casiopeia

Release 0.2.1

Mar 11, 2020

Contents

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 makes use of the optimization framework CasADi. For casiopeia to work, you need CasADi version = 3.1.0 to be installed on your system, otherwise the installation of casiopeia will abort or not might work as expected. Also, casiopeia is available only for Python 2.7.

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(dot)buerger(at)hs-karlsruhe(dot)de.

CHAPTER 1

Contents

1.1 Get and install casiopeia

The following instructions have been tested on Ubuntu 18.04. If you are planning to install casiopeia on Linux systems different from Ubuntu 18.04, these commands need to be adapted accordingly.

1.1.1 Installation on Ubuntu 18.04

casiopeia is available only for Python 2.7. Also, the version of CasADi required by casiopeia requires the installation of `libgfortran3`. You might need root priviliges to run the following commands:

```
apt install libgfortran3
```

Note: Some functionalitites of CasADi might required other additional open-source libraries to be installed. For a list of possibly 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.

Depending on your setup, you might need root privileges to install the following packages via pip:

```
pip install numpy==1.8 casadi==3.1.0
```

casiopeia can then also be obtained via pip by running the command

```
pip install casiopeia
```

or by cloning the corresponding repository and installing casiopeia by

```
git clone https://github.com/adbuerger/casiopeia.git
cd casiopeia
python setup.py install
```

To run the examples shipped with casiopeia, `pylab` is required, which can be made available by installing the following packages in addition to `numpy`:

```
pip install scipy==0.13 matplotlib==2.2.4
```

1.1.2 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 casiopeia 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 `casiopeia.System` object.

class `casiopeia.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 systems within casiopeia.

Raises TypeError, NotImplementedError

Parameters

- **u** (*casadi.casadi.MX*) – time-varying controls $u \in \mathbb{R}^{\mathrm{n_u}}$ 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 $q \in \mathbb{R}^{\mathrm{n_q}}$ that are constant over time, e.g. initial mass concentrations of reactants, elevation angles (optional)
- **p** (*casadi.casadi.MX*) – unknown parameters $p \in \mathbb{R}^{\mathrm{n_p}}$
- **x** (*casadi.casadi.MX*) – differential states $x \in \mathbb{R}^{\mathrm{n_x}}$ (optional)
- **eps_u** (*casadi.casadi.MX*) – input errors $\epsilon_u \in \mathbb{R}^{\mathrm{n}_{\epsilon_\mathrm{u}}}$ (optional)
- **phi** (*casadi.casadi.MX*) – output function $\phi(u, q, x, p) = y \in \mathbb{R}^{\mathrm{n_y}}$
- **f** (*casadi.casadi.MX*) – explicit system of ODEs $f(u, q, x, p, \epsilon_\mathrm{u}) = \dot{x} \in \mathbb{R}^{\mathrm{n_x}}$ (optional)
- **g** (*casadi.casadi.MX*) – equality constraints $g(u, q, p) = 0 \in \mathbb{R}^{\mathrm{n_g}}$ (optional)

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

Non-dynamic system (x = None):

$$y = \phi(u, q, p)$$
$$0 = g(u, q, p).$$

Explicit ODE system (x != None):

$$y = \phi(u, q, x, p)$$
$$\dot{x} = f(u, q, x, p, \epsilon_\mathrm{u}).$$

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

1.3 System simulation

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

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

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

Parameters

- **`system`** (`casiopeia.system.System`) – system considered for simulation, specified using the *`casiopeia.system.System`* class
- **`pdata`** (*`numpy.ndarray, casadi.DMatrix`*) – values of the time-constant parameters $p \in \mathbb{R}^{\mathrm{n_p}}$
- **`qdata`** (*`numpy.ndarray, casadi.DMatrix`*) – optional, values of the time-constant controls $q \in \mathbb{R}^{\mathrm{n_q}}$; 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 $x_0 \in \mathbb{R}^{\mathrm{n_x}}$ at the first time point t_0
- **`time_points`** (*`numpy.ndarray, casadi.DMatrix, list`*) – switching time points for the controls $t_{\mathrm{N}} \in \mathbb{R}^{\mathrm{N}}$
- **`udata`** (*`numpy.ndarray, casadi.DMatrix`*) – optional, values for the time-varying controls at the first $N-1$ switching time points $u_{\mathrm{N}} \in \mathbb{R}^{\mathrm{n_u} \times \mathrm{N}-1}$; 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 `casiopeia.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 casiopeia.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 *casiopeia.pe.LSq* is used to set up least squares parameter estimation problems for systems defined with the *casiopeia.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 $t_\mathrm{N} \in \mathbb{R}^\mathrm{N}$ 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 $u_\mathrm{N} \in \mathbb{R}^{\mathrm{n_u} \times \mathrm{N}-1}$ that can change at the switching time points; if no values are given, 0 will be used; note that the 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 $q_\mathrm{N} \in \mathbb{R}^{\mathrm{n_q}}$; if not values are given, 0 will be used
- **ydata** (*numpy.ndarray, casadi.DMatrix*) – values for the measurements at the switching time points $y_\mathrm{N} \in \mathbb{R}^{\mathrm{n_y} \times \mathrm{N}}$
- **wv** (*numpy.ndarray, casadi.DMatrix*) – weightings for the measurements $w_\mathrm{v} \in \mathbb{R}^{\mathrm{n_y} \times \mathrm{N}}$
- **weps_u** (*numpy.ndarray, casadi.DMatrix*) – weightings for the input errors $w_{\epsilon_\mathrm{u}} \in \mathbb{R}^{\mathrm{n}_{\epsilon_\mathrm{u}}}$ (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 $p_\mathrm{init} \in \mathbb{R}^{\mathrm{n_p}}$; 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 $x_\mathrm{init} \in \mathbb{R}^{\mathrm{n_x} \times \mathrm{N}}$; 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:

$$\begin{aligned}
\arg \min_{p,x,v,\epsilon_\mathrm{u}} \quad & \frac{1}{2}\|R(\cdot)\|_2^2 \\
\text{subject to:} \quad & v_\mathrm{k} + y_\mathrm{k} - \phi(x_\mathrm{k}, p; u_\mathrm{k}, q) = 0 \qquad k = 1, \dots, N \\
& g(x, p, \epsilon_\mathrm{u}; u, q) = 0 \\
\text{with:} \quad & \begin{pmatrix} {w_\mathrm{v}}^T & {w_{\epsilon_\mathrm{u}}}^T \end{pmatrix}^{\nVdash / \nvDash} \begin{pmatrix} v \\ \epsilon_\mathrm{u} \end{pmatrix} = R
\end{aligned}$$

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 $\Sigma_\mathrm{\hat{p}}$ can afterwards be accessed via the class attribute `LSq.covariance_matrix`, and the contained standard deviations $\sigma_\mathrm{\hat{p}}$ 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 $\sigma_\mathrm{\hat{p}}$, (the values of the standard deviations are presented only if the covariance matrix had already been computed),
- the values of the covariance matrix $\Sigma_\mathrm{\hat{p}}$ 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`.

[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 Prozanto, Luc: Identification of Parametric Models from Experimental Data, Springer, 1997, pages 288/289.*

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 optimization problem. You have in any case to verify your results, e. g. by simulation using the casiopeia `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 $t_\mathrm{N} \in \mathbb{R}^\mathrm{N}$ 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 $u_\mathrm{min} \in \mathbb{R}^{\mathrm{n_u}}$; if not values are given, $-\infty$ will be used
- **umax** (*numpy.ndarray, casadi.DMatrix*) – optional, upper bounds of the time-vaying controls $u_\mathrm{max} \in \mathbb{R}^{\mathrm{n_u}}$; if not values are given, ∞ will be used
- **uinit** (*numpy.ndarray, casadi.DMatrix*) – optional, initial guess for the values of the time-varying controls $u_\mathrm{N} \in \mathbb{R}^{\mathrm{n_u} \times \mathrm{N}-1}$ 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 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 $q_\mathrm{min} \in \mathbb{R}^{\mathrm{n_q}}$; if not values are given, $-\infty$ will be used
- **qmax** (*numpy.ndarray, casadi.DMatrix*) – optional, upper bounds of the time-constant controls $q_\mathrm{max} \in \mathbb{R}^{\mathrm{n_q}}$; 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 $q_\mathrm{init} \in \mathbb{R}^{\mathrm{n_q}}$; 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 $p \in \mathbb{R}^{\mathrm{n_p}}$
- **x0** (*numpy.ndarray, casadi.DMatrix, list*) – state values $x_0 \in \mathbb{R}^{\mathrm{n_x}}$ at the first time point t_0
- **xmin** (*numpy.ndarray, casadi.DMatrix*) – optional, lower bounds of the states $x_\mathrm{min} \in \mathbb{R}^{\mathrm{n_x}}$; if no value is given, $-\infty$ will be used
- **xmax** (*numpy.ndarray, casadi.DMatrix*) – optional, lower bounds of the states $x_\mathrm{max} \in \mathbb{R}^{\mathrm{n_x}}$; if no value is given, ∞ will be used
- **wv** (*numpy.ndarray, casadi.DMatrix*) – weightings for the measurements $w_\mathrm{v} \in \mathbb{R}^{\mathrm{n_y} \times \mathrm{N}}$
- **weps_u** (*numpy.ndarray, casadi.DMatrix*) – weightings for the input errors $w_{\epsilon_\mathrm{u}} \in \mathbb{R}^{\mathrm{n}_{\epsilon_\mathrm{u}}}$ (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 $I_\mathrm{X}(\cdot)$ to be used on the covariance matrix, possible values are *A* (default) and *D*, while

$$I_\mathrm{A}(\Sigma_\mathrm{p}) = \frac{1}{n_\mathrm{p}} \mathrm{Tr}(\Sigma_\mathrm{p}),$$
$$I_\mathrm{D}(\Sigma_\mathrm{p}) = \left|\Sigma_\mathrm{p}\right|^{\frac{1}{n_\mathrm{p}}},$$

  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:

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

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)

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:

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

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

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

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

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

[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.*

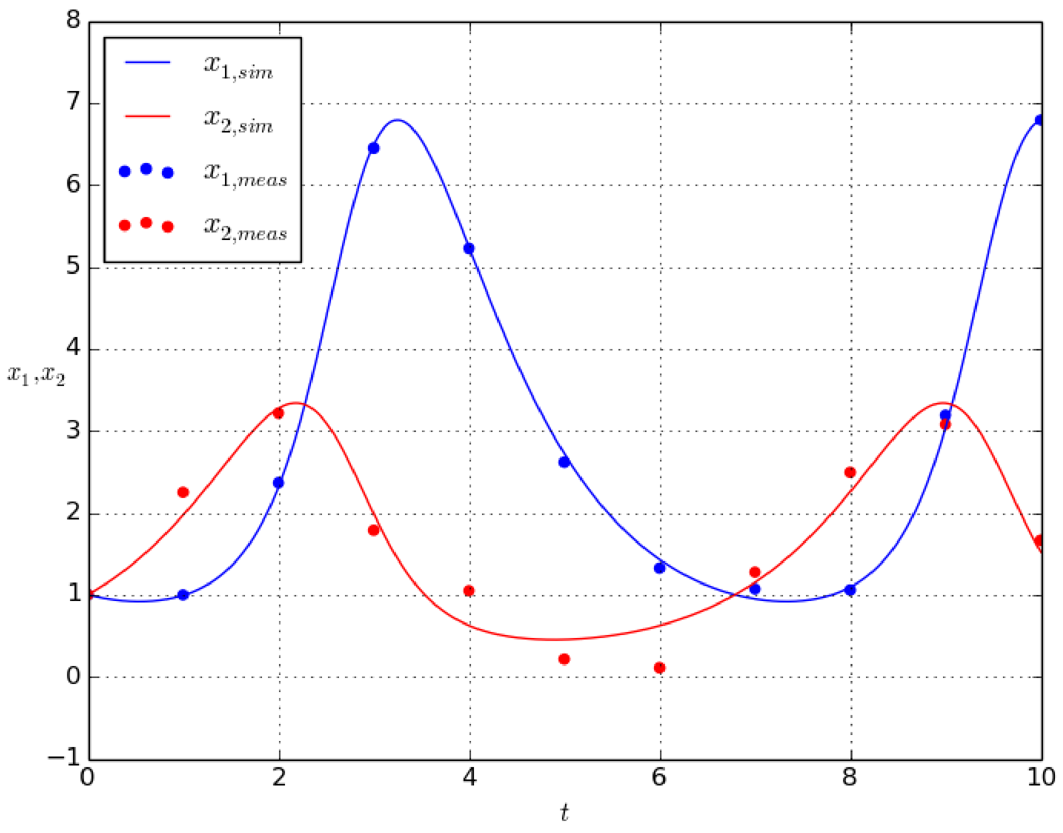

Fig. 1: Figure: Simulation results for the Lotka-Volterra predator-prey-model using the estimated parameters, compared to the given measurement data

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

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

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

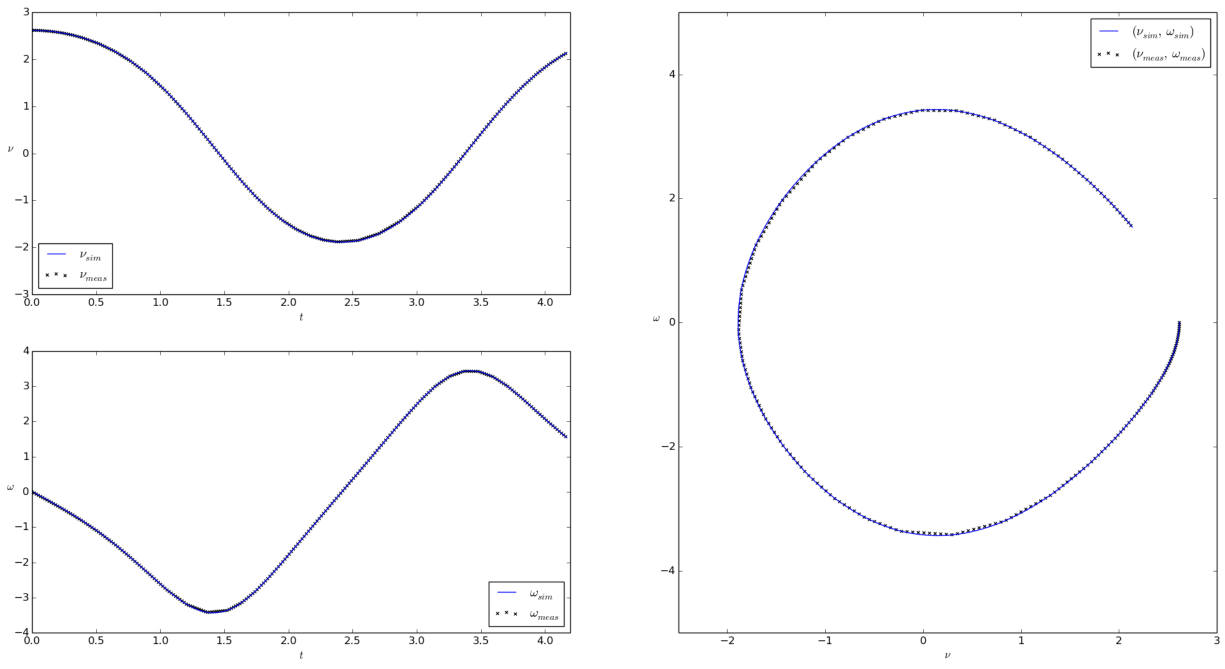

Fig. 2: Figure: Simulation results for the pedulum model using the estimated parameters, compared to the given measurement data

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 data[3]. The race car and the interpretation of the model states are shown in the figure below[4].

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

$$\begin{aligned}
\dot{X} &= v\cos(\psi + C_1\delta) \\
\dot{Y} &= v\sin(\psi + C_1\delta) \\
\dot{\psi} &= v\,\delta\,C_2 \\
\dot{v} &= C_{\mathrm{m}_1}\,D - C_{\mathrm{m}_2}\,D\,v - C_{\mathrm{r}_2}\,v^2 - C_{\mathrm{r}_0} - (v\,\delta)^2\,C_2\,C_1,
\end{aligned}$$

[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.*

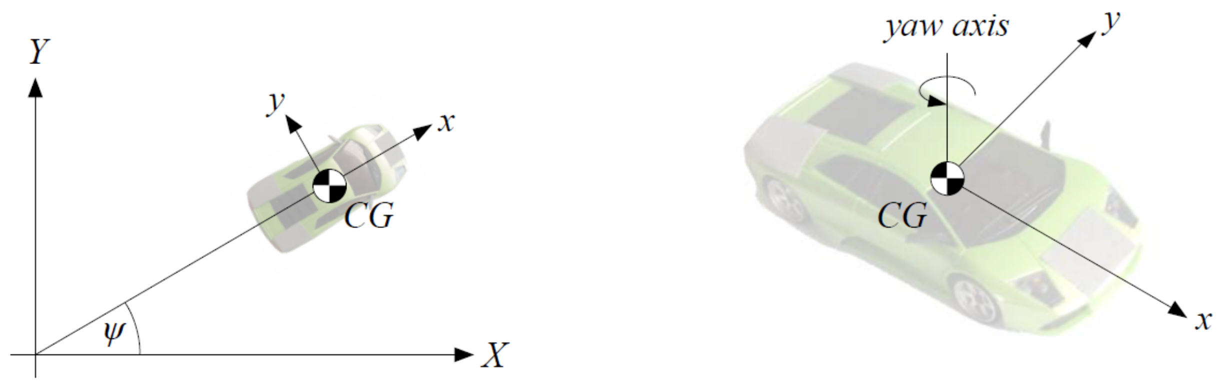

Fig. 3: Figure: Depiction of the race car showing the models states

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

$$x = \begin{pmatrix} X \\ Y \\ \psi \\ v \end{pmatrix}, \; u = \begin{pmatrix} \delta \\ D \end{pmatrix}, \; p = \begin{pmatrix} C_1 \\ C_2 \\ C_{\mathrm{m}_1} \\ C_{\mathrm{m}_2} \\ C_{\mathrm{r}_2} \\ C_{\mathrm{r}_0} \end{pmatrix},$$

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

$$\phi = x.$$

The values resulting from the parameter estimation are

$$\hat{p} = \begin{pmatrix} \hat{C_1} \\ \hat{C_2} \\ \hat{C_{\mathrm{m}_1}} \\ \hat{C_{\mathrm{m}_2}} \\ \hat{C_{\mathrm{r}_2}} \\ \hat{C_{\mathrm{r}_0}} \end{pmatrix} = \begin{pmatrix} 0.273408 \\ 11.5602 \\ 2.45652 \\ 7.90959 \\ -0.44353 \\ -0.249098 \end{pmatrix}.$$

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 relatively big.

$$\hat{p} = \begin{pmatrix} \hat{C_1} \\ \hat{C_2} \\ \hat{C_{\mathrm{m}_1}} \\ \hat{C_{\mathrm{m}_2}} \\ \hat{C_{\mathrm{r}_2}} \\ \hat{C_{\mathrm{r}_0}} \end{pmatrix} = \begin{pmatrix} 0.273408 \\ 11.5602 \\ 2.45652 \\ 7.90959 \\ -0.44353 \\ -0.249098 \end{pmatrix} \pm \begin{pmatrix} 0.034497452 \\ 0.058569592 \\ 2.72097859 \\ 5.448817078 \\ 1.478999406 \\ 0.37343932 \end{pmatrix}$$

This intends that the estimation results for the parameters $\hat{C_{\mathrm{m}_1}}$, $\hat{C_{\mathrm{m}_2}}$, $\hat{C_{\mathrm{r}_2}}$ and $\hat{C_{\mathrm{r}_0}}$ 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.

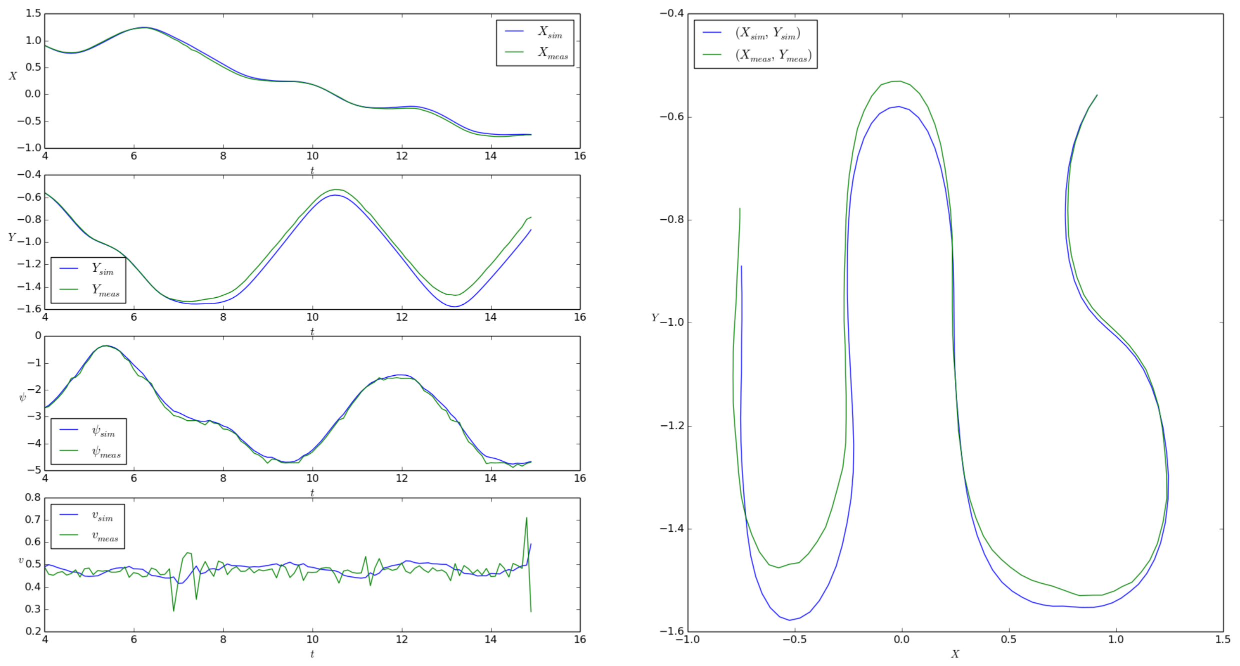

Fig. 4: Figure: Simulation results for the race car model using the estimated parameters, compared to the given measurement data

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

$$p_{\mathrm{I}} = \begin{pmatrix} C_{1,\mathrm{I}} \\ C_{2,\mathrm{I}} \\ C_{\mathrm{m}_1,\mathrm{I}} \\ C_{\mathrm{m}_2,\mathrm{I}} \\ C_{\mathrm{r}_2,\mathrm{I}} \\ C_{\mathrm{r}_0,\mathrm{I}} \end{pmatrix} = \begin{pmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{pmatrix} \pm \begin{pmatrix} 6.1591763006 \\ 0.318683714861 \\ 92.0037213296 \\ 62.6460661875 \\ 286.556042737 \\ 108.733245939 \end{pmatrix}$$

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.

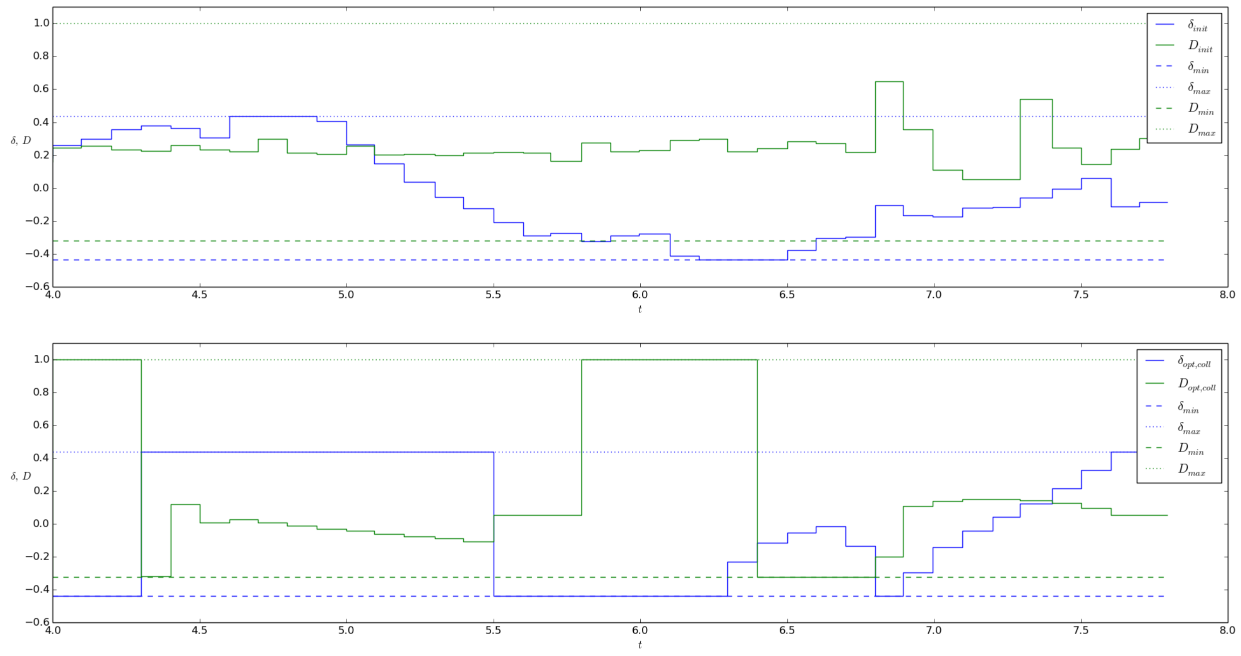

Fig. 5: Figure: Optimized control values in comparison to the initially used control values

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 it's 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

$$p_{\mathrm{O}} = \begin{pmatrix} C_{1,\mathrm{O}} \\ C_{2,\mathrm{O}} \\ C_{\mathrm{m}_1,\mathrm{O}} \\ C_{\mathrm{m}_2,\mathrm{O}} \\ C_{\mathrm{r}_2,\mathrm{O}} \\ C_{\mathrm{r}_0,\mathrm{O}} \end{pmatrix} = \begin{pmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{pmatrix} \pm \begin{pmatrix} 1.93054150676 \\ 0.278656552587 \\ 1.96689422255 \\ 1.51815346784 \\ 3.42713773836 \\ 1.88475684297 \end{pmatrix}$$

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.

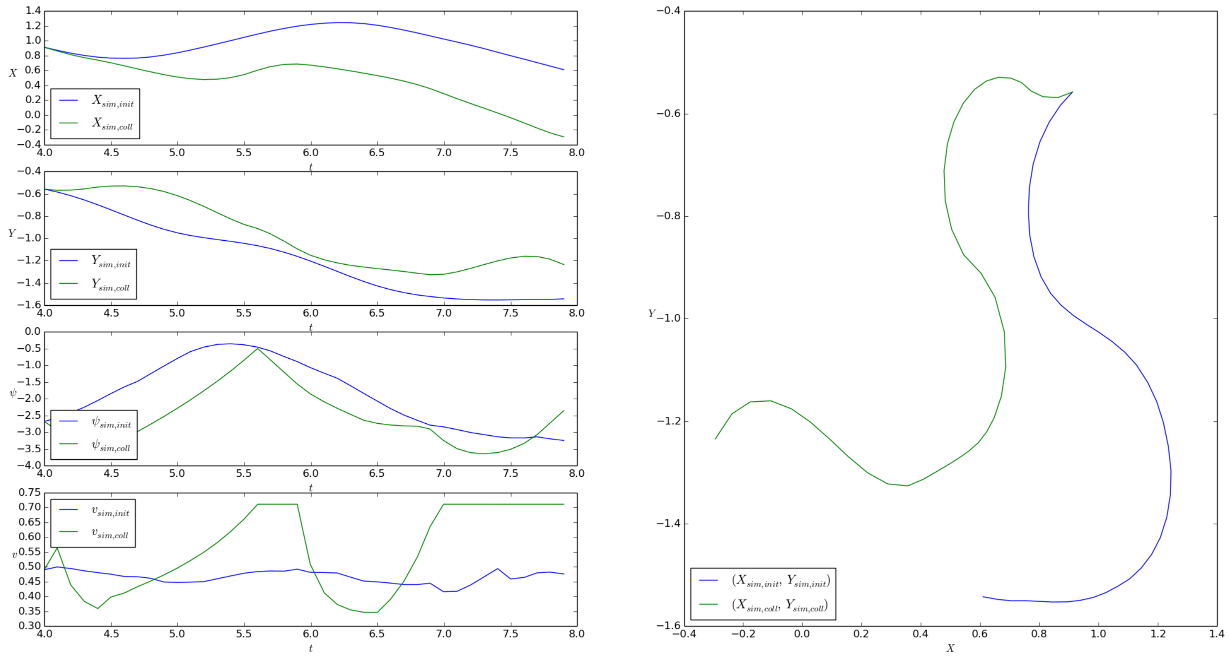

Fig. 6: Figure: Comparison of the simulated states values for initial and optimized controls

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

Python Module Index

Index