udata=None*, *qdata=None*, *ydata=None*, *pinit=None*, *xinit=None*, *wv=None*, *weps_u=None*, *discretization_method='collocation'*, ***kwargs*)

The class `casiopelia.pe.LSq` is used to set up least squares parameter estimation problems for systems defined with the `casiopelia.system.System` class, using a given set of user-provided control data, measurement data and different kinds of weightings.

Raises `AttributeError`, `NotImplementedError`

Parameters

- **system** (`casiopeia.system.System`) – system considered for parameter estimation, specified using the `casiopeia.system.System` class
- **time_points** (`numpy.ndarray`, `casadi.DMatrix`, `list`) – time points $t_N \in \mathbb{R}^N$ used to discretize the continuous time problem. Controls will be applied at the first $N - 1$ time points, while measurements take place at all N time points.
- **udata** (`numpy.ndarray`, `casadi.DMatrix`) – optional, values for the time-varying controls $u_N \in \mathbb{R}^{n_u \times N-1}$ that can change at the switching time points; if no values are given, 0 will be used; note that the second dimension of u_N is $N - 1$ and not N , since there is no control value applied at the last time point
- **qdata** (`numpy.ndarray`, `casadi.DMatrix`) – optional, values for the time-constant controls $q_N \in \mathbb{R}^{n_q}$; if not values are given, 0 will be used
- **ydata** (`numpy.ndarray`, `casadi.DMatrix`) – values for the measurements at the switching time points $y_N \in \mathbb{R}^{n_y \times N}$
- **wv** (`numpy.ndarray`, `casadi.DMatrix`) – weightings for the measurements $w_v \in \mathbb{R}^{n_y \times N}$
- **weeps_u** (`numpy.ndarray`, `casadi.DMatrix`) – weightings for the input errors $w_{\epsilon_u} \in \mathbb{R}^{n_{\epsilon_u}}$ (only necessary if input errors are used within system)
- **pinit** (`numpy.ndarray`, `casadi.DMatrix`) – optional, initial guess for the values of the parameters that will be estimated $p_{\text{init}} \in \mathbb{R}^{n_p}$; if no value is given, 0 will be used; note that a poorly or wrongly chosen initial guess can cause the estimation to fail
- **xinit** (`numpy.ndarray`, `casadi.DMatrix`) – optional, initial guess for the values of the states that will be estimated $x_{\text{init}} \in \mathbb{R}^{n_x \times N}$; if no value is given, 0 will be used; note that a poorly or wrongly chosen initial guess can cause the estimation to fail
- **discretization_method** (`str`) – optional, the method that shall be used for discretization of the continuous time problem w. r. t. the time points given in t_N ; possible values are “collocation” (default) and “multiple_shooting”

Depending on the discretization method specified in `discretization_method`, the following parameters can be used for further specification:

Parameters

- **collocation_scheme** (`str`) – optional, scheme used for setting up the collocation polynomials, possible values are *radau* (default) and *legendre*
- **number_of_collocation_points** (`int`) – optional, order of collocation polynomials $d \in \mathbb{Z}$ (default values is 3)
- **integrator** (`str`) – optional, integrator to be used with multiple shooting. See the CasADi documentation for a list of all available integrators. As a default, *cvodes* is used.
- **integrator_options** (`dict`) – optional, options to be passed to the CasADi integrator used with multiple shooting (see the CasADi documentation for a list of all possible options)

The resulting parameter estimation problem has the following form:

$$\begin{aligned} \arg \min_{p, x, v, \epsilon_u} \quad & \frac{1}{2} \|R(\cdot)\|_2^2 \\ \text{subject to:} \quad & v_k + y_k - \phi(x_k, p; u_k, q) = 0 \quad k = 1, \dots, N \\ & g(x, p, \epsilon_u; u, q) = 0 \\ \text{with:} \quad & \begin{pmatrix} w_v^T & w_{\epsilon_u}^T \end{pmatrix}^{\mu/\mu} \begin{pmatrix} v \\ \epsilon_u \end{pmatrix} = R \end{aligned}$$

while $g(\cdot)$ contains the discretized system dynamics according to the specified discretization method. If the system is non-dynamic, it only contains the user-provided equality constraints.

`compute_covariance_matrix()`

This function computes the covariance matrix for the estimated parameters from the inverse of the KKT matrix for the parameter estimation problem, which allows for statements on the quality of the values of the estimated parameters ^{1 2}.

For efficiency, only the inverse of the relevant part of the matrix is computed ³.

The values of the covariance matrix $\Sigma_{\hat{p}}$ can afterwards be accessed via the class attribute `LSq.covariance_matrix`, and the contained standard deviations $\sigma_{\hat{p}}$ for the estimated parameters directly via `LSq.standard_deviations`.

References

`print_estimation_results()`

Raises `AttributeError`

This function displays the results of the parameter estimation computations. It can not be used before function `run_parameter_estimation()` has been used. The results displayed by the function contain:

- the values of the estimated parameters \hat{p} and their corresponding standard deviations $\sigma_{\hat{p}}$, (the values of the standard deviations are presented only if the covariance matrix had already been computed),
- the values of the covariance matrix $\Sigma_{\hat{p}}$ for the estimated parameters (if it had already been computed), and
- the durations of the estimation and (if already executed) of the covariance matrix computation.

`run_parameter_estimation(solver_options={})`

Parameters `solver_options` (*dict*) – options to be passed to the IPOPT solver (see the CasADi documentation for a list of all possible options)

This functions will pass the least squares parameter estimation problem to the IPOPT solver. The status of IPOPT printed to the console provides information whether the optimization finished successfully. The estimated parameters \hat{p} can afterwards be accessed via the class attribute `LSq.estimated_parameters`.

Note: IPOPT finishing successfully does not necessarily mean that the estimation results for the unknown parameters are useful for your purposes, it just means that IPOPT was able to solve the given

¹ Kostina, Ekaterina and Kostyukova, Olga: *Computing Covariance Matrices for Constrained Nonlinear Large Scale Parameter Estimation Problems Using Krylov Subspace Methods*, 2012.

² Kostina, Ekaterina and Kriwet, Gregor: *Towards Optimum Experimental Design for Partial Differential Equations*, SPP 1253 annual conference 2010, slides 12/13.

³ Walter, Eric and Prozano, Luc: *Identification of Parametric Models from Experimental Data*, Springer, 1997, pages 288/289.

optimization problem. You have in any case to verify your results, e. g. by simulation using the `casiopeia.sim.Simulation` class!

1.4.2 Parameter estimation from multiple experiments

class `casiopeia.pe.MultiLSq`(*pe_setups*=[])

The class `casiopeia.pe.MultiLSq` is used to construct a single least squares parameter estimation problem from multiple least squares parameter estimation problems defined via two or more objects of type `casiopeia.pe.LSq`.

In this way, the results of multiple independent experimental setups can be used for parameter estimation.

Note: It is assumed that the system description used for setting up the several parameter estimation problems is the same.

Parameters `pe_setups` (*list*) – list of two or more objects of type `casiopeia.pe.LSq`

1.5 Optimum experimental design

The module `casiopeia.doe` contains the classes used for optimum experimental design.

1.5.1 Optimum experimental design of single experiments

class `casiopeia.doe.DoE`(*system*, *time_points*, *uinit*=None, *umin*=None, *umax*=None, *qinit*=None, *qmin*=None, *qmax*=None, *pdata*=None, *x0*=None, *xmin*=None, *xmax*=None, *wv*=None, *weps_u*=None, *discretization_method*='collocation', *optimality_criterion*='A', **kwargs)

The class `casiopeia.doe.DoE` is used to set up Design-of-Experiments-problems for systems defined with the `casiopeia.system.System` class.

The aim of the experimental design optimization is to identify a set of controls that can be used for the generation of measurement data which allows for a better estimation of the unknown parameters of a system.

To achieve this, an information function on the covariance matrix of the estimated parameters is minimized. The values of the estimated parameters, though they are mostly an initial guess for their values, are not changed during the optimization.

Optimum experimental design and parameter estimation methods can be used interchangeably until a desired accuracy of the parameters has been achieved.

Raises `AttributeError`, `NotImplementedError`

Parameters

- **system** (`casiopeia.system.System`) – system considered for parameter estimation, specified using the `casiopeia.system.System` class
- **time_points** (`numpy.ndarray`, `casadi.DMatrix`, *list*) – time points $t_N \in \mathbb{R}^N$ used to discretize the continuous time problem. Controls will be applied at the first $N - 1$ time points, while measurements take place at all N time points.
- **umin** (`numpy.ndarray`, `casadi.DMatrix`) – optional, lower bounds of the time-varying controls $u_{\min} \in \mathbb{R}^{n_u}$; if not values are given, $-\infty$ will be used

- **umax** (*numpy.ndarray*, *casadi.DMatrix*) – optional, upper bounds of the time-varying controls $u_{\max} \in \mathbb{R}^{n_u}$; if not values are given, ∞ will be used
- **uinit** (*numpy.ndarray*, *casadi.DMatrix*) – optional, initial guess for the values of the time-varying controls $u_N \in \mathbb{R}^{n_u \times N-1}$ that (might) change at the switching time points; if no values are given, 0 will be used; note that a poorly or wrongly chosen initial guess can cause the optimization to fail, and note that the second dimension of u_N is $N - 1$ and not N , since there is no control value applied at the last time point
- **qmin** (*numpy.ndarray*, *casadi.DMatrix*) – optional, lower bounds of the time-constant controls $q_{\min} \in \mathbb{R}^{n_q}$; if not values are given, $-\infty$ will be used
- **qmax** (*numpy.ndarray*, *casadi.DMatrix*) – optional, upper bounds of the time-constant controls $q_{\max} \in \mathbb{R}^{n_q}$; if not values are given, ∞ will be used
- **qinit** (*numpy.ndarray*, *casadi.DMatrix*) – optional, initial guess for the optimal values of the time-constant controls $q_{\text{init}} \in \mathbb{R}^{n_q}$; if not values are given, 0 will be used; note that a poorly or wrongly chosen initial guess can cause the optimization to fail
- **pdata** (*numpy.ndarray*, *casadi.DMatrix*) – values of the time-constant parameters $p \in \mathbb{R}^{n_p}$
- **x0** (*numpy.ndarray*, *casadi.DMatrix*, *list*) – state values $x_0 \in \mathbb{R}^{n_x}$ at the first time point t_0
- **xmin** (*numpy.ndarray*, *casadi.DMatrix*) – optional, lower bounds of the states $x_{\min} \in \mathbb{R}^{n_x}$; if no value is given, $-\infty$ will be used
- **xmax** (*numpy.ndarray*, *casadi.DMatrix*) – optional, lower bounds of the states $x_{\max} \in \mathbb{R}^{n_x}$; if no value is given, ∞ will be used
- **wv** (*numpy.ndarray*, *casadi.DMatrix*) – weightings for the measurements $w_v \in \mathbb{R}^{n_y \times N}$
- **weeps_u** (*numpy.ndarray*, *casadi.DMatrix*) – weightings for the input errors $w_{\epsilon_u} \in \mathbb{R}^{n_{\epsilon_u}}$ (only necessary if input errors are used within system)
- **discretization_method** (*str*) – optional, the method that shall be used for discretization of the continuous time problem w. r. t. the time points given in t_N ; possible values are “collocation” (default) and “multiple_shooting”
- **optimality_criterion** (*str*) – optional, the information function $I_X(\cdot)$ to be used on the covariance matrix, possible values are A (default) and D, while

$$I_A(\Sigma_p) = \frac{1}{n_p} \text{Tr}(\Sigma_p),$$

$$I_D(\Sigma_p) = |\Sigma_p|^{\frac{1}{n_p}},$$

for further information see e. g. ¹

Depending on the discretization method specified in *discretization_method*, the following parameters can be used for further specification:

Parameters

- **collocation_scheme** (*str*) – optional, scheme used for setting up the collocation polynomials, possible values are *radau* (default) and *legendre*
- **number_of_collocation_points** (*int*) – optional, order of collocation polynomials $d \in \mathbb{Z}$ (default values is 3)

¹ Körkel, Stefan: *Numerische Methoden für Optimale Versuchsplanungsprobleme bei nichtlinearen DAE-Modellen*, PhD Thesis, Heidelberg university, 2002, pages 74/75.

- **integrator** (*str*) – optional, integrator to be used with multiple shooting. See the CasADi documentation for a list of all available integrators. As a default, *cvodes* is used.
- **integrator_options** (*dict*) – optional, options to be passed to the CasADi integrator used with multiple shooting (see the CasADi documentation for a list of all possible options)

You do not need to specify initial guesses for the estimated states, since these are obtained with a system simulation using the initial states and the provided initial guesses for the controls.

The resulting optimization problem has the following form:

$$\begin{aligned} \arg \min_{u,q,x} \quad & I(\Sigma_p(x, u, q; p)) \\ \text{subject to:} \quad & g(x, u, q; p) = 0 \\ & u_{\min} \leq u_k \leq u_{\max} \quad k = 1, \dots, N-1 \\ & x_{\min} \leq x_k \leq x_{\max} \quad k = 1, \dots, N \\ & x_1 \leq x(t_1) \leq x_1 \end{aligned}$$

where $\Sigma_p = \text{Cov}(p)$ and $g(\cdot)$ contains the discretized system dynamics according to the specified discretization method. If the system is non-dynamic, it only contains the user-provided equality constraints.

References

plot_confidence_ellipsoids (*properties*='initial')

Parameters **properties** (*str*) – Set whether the experimental properties for the initial setup (“initial”, default), the optimized setup (“optimized”) or for both setups (“all”) shall be plotted. In the later case, both ellipsoids for one pair of parameters will be displayed within one plot.

Plot confidence ellipsoids for all parameter pairs. Since the number of plots is possibly big, all plots will be saved within a folder *confidence_ellipsoids_scriptname* in you current working directory rather than being displayed directly.

1.5.2 Optimum experimental design of multiple experiments

class *casiopeia.doe.MultiDoE* (*doe_setups*=[], *optimality_criterion*='A')

The class *casiopeia.doe.MultiDoE* is used to construct a single experimental design problem from multiple experimental design problems defined via two or more objects of type *casiopeia.doe.DoE*.

This provides the possibility to design multiple experiments within one single optimization, so that the several experiments can focus on different aspects of the system which in combination then yields more information about the complete system.

Also, this functionality is in particular useful in case an experiment is limited to only small variable bounds, small time horizons, highly depends on the initialization of the system, or any other case when a single experiment might not be enough to capture enough information about a system.

Note: It is assumed that the system description used for setting up the several experimental design problems is the same!

Parameters **doe_setups** (*list*) – list of two or more objects of type *casiopeia.doe.DoE*

1.6 Sample applications

The following sample applications give hands-on impressions on how to use casiopeia in practice. They all (and some more) are contained within the [examples directory](#) of the casiopeia sources.

1.6.1 Parameter estimation for a Lotka-Volterra predator-prey-model

The aim of the application [lotka_volterra_pe.py](#) is to estimate the unknown parameters of a Lotka-Volterra predator-prey-model for experimentally received measurement data and given standard deviations for the measurements ¹. The predator-prey-model is an ODE of the form $\dot{x} = f(x, p)$, given by

$$\begin{aligned}\dot{x}_1 &= -\alpha x_1 + \beta x_1 x_2 \\ \dot{x}_2 &= \gamma x_2 - \delta x_1 x_2\end{aligned}$$

where $\alpha = 1.0$ and $\gamma = 1.0$, the states x and parameters p are defined as

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \quad p = \begin{pmatrix} \beta \\ \delta \end{pmatrix},$$

and we can measure the full state, i. e.

$$\phi = x.$$

The values resulting from the parameter estimation are

$$\hat{p} = \begin{pmatrix} \hat{\beta} \\ \hat{\delta} \end{pmatrix} = \begin{pmatrix} 0.693379029 \\ 0.341128482 \end{pmatrix}.$$

The results for the system simulation using the estimated parameters in comparison to the measurement data are shown in the figure below.

1.6.2 Parameter estimation for a pendulum model

The aim of the application [pendulum_pe.py](#) is to estimate the spring constant k of a pendulum model for experimentally received measurement data ². The pendulum model is an ODE of the form $\dot{x} = f(x, u, p)$, given by

$$\begin{aligned}\dot{\nu} &= \omega \\ \dot{\omega} &= \frac{k}{mL^2}(\psi - \nu) - \frac{g}{L} * \sin(\nu)\end{aligned}$$

where $m = 1.0$, $L = 3.0$ and $g = 9.81$, the states x , controls u and parameters p are defined as

$$x = \begin{pmatrix} \nu \\ \omega \end{pmatrix}, \quad u = (\psi), \quad p = (k),$$

while the only control ψ is the initial actuation angle of the pendulum, and therefor stays constant over time. Also, we can measure the full state, i. e.

$$\phi = x.$$

The value resulting from the parameter estimation is

$$\hat{p} = (\hat{k}) = (2.99763513).$$

The results for the system simulation using the estimated parameter in comparison to the measurement data are shown in the figures below.

¹ Bock, Sager et al.: *Übungen zur Numerischen Mathematik II*, sheet 9, IWR, Heidelberg university, 2006.

² Diehl, Moritz: *Course on System Identification*, exercise 7, SYSCOP, IMTEK, University of Freiburg, 2014/2015.

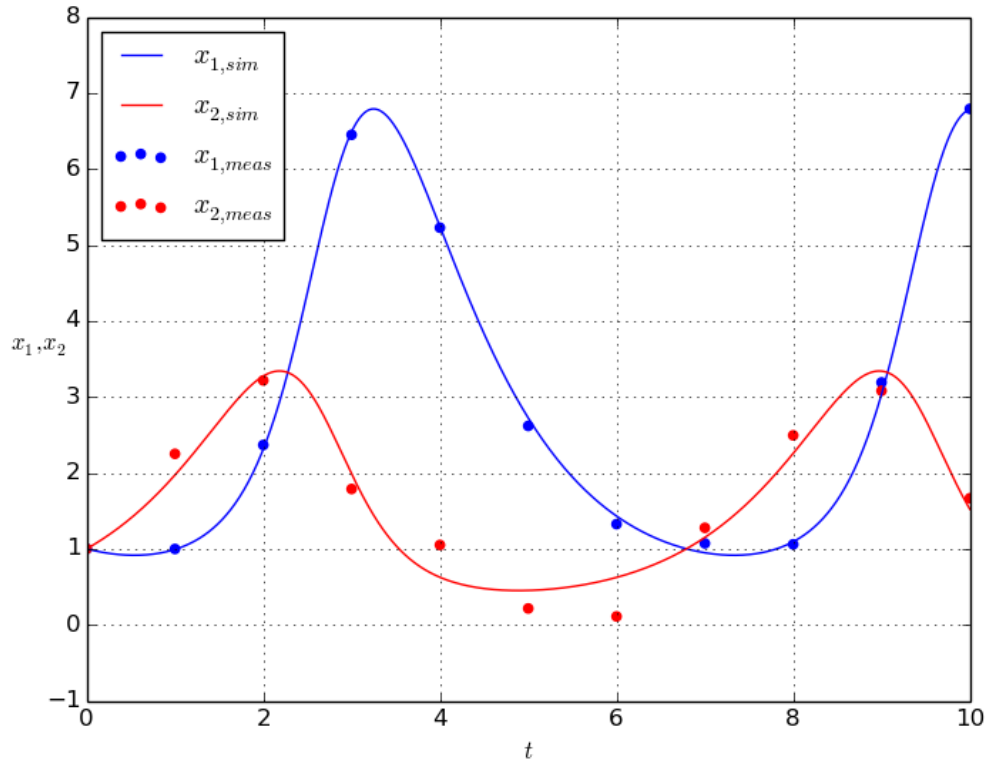


Fig. 1.1: Figure: Simulation results for the Lotka-Volterra predator-prey-model using the estimated parameters, compared to the given measurement data

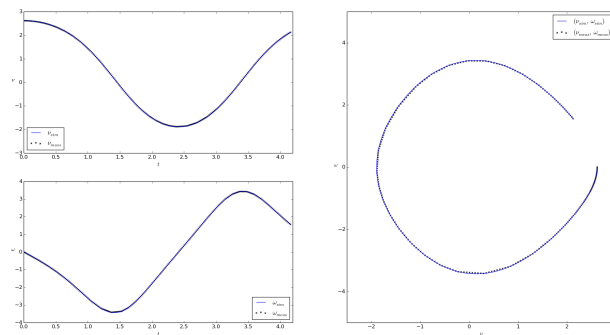


Fig. 1.2: Figure: Simulation results for the pedulum model using the estimated parameters, compared to the given measurement data

1.6.3 Parameter estimation for a model race car

The aim of the application `2d_vehicle_pe.py` is to estimate the unknown parameters of a 2D race car model for experimentally received measurement data³. The race car and the interpretation of the model states are shown in the figure below⁴.

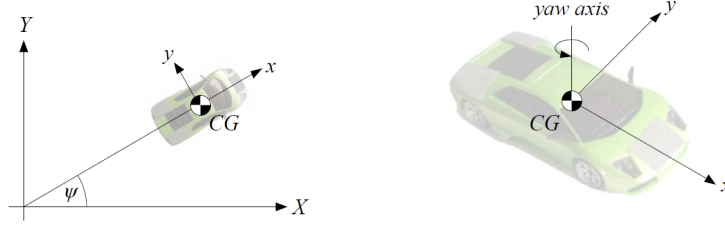


Fig. 1.3: Figure: Depiction of the race car showing the models states

The 2D model of the race car is an ODE of the form $\dot{x} = f(x, u, p)$, given by

$$\begin{aligned}\dot{X} &= v \cos(\psi + C_1 \delta) \\ \dot{Y} &= v \sin(\psi + C_1 \delta) \\ \dot{\psi} &= v \delta C_2 \\ \dot{v} &= C_{m_1} D - C_{m_2} D v - C_{r_2} v^2 - C_{r_0} - (v \delta)^2 C_2 C_1,\end{aligned}$$

where the states x , controls u and parameters p are defined as

$$x = \begin{pmatrix} X \\ Y \\ \psi \\ v \end{pmatrix}, \quad u = \begin{pmatrix} \delta \\ D \end{pmatrix}, \quad p = \begin{pmatrix} C_1 \\ C_2 \\ C_{m_1} \\ C_{m_2} \\ C_{r_2} \\ C_{r_0} \end{pmatrix},$$

and we can measure the full state, i. e.

$$\phi = x.$$

The values resulting from the parameter estimation are

$$\hat{p} = \begin{pmatrix} \hat{C}_1 \\ \hat{C}_2 \\ \hat{C}_{m_1} \\ \hat{C}_{m_2} \\ \hat{C}_{r_2} \\ \hat{C}_{r_0} \end{pmatrix} = \begin{pmatrix} 0.273408 \\ 11.5602 \\ 2.45652 \\ 7.90959 \\ -0.44353 \\ -0.249098 \end{pmatrix}.$$

The results for the system simulation using the estimated parameter in comparison to the measurement data are shown in the figures below.

An evaluation of the covariance matrix for the estimated parameters shows that the standard deviations of \hat{C}_1 and \hat{C}_2 are relatively small in comparison to their own values, while the standard deviations of the other parameters are

³ Verschueren, Robin: *Design and implementation of a time-optimal controller for model race cars*, Master's thesis, KU Leuven, 2014.

⁴ Spengler, Patrick and Gammeter, Christoph: *Modeling of 1:43 scale race cars*, Master's thesis, ETH Zürich, 2010.

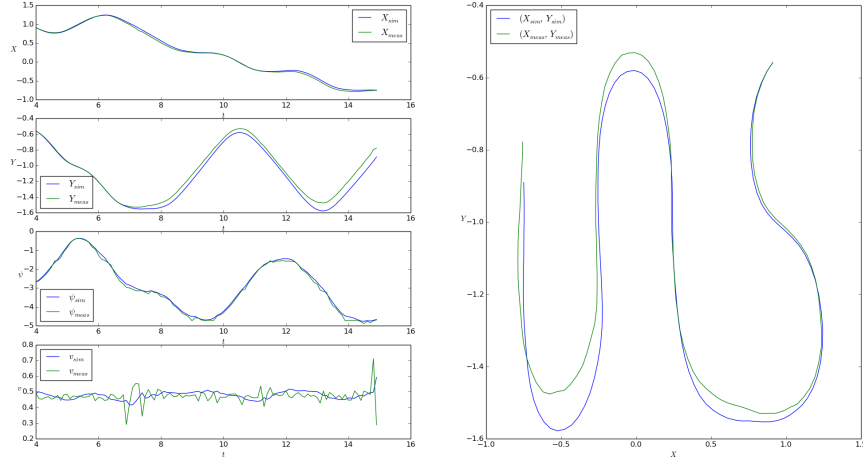


Fig. 1.4: Figure: Simulation results for the race car model using the estimated parameters, compared to the given measurement data

relatively big.

$$\hat{p} = \begin{pmatrix} \hat{C}_1 \\ \hat{C}_2 \\ \hat{C}_{m1} \\ \hat{C}_{m2} \\ \hat{C}_{r2} \\ \hat{C}_{r0} \end{pmatrix} = \begin{pmatrix} 0.273408 \\ 11.5602 \\ 2.45652 \\ 7.90959 \\ -0.44353 \\ -0.249098 \end{pmatrix} \pm \begin{pmatrix} 0.034497452 \\ 0.058569592 \\ 2.72097859 \\ 5.448817078 \\ 1.478999406 \\ 0.37343932 \end{pmatrix}$$

This intends that the estimation results for the parameters \hat{C}_{m1} , \hat{C}_{m2} , \hat{C}_{r2} and \hat{C}_{r0} are probably not accurate, and might change substantially for other measurement and control data. Optimum experimental design can be an option to encounter this problem.

1.6.4 Optimum experimental design for a model race car

The aim of the application `2d_vehicle_doe_scaled.py` is to solve an optimum experimental design problem for the 2D race car model from *Parameter estimation for a model race car* to obtain control values that allow for a better estimation of the unknown parameters of the model.

Initial setup

For this application, we assume that we are not bound to the previous race track to obtain measurements for the race car, but can drive the car on a rectangular mat of the racetrack's material. The controls are bounded by the maximum and minimum values of the controls measurements from *Parameter estimation for a model race car*, as well as the states are bounded by their corresponding maximum and minimum values of the states measurements. The bounds are introduced to prevent the optimizer from creating unrealistic scenarios that could e. g. cause the race car to fall over when taking too sharp turns, which is not explicitly considered within the model.

The previous parameter estimation results \hat{p} from *Parameter estimation for a model race car* are used as a “guess” for the parameter values for the experimental design, and with this, to scale all parameter values within the optimization to 1.0 to prevent influences of the numerical values of the parameters on the optimization result.

A subset of the control values from the previous estimation is used as initial guesses for the optimized controls. The quality of the initial experimental setup in terms of estimated standard deviations of the unknown parameters is

evaluated as follows

$$p_I = \begin{pmatrix} C_{1,I} \\ C_{2,I} \\ C_{m_1,I} \\ C_{m_2,I} \\ C_{r_2,I} \\ C_{r_0,I} \end{pmatrix} = \begin{pmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{pmatrix} \pm \begin{pmatrix} 6.1591763006 \\ 0.318683714861 \\ 92.0037213296 \\ 62.6460661875 \\ 286.556042737 \\ 108.733245939 \end{pmatrix}$$

which indicates that the experimental setup is rather inappropriate for a sufficient estimation.

Optimized setup

Note: Running this optimization takes about 10 min on an Intel(R) Core(TM) i5-4570 3.20GHz CPU.

We use the A-criterion as objective for the experimental design (see *Optimum experimental design*). The results of the optimization can be analyzed and visualized with the script `2d_vehicle_doe_scaled_validation.py`. The figure below shows the optimized control values in comparison to the initially used control values, while the suffix *coll* indicates that the values were obtained using collocation discretization.

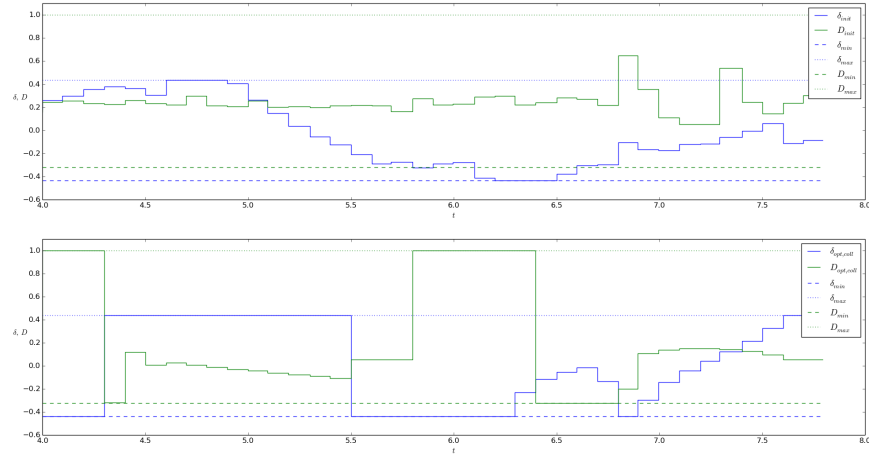


Fig. 1.5: Figure: Optimized control values in comparison to the initially used control values

The figure below shows a comparison of the simulated states values for both initially used and optimized control values, and with this, the effect of the optimization on the route of the race car and its velocity during the measurements.

The quality of the optimized experimental setup in terms of estimated standard deviations of the unknown parameters is evaluated as follows

$$p_O = \begin{pmatrix} C_{1,O} \\ C_{2,O} \\ C_{m_1,O} \\ C_{m_2,O} \\ C_{r_2,O} \\ C_{r_0,O} \end{pmatrix} = \begin{pmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{pmatrix} \pm \begin{pmatrix} 1.93054150676 \\ 0.278656552587 \\ 1.96689422255 \\ 1.51815346784 \\ 3.42713773836 \\ 1.88475684297 \end{pmatrix}$$

which indicates that the optimized setup is more appropriate for parameter estimation compared to the initial experimental design. Though, the estimated standard deviations are still relatively big in comparison to the scaled parameter values, so it would probably make sense to further refine the experimental design.

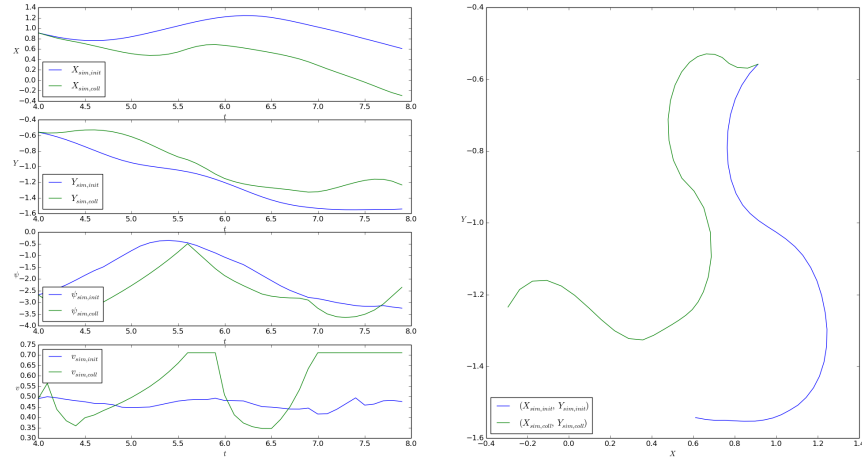


Fig. 1.6: Figure: Comparison of the simulated states values for initial and optimized controls

Further steps

Possible strategies for further refinement of the experimental design could be to increase the duration of the experiment so that more measurements can be taken, or to loosen control and state bounds to allow for greater system excitation.

In case these strategies are not applicable (physical limitations, safety concerns or alike), designing multiple experiments within one optimization problem can be a useful approach, so that several independent experiments can focus on different aspects of the system, which allows for a structured gathering of additional information about the system that can later be used within one parameter estimation.

Both planning of such experiments and using independent measurements data sets within one parameter estimation can be realized with *casiopeia* as well, see *Optimum experimental design of multiple experiments* and *Parameter estimation from multiple experiments*.

References

C

`casiopeia.doe`, [12](#)
`casiopeia.pe`, [9](#)
`casiopeia.sim`, [8](#)
`casiopeia.system`, [8](#)

C

`casiopeia.doe` (module), 12
`casiopeia.pe` (module), 9
`casiopeia.sim` (module), 8
`casiopeia.system` (module), 8
`compute_covariance_matrix()` (`casiopeia.pe.LSq`
method), 11

D

`DoE` (class in `casiopeia.doe`), 12

L

`LSq` (class in `casiopeia.pe`), 9

M

`MultiDoE` (class in `casiopeia.doe`), 14
`MultiLSq` (class in `casiopeia.pe`), 12

P

`plot_confidence_ellipsoids()` (`casiopeia.doe.DoE`
method), 14
`print_estimation_results()` (`casiopeia.pe.LSq` method), 11

R

`run_parameter_estimation()` (`casiopeia.pe.LSq` method),
11
`run_system_simulation()` (`casiopeia.sim.Simulation`
method), 9

S

`Simulation` (class in `casiopeia.sim`), 8
`System` (class in `casiopeia.system`), 8