YAOCPTool Documentation

Release 0.2

Marco Aurelio S. de Aguiar

Oct 15, 2018

CONTENTS

CHAPTER

ONE

YAOCPTOOL PACKAGE

1.1 Subpackages

1.1.1 yaocptool.estimation package

1.1.1.1 Submodules

1.1.1.2 yaocptool.estimation.estimator_abstract module

class yaocptool.estimation.estimator_abstract.**EstimatorAbstract**(***kwargs*)

__init__(***kwargs*)

estimate(*t_k*, *y_k*, *u_k*)
Estimate the state given the measurement y_k and the control u_k :param DM t_k: time of the measurements :param DM y_k: measurement :param DM u_k: control

1.1.1.3 yaocptool.estimation.extended_kalman_filter module

class yaocptool.estimation.extended_kalman_filter.**ExtendedKalmanFilter**(*model*, ***kwargs*)

Bases: *yaocptool.estimation.estimator_abstract.EstimatorAbstract*

__init__(*model*, ***kwargs*)
Extended Kalman Filter for ODE and DAE systems implementation based on (1)

(1) Mandela, R. K., Narasimhan, S., & Rengaswamy, R. (2009). Nonlinear State Estimation of Differential Algebraic Systems. Proceedings of the 2009 ADCHEM (Vol. 42). IFAC. http://doi.org/10.3182/20090712-4-TR-2008.00129

Parameters

- **model** (SystemModel) – filter model
- **t_s** (*float*) – sampling time
- **t** (*float*) – current estimator time
- **x_mean** (*DM*) – current state estimation
- **p_k** (*DM*) – current covariance estimation
- **r_v** (*DM*) – process noise covariance matrix
- **r_n** (*DM*) – measurement noise covariance matrix

- **y_guess** (*DM*) – initial guess of the algebraic variables for the model simulation

estimate (*t_k*, *y_k*, *u_k*)

n_meas

Number of measurements

Return type int

Returns Number of measurements

1.1.1.4 yaocptool.estimation.pce_kalman_filter module

class yaocptool.estimation.pce_kalman_filter.**PCEKalmanFilter**(*model*, ***kwargs*)

Bases: *yaocptool.estimation.estimator_abstract.EstimatorAbstract*

__init__(*model*, ***kwargs*)

Parameters

- **model** (SystemModel) – estimator model
- **x_mean** – a initial guess for the mean
- **p_k** – a initial guess for the covariance of the estimator
- **h_function** – a function that receives 3 parameters (x, y_algebraic, u) and returns an measurement.
- **c_matrix** – if h_function is not give, c_matrix is "C" from the measurement equation: y_meas = C*[x y_alg] + D*u. note that it has to have n_x + n_y (algebraic) columns.
- **r_v** (*DM*) – process noise matrix
- **r_n** (*DM*) – measurement noise matrix
- **pc_order** (*int*) – Polynomial chaos order
- **n_samples** (*int*) – Number of samples to be used

estimate (*t_k*, *y_k*, *u_k*)

n_pol_parameters

1.1.1.5 yaocptool.estimation.unscented_kalman_filter module

class yaocptool.estimation.unscented_kalman_filter.**UnscentedKalmanFilter**(*model*, ***kwargs*)

Bases: *yaocptool.estimation.estimator_abstract.EstimatorAbstract*

__init__(*model*, ***kwargs*)

Unscented Kalman Filter. Two versions are implemented standard and square-root. Implemented based on [1] and [2].

References:

[1] Wan, E. A., & Van Der Merwe, R. (2000). The unscented Kalman filter for nonlinear estimation. In Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium (Cat. No.00EX373) (Vol. v, pp. 153–158). IEEE. http://doi.org/10.1109/ASSPCC.2000.882463

[2] Merwe, R. Van Der, & Wan, E. a. (2001). The square-root unscented Kalman filter for state and parameter-estimation. 2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.01CH37221), 6, 1–4. http://doi.org/10.1109/ICASSP.2001.940586

Parameters

- **model** (SystemModel) – estimator model
- **x_mean** – a initial guess for the mean
- **p_k** – a initial guess for the covariance of the estimator
- **h_function** – a function that receives 3 parameters (x, y_algebraic, u) and returns an measurement.
- **c_matrix** – if h_function is not give, c_matrix is "C" from the measurement equation: y_meas = C*[x y_alg] + D*u. note that it has to have n_x + n_y (algebraic) columns.
- **r_v** (*DM*) – process noise matrix
- **r_n** (*DM*) – measurement noise matrix
- **implementation** – options: 'standard' or 'square-root'. (default: 'standard')

static cholupdate(*r_matrix, x, sign*)

estimate(*t_k, y_k, u_k*)

n_meas

Number of measurements

Return type int

Returns Number of measurements

1.1.1.6 Module contents

1.1.2 yaocptool.methods package

1.1.2.1 Subpackages

yaocptool.methods.base package

Submodules

yaocptool.methods.base.discretizationschemebase module

class yaocptool.methods.base.discretizationschemebase.**DiscretizationSchemeBase**(*solution_method*

__init__(*solution_method*)

Base class for discretization methods. A discretization class transforms and OCP into a NLP

create_initial_guess(*p=None, theta=None*)

Create an initial guess for the optimal control problem using problem.x_0, problem.y_guess, problem.u_guess, and a given p and theta (for p_opt and theta_opt) if they are given. If y_guess or u_guess are None the initial guess uses a vector of zeros of appropriate size. If no p or theta is given, an vector of zeros o appropriate size is used.

Parameters

- **p** – Optimization parameters
- **theta** – Optimization theta

Returns

create_initial_guess_with_simulation (*u=None*, *p=None*, *theta=None*)

Create an initial guess for the optimal control problem using by simulating with a given control u, and a given p and theta (for p_opt and theta_opt) if they are given. If no u is given the value of problem.u_guess is used, or problem.u_past, then a vector of zeros of appropriate size is used. If no p or theta is given, an vector of zeros o appropriate size is used.

Parameters

- **u** – Control initial guess
- **p** – Optimization parameters
- **theta** – Optimization theta

Returns

create_nlp_symbolic_variables (*nlp*)

Create the symbolic variables that will be used by the NLP problem :rtype: (DM, List[List[DM]], List(List(DM)), List(DM), DM, DM, DM)

degree

degree_control

delta_t

discretize (*x_0=None*, *p=None*, *theta=None*, *last_u=None*)

Discretize the OCP, returning a Optimization Problem

Parameters

- **x_0** – initial condition
- **p** – parameters
- **theta** – theta parameters
- **last_u** – last applied control

Returns

finite_elements

get_system_at_given_times (*x*, *y*, *u*, *time_dict=None*, *p=None*, *theta=None*, *functions=None*, *start_at_t_0=False*)

model

Return type *SystemModel*

problem

Return type *OptimalControlProblem*

set_data_to_optimization_result_from_raw_data (*optimization_result*, *raw_solution_dict*)

Set the raw data received from the solver and put it in the Optimization Result object :type optimization_result: yaocptool.methods.optimizationresult.OptimizationResult :type raw_solution_dict: dict

split_x_and_u (*results_vector*, *all_subinterval=False*)

split_x_y_and_u (*v*, *all_subinterval=False*)

time_breakpoints

unpack_decision_variables (*decision_variables*)

Return a structured data from the decision variables vector

Returns: (x_data, y_data, u_data, p_opt, eta)

Parameters **decision_variables** – DM

Returns tuple

vectorize (*vector*)

yaocptool.methods.base.optimizationresult module

class yaocptool.methods.base.optimizationresult.**OptimizationResult** (***kwargs*)

__init__ (***kwargs*)

dataset

first_control ()

Return the first element of the control vector

Return type DM

is_collocation

is_valid

plot (*plot_list*, *figures=None*, *show=True*)

Plot the optimization result. It takes as input a list of dictionaries, each dictionary represents a plot. In the dictionary use keyword 'x' to specify which states you want to print, the value of the dictionary should be a list of state to be printed. The keywords that are accepted are: 'x', 'y', 'u' :param list plot_list: List of dictionaries to generate the plots. :param list figures: OPTIONAL: list of figures to be plotted in. If not provided it will create new figures. :param bool show: OPTIONAL: select if matplotlib.pyplot.show should be applied after the plots.

to_dataset ()

Return a dataset with the data of x, y, and u

Return type *DataSet*

yaocptool.methods.base.solutionmethodsbase module

class yaocptool.methods.base.solutionmethodsbase.**SolutionMethodsBase** (*problem*, ***kwargs*)

Bases: object

__init__ (*problem*, ***kwargs*)

Parameters

- **problem** (OptimalControlProblem) –
- **integrator_type** (*str*) – str
- **solution_method** (*str*) – str

- **degree** (*int*) – discretization polynomial degree
- **degree_control** (*int*) –
- **discretization_scheme** (*str*) – ('multiple-shooting' | 'collocation')
- **initial_guess_heuristic** (*str*) – 'simulation' or 'problem_info'

call_solver (*initial_guess=None*, *p=None*, *theta=None*, *x_0=None*, *last_u=None*, *initial_guess_dict=None*)

static collocation_points (*degree*, *cp='radau'*, *with_zero=False*)

create_control_approximation ()

Parametrize the control variable, accordingly to the 'degree_control' attribute. If degree_control == 1, then a piecewise constant control will be used (most common). If degree_control > 1, then a piecewise polynomial approximation will be used with order 'degree_control'.

Returns

create_optimization_problem ()

create_optimization_result (*raw_solution_dict*, *p=None*, *theta=None*, *x_0=None*)

create_variable_polynomial_approximation (*size*, *degree*, *name='var_appr'*, *tau=None*, *point_at_t0=False*)

delta_t

model

prepare ()

solve (*initial_guess=None*, *p=None*, *theta=None*, *x_0=None*, *last_u=None*, *initial_guess_dict=None*)

Parameters

- **initial_guess** – Initial guess
- **p** – Parameters values
- **theta** – Theta values
- **x_0** – Initial condition value
- **last_u** – Last control value
- **initial_guess_dict** – Initial guess as dict

Returns OptimizationResult

split_x_and_u

split_x_y_and_u

time_breakpoints

unvec (*vector*, *degree=None*)

Unvectorize 'vector' a vectorized matrix, assuming that it was a matrix with 'degree' number of columns :type vector: DM a vector (flattened matrix) :type degree: int

Module contents

yaocptool.methods.classic package

Submodules

yaocptool.methods.classic.collocationscheme module

Created on Thu Jul 13 17:08:34 2017

@author: marco

class yaocptool.methods.classic.collocationscheme.**CollocationScheme**(*solution_method*)

Bases: *yaocptool.methods.base.discretizationschemebase.DiscretizationSchemeBase*

create_initial_guess(*p=None*, *theta=None*)

Create an initial guess for the optimal control problem using problem.x_0, problem.y_guess, problem.u_guess, and a given p and theta (for p_opt and theta_opt) if they are given. If y_guess or u_guess are None the initial guess uses a vector of zeros of appropriate size.

Parameters

- **p** – Optimization parameters
- **theta** – Optimization theta

Returns

create_initial_guess_with_simulation(*u=None*, *p=None*, *theta=None*)

Create an initial guess for the optimal control problem using by simulating with a given control u, and a given p and theta (for p_opt and theta_opt) if they are given. If no u is given the value of problem.u_guess is used, or problem.last_u, then a vector of zeros of appropriate size is used. If no p or theta is given, an vector of zeros of appropriate size is used.

Parameters

- **u** –
- **p** – Optimization parameters
- **theta** – Optimization theta

Returns

create_nlp_symbolic_variables(*nlp*)

Create the symbolic variables that will be used by the NLP problem :param NonlinearOptimizationProblem nlp: nonlinear optimization problem in which the variables will be created :rtype: tuple

discretize(*x_0=None*, *p=None*, *theta=None*, *last_u=None*)

Discretize the OCP, returning a Optimization Problem

Parameters

- **x_0** – initial condition
- **p** – parameters
- **theta** – theta parameters
- **last_u** – last applied control

Return type *NonlinearOptimizationProblem*

get_system_at_given_times(*x*, *y*, *u*, *time_dict=None*, *p=None*, *theta=None*, *functions=None*, *start_at_t_0=False*)

Parameters

- **x** (*list*) –
- **y** (*list*) –
- **u** (*list*) –
- **time_dict** (*dict*) – Dictionary of simulations times, where the KEY is the finite_element and the VALUE list a list of desired times example : {1:{'t_0': 0.0, 'x':[0.0, 0.1, 0.2], y:[0.2]}}
- **p** – list
- **theta** – dict
- **start_at_t_0** – bool If TRUE the simulations in each finite_element will start at the element t_0, Otherwise the simulation will start the end of the previous element
- **functions** (*dict*) – Dictionary of Functions to be evaluated, KEY is the function identifier, VALUE is a CasADi Function with model.all_sym as input

set_data_to_optimization_result_from_raw_data(*optimization_result*, *raw_solution_dict*)

Set the raw data received from the solver and put it in the Optimization Result object :type optimization_result: yaocptool.methods.base.optimizationresult.OptimizationResult :type raw_solution_dict: dict

split_x_y_and_u(*decision_variables*, *all_subinterval=False*)

Parameters

- **all_subinterval** – Bool 'Returns all elements of the subinterval (or only the first one)'
- **decision_variables** – DM

Returns X, Y, and U -> list with a DM for each element

time_interpolation

time_interpolation_algebraics

time_interpolation_controls

time_interpolation_states

unpack_decision_variables(*decision_variables*, *all_subinterval=True*)

Return a structured data from the decision variables vector

Returns: (x_data, y_data, u_data, p_opt, eta)

Parameters

- **decision_variables** – DM
- **all_subinterval** – bool

Returns tuple

yaocptool.methods.classic.directmethod module

Created on Fri Oct 21 16:40:15 2016

@author: marco

class yaocptool.methods.classic.directmethod.**DirectMethod**(*problem*, ***kwargs*)

Bases: *yaocptool.methods.base.solutionmethodsbase.SolutionMethodsBase*

__init__(*problem*, ***kwargs*)

Parameters

- **problem** – yaocptool.modelling.ocp.OptimalControlProblem
- **integrator_type** – str
- **solution_method** – str
- **degree** – int
- **discretization_scheme** – str 'multiple-shooting' | 'collocation'

prepare()

yaocptool.methods.classic.indirectmethod module

Created on Fri Oct 21 16:39:52 2016

@author: marco

class yaocptool.methods.classic.indirectmethod.**IndirectMethod**(*problem*, ***kwargs*)

Bases: *yaocptool.methods.base.solutionmethodsbase.SolutionMethodsBase*

__init__(*problem*, ***kwargs*)

Parameters

- **problem** – yaocptool.modelling.ocp.OptimalControlProblem
- **integrator_type** – str
- **solution_method** – str
- **degree** – int
- **discretization_scheme** – str 'multiple-shooting' | 'collocation'

calculate_optimal_control()

include_adjoint_states()

prepare()

replace_with_optimal_control(*u_opt*)

yaocptool.methods.classic.multipleshooting module

Created on Thu Jul 13 17:08:34 2017

@author: marco

class yaocptool.methods.classic.multipleshooting.**MultipleShootingScheme**(*solution_method*)

Bases: *yaocptool.methods.base.discretizationschemebase.DiscretizationSchemeBase*

create_initial_guess(*p=None*, *theta=None*)

Create an initial guess for the optimal control problem using problem.x_0, problem.y_guess, problem.u_guess, and a given p and theta (for p_opt and theta_opt) if they are given. If y_guess or u_guess are None the initial guess uses a vector of zeros of appropriate size.

Parameters

- **p** – Optimization parameters
- **theta** – Optimization theta

Returns

create_initial_guess_with_simulation(*u=None*, *p=None*, *theta=None*)

Create an initial guess for the optimal control problem using by simulating with a given control u, and a given p and theta (for p_opt and theta_opt) if they are given. If no u is given the value of problem.u_guess is used, or problem.last_u, then a vector of zeros of appropriate size is used. If no p or theta is given, an vector of zeros o appropriate size is used.

Parameters

- **u** – Control initial guess
- **p** – Optimization parameters
- **theta** – Optimization theta

Returns

create_nlp_symbolic_variables(*nlp*)

Create the symbolic variables that will be used by the NLP problem :rtype: (MX, List[List[MX]], List[List[MX]], List[MX], MX, MX, List[MX])

discretize(*x_0=None*, *p=None*, *theta=None*, *last_u=None*)

Discretize the OCP, returning a Optimization Problem

Parameters

- **x_0** – initial condition
- **p** – parameters
- **theta** – theta parameters
- **last_u** – last applied control

Return type *NonlinearOptimizationProblem*

get_system_at_given_times(*x_var*, *y_var*, *u_var*, *time_dict=None*, *p=None*, *theta=None*, *functions=None*, *start_at_t_0=False*)

Parameters

- **x_var** – List[List[MX]]
- **y_var** – List[List[MX]]
- **u_var** – List[List[MX]]
- **p** – list
- **theta** – dict
- **start_at_t_0** – bool If TRUE the simulations in each finite_element will start at the element t_0, Otherwise the simulation will start the end of the previous element
- **functions** – Dict[str, Function|Dict[int] dictionary of Functions to be evaluated, KEY is the function identifier, VALUE is a CasADi Function with model.all_sym as input

set_data_to_optimization_result_from_raw_data (*optimization_result*, *raw_solution_dict*)
Set the raw data received from the solver and put it in the Optimization Result object :param optimization_result: OptimizationResult :param raw_solution_dict: dict

split_x_y_and_u (*results_vector*, *all_subinterval=False*)

Parameters

- **all_subinterval** – Bool 'Returns all elements of the subinterval (or only the first one)'
- **results_vector** – DM

Returns X, Y, and U -> list with a DM for each element

unpack_decision_variables (*decision_variables*)
Return a structured data from the decision variables vector

Returns: (x_data, y_data, u_data, p_opt, eta, theta_opt)

Parameters **decision_variables** – DM

Returns tuple

Module contents

1.1.2.2 Submodules

1.1.2.3 yaocptool.methods.augmented_lagrangian module

Created on Sat Oct 22 16:53:36 2016

@author: marco

class `yaocptool.methods.augmented_lagrangian.`**AugmentedLagrangian** (*problem*, *ocp_solver_class*, *solver_options=None*, ***kwargs*)

Bases: *yaocptool.methods.base.solutionmethodsbase.SolutionMethodsBase*

For a minimization problem in the form min f(x,u) = int L(x,u) dt s.t.: dot{x} = f(x,u), g_ineq (x,u) leq 0

Transforms the problem in a sequence of solution of the problem min f(x,u) = int L(x,u) -mu sum log(-g_ineq(x,u)) dt s.t.: dot{x} = f(x,u),

__init__ (*problem*, *ocp_solver_class*, *solver_options=None*, ***kwargs*)

Parameters

- **problem** (`OptimalControlProblem`) –
- **integrator_type** (*str*) – str
- **solution_method** (*str*) – str
- **degree** (*int*) – discretization polynomial degree
- **degree_control** (*int*) –
- **discretization_scheme** (*str*) – ('multiple-shooting' | 'collocation')
- **initial_guess_heuristic** (*str*) – 'simulation' or 'problem_info'

create_nu_initial_guess (*n_r=None*)

create_optimization_result (*raw_solution_dict, p=None, theta=None, x_0=None*)

static join_nu_to_theta (*theta, nu*)

model

relaxed_alg

solve_raw (*initial_guess=None, p=None, theta=None, x_0=None, last_u=None*)

split_x_and_u

split_x_y_and_u

time_interpolation_nu

1.1.2.4 yaocptool.methods.distributed_solution module

class yaocptool.methods.distributed_solution.**DistributedSolution** (*subsystem_classes_list, parameters_list, connection_list, main_to_subsystems_list=None, **kwargs*)

__init__ (*subsystem_classes_list, parameters_list, connection_list, main_to_subsystems_list=None, **kwargs*)

initialize ()

kill ()

n_subsystems

solve (*x_0, initial_guess_dict=None*)

start ()

1.1.2.5 Module contents

1.1.3 yaocptool.modelling package

1.1.3.1 Submodules

1.1.3.2 yaocptool.modelling.dae_system module

class yaocptool.modelling.dae_system.**DAESystem** (***kwargs*)

__init__ (***kwargs*)

Parameters

- **ode** – ODE equations
- **alg** – Algebraic equations
- **x** – State variables
- **y** – Algebraic variables
- **p** – Parameters

- **t** – Time variable
- **tau** – Tau variable

convert_from_tau_to_time (*t_k*, *t_kp1*)

dae_system_dict

has_parameters

is_dae

is_ode

join (*dae_sys*)

simulate (*x_0*, *t_f*, *t_0=0*, *p=None*, *y_0=None*, *integrator_type='implicit'*, *integrator_options=None*)

substitute_variable (*old_var*, *new_var*)

type

1.1.3.3 yaocptool.modelling.dataset module

class yaocptool.modelling.dataset.**DataSet** (*name='dataset'*, ***kwargs*)

__init__ (*name='dataset'*, ***kwargs*)

Generic time dependent data storage. The data is stored in the self.data dictionary. self.data['entry_name']['time'] is a row vector self.data['entry_name']['values'] is a matrix with the same number of columns as the time vector, and rows equal to self.data['entry_name']['size']. The data can be more easily managed using create_entry, get_entry, insert_data.

Parameters

- **name** (`str`) – name of th dataset
- **plot_style** (`str`) – default plot style. plot = linear interpolation, step = piecewise constant ('plot' | 'step')
- **find_discontinuity** (`bool`) – Default: True. If True, it will try to find discontinuity on the data, and plot with gaps where data is missing/not available, instead of a line connecting all data points.
- **max_sampling_time** (`float`) – maximum expected distance between two time data. This is used to detect discontinuity on the data, and plot it separately.

create_entry (*entry*, *size*, *names=None*, *plot_style=None*)

Create an entry in the dataset

Parameters

- **entry** – entry name
- **size** – number of rows in the vector
- **names** (`list`) – name for each row, it should be a list with size 'size'. If 'names' is not given, then the name list [entry_1, entry_2, . . . , entry_size]
- **plot_style** (`str`) – ('plot' | 'step') choose if the plot will be piecewise constant (step) or a first order interpolation (plot).

extend (*other_dataset*)
Extend this DataSet with another DataSet. They don't need to be ordered, after the merging a chronological sort of the data is performed.

Parameters **other_dataset** (DataSet) –

Returns

get_copy ()

Return a copy of this dataset. The copy is not connected to the original data set, therefore changes in one of the dataset will not affect the other.

Returns

get_entry (*entry*)

Return the time and values for a given entry.

Parameters **entry** (*str*) – entry name

Returns entry time, entry value

Return type tuple

get_entry_names (*entry*)

Get list of names of an entry

Parameters **entry** –

Return type list

get_entry_size (*entry*)

Get size of an entry

Parameters **entry** –

Returns

insert_data (*entry*, *time*, *value*)

plot (*plot_list*, *figures=None*, *show=True*)
Plot DataSet information. It takes as input a list of dictionaries, each dictionary represents a plot. In the dictionary use keyword 'x' to specify which states you want to print, the value of the dictionary should be a list of index of the states to be printed.

Parameters

- **plot_list** (*list*) – List of dictionaries to generate the plots.
- **figures** (*list*) – list of figures to be plotted on top (optional)
- **show** (*bool*) – if the plotted figures should be shown after plotting (optional, default=True).

sort (*entries=None*)

Sort the dataset for given 'entries' in an chronological order, this can be used when data is not inserted in an ordered fashion.

Parameters **entries** (*list*) – list of entries to be sorted, if this parameter is no given all the entries will be sorted.

1.1.3.4 yaocptool.modelling.network module

Created on Fri Jul 07 16:05:50 2017

@author: marco

class yaocptool.modelling.network.**Connection**(*connection_id*, *nodes=None*, *z_sym_indices_list=None*)

__init__(*connection_id*, *nodes=None*, *z_sym_indices_list=None*)

Connection class for Network systems

class yaocptool.modelling.network.**Network**(*nodes*, *connections_settings_dict*)

__init__(*nodes*, *connections_settings_dict*)

Example of connections_dict: Connections 0: z_sym[0] in Node 0 is connected to z_sym[2] in Node 1 connections_dict = {0: [{0:[0]}, {1:[2]}]}

it can also be a multidimensional connection: Connection 1: z_sym[0:1] in Node 1 is connected to z_sym[1:2] in Node 4 connections_dict = {1: [{1:[0,1]}, {4:[1,2]}]}

create_connections()

get_connection_defined_z()

get_connection_equations()

models

problems

1.1.3.5 yaocptool.modelling.node module

Created on Fri Jul 07 16:33:52 2017

@author: marco

class yaocptool.modelling.node.**Node**(*node_id=-1*, *name="*, *model=None*, *problem=None*, ***kwargs*)

__init__(*node_id=-1*, *name="*, *model=None*, *problem=None*, ***kwargs*)

1.1.3.6 yaocptool.modelling.ocp module

Created on Mon Apr 03 11:15:03 2017

@author: marco

class yaocptool.modelling.ocp.**OptimalControlProblem**(*model*, ***kwargs*)

Optimal Control Problem class, used to define a optimal control problem based on a model (SystemModel) It has the following form:

$$\min J = V(x(t_f), p) + \int_{t_0}^{t_f} L(x, y, u, t, p, \theta) + \sum_k S(x(t_k), y(t_k), u(t_k), t_k, p, \theta_k)$$

$$\begin{aligned}
\text{s.t.: } & \dot{x} = f(x, y, u, t, p, \theta) \\
& g(x, y, u, t, p, \theta) = 0 \\
& g_e q(x, y, u, t, p, \theta) = 0 \\
& g_i neq(x, y, u, t, p, \theta) \leq 0 \\
& x_{min} \leq x \leq x_{max} \\
& y_{min} \leq y \leq y_{max} \\
& u_{min} \leq u \leq u_{max} \\
& \Delta u_{min} \leq \Delta u \leq \Delta u_{max} \\
& h_{initial}(x_(t_0), t_0, p, heta) = 0 \\
& h_{final}(x_(t_f), t_f, p, heta) = 0 \\
& h(p) = 0
\end{aligned}$$

where t_k is final time in each finite element.

__init__(*model*, ***kwargs*)

create_algebraic(*name*, *size*, *alg=None*, *y_max=None*, *y_min=None*, *y_guess=None*)

create_cost_state()

Transforms the integral int_{t_0}^{t_f} L(. . .) dt into a dynamic cost state: dot{x}_c = L(. . .) and include x_c(t_f) into the final time cost (V(t_f) += x_c(t_f)).

Return type object

create_optimization_parameter(*name*, *size=1*, *p_opt_min=None*, *p_opt_max=None*)

create_optimization_theta(*name*, *size=1*, *new_theta_opt_min=None*, *new_theta_opt_max=None*)

create_parameter(*name*, *size=1*)

create_quadratic_cost(*par_dict*)

create_state(*name*, *size*, *ode=None*, *x_0=None*, *var_max=None*, *var_min=None*, *h_initial=None*)

get_p_opt_indices()

get_theta_opt_indices()

has_delta_u

include_algebraic(*var*, *alg=None*, *y_min=None*, *y_max=None*, *y_guess=None*)

include_control(*var*, *u_min=None*, *u_max=None*)

include_equality(*eq*)

Include time independent equality. Equality is concatenated "h".

Parameters **eq** – time independent equality

include_final_time_equality(*eq*)

Include final time equality. Equality that is evaluated at t=t_f. The equality is concatenated to "h_final"

Parameters **eq** – final equality constraint

include_initial_time_equality(*eq*)
Include initial time equality. Equality that is evaluated at t=t_0. The equality is concatenated to "h_initial"

Parameters **eq** – initial equality constraint

include_optimization_parameter(*var*, *p_opt_min=None*, *p_opt_max=None*)

include_optimization_theta(*var*, *theta_opt_min=None*, *theta_opt_max=None*)

include_state(*var*, *ode=None*, *x_0=None*, *x_min=None*, *x_max=None*, *h_initial=None*, *x_0_sym=None*, *suppress=False*)

include_system_equations(*ode=None*, *alg=None*)

include_time_equality(*eq*, *when='default'*)
Include time dependent equality. g_eq(…, t) = 0, for t in [t_0, t_f]

The inequality is concatenated to "g_ineq"

Parameters

- **eq** – equality
- **when** (*str*) – Can be 'default', 'end', 'start'. 'start' - the constraint will be evaluated at the start of every finite element 'end' - the constraint will be evaluated at the end of every finite element 'default' - will be evaluated at each collocation point of every finite element. For the multiple shooting, the constraint will be evaluated at the end of each finite element

include_time_inequality(*eq*, *when='default'*)
Include time dependent inequality. g_ineq(…, t) <= 0, for t in [t_0, t_f]

The inequality is concatenated to "g_ineq"

Parameters

- **when** (*str*) – Can be 'default', 'end', 'start'. 'start' - the constraint will be evaluated at the start of every finite element 'end' - the constraint will be evaluated at the end of every finite element 'default' - will be evaluated at each collocation point of every finite element. For the multiple shooting, the constraint will be evaluated at the end of each finite element
- **eq** – inequality

merge(*problems*)

n_eta

n_g_eq

n_g_ineq

n_h_final

n_h_initial

n_p_opt

n_theta_opt

pre_solve_check()

remove_algebraic(*var*, *eq=None*)

remove_control(*var*)

replace_variable(*original*, *replacement*, *variable_type='other'*)

reset_working_model()

set_parameter_as_optimization_parameter(*new_p_opt*, *new_p_opt_min=None*, *new_p_opt_max=None*)

set_theta_as_optimization_theta(*new_theta_opt*, *new_theta_opt_min=None*, *new_theta_opt_max=None*)

class yaocptool.modelling.ocp.**SuperOCP**(*problems*, ***kwargs*)
Bases: *yaocptool.modelling.ocp.OptimalControlProblem*

__init__(*problems*, ***kwargs*)

1.1.3.7 yaocptool.modelling.simualtion_result module

class yaocptool.modelling.simualtion_result.**SimulationResult**(*n_x*, *n_y*, *n_u*, *x_names=None*, *y_names=None*, *u_names=None*, ***kwargs*)
Bases: *yaocptool.modelling.dataset.DataSet*

__init__(*n_x*, *n_y*, *n_u*, *x_names=None*, *y_names=None*, *u_names=None*, ***kwargs*)

extend(*other_sim_result*)
Extend this SimulationResult with other SimulationResult. It is only implemented for the case where the other simulation result starts at the end of this simulation. That is this.t_f == other_sim_result.t_0 .

Parameters **other_sim_result** (SimulationResult) –

Returns

final_condition()
Return the simulation final condition in the form of a tuple (x_f, y_f, u_f)

Return type DM, DM, DM

n_u

n_x

n_y

t

u

u_names

x

x_names

y

y_names

1.1.3.8 yaocptool.modelling.stochastic_ocp module

class yaocptool.modelling.stochastic_ocp.**StochasticOCP**(*model*, ***kwargs*)
Bases: *yaocptool.modelling.ocp.OptimalControlProblem*

__init__(*model*, ***kwargs*)

create_uncertain_parameter(*name*, *mean*, *var*, *size=1*, *distribution='normal'*)

get_p_unc_indices()

get_p_without_p_unc()

get_uncertain_initial_cond_indices()

include_time_chance_inequality(*ineq*, *prob*, *rhs=None*, *when='default'*)

Include time dependent chance inequality. Prob[ineq(. . . , t) <= rhs] <= prob, for t in [t_0, t_f]

The inequality is concatenated to "g_stochastic_ineq"

Parameters

- **ineq** – inequality
- **rhs** (*list* | *DM*) – Right-hand size of the inequality
- **prob** (*list* | *DM*) – Chance/probability of the constraint being satisfied
- **when** (*str*) – Can be 'default', 'end', 'start'. 'start' - the constraint will be evaluated at the start of every finite element 'end' - the constraint will be evaluated at the end of every finite element 'default' - will be evaluated at each collocation point of every finite element. For the multiple shooting, the constraint will be evaluated at the end of each finite element

include_uncertain_parameter(*par*, *mean*, *var*, *distribution='normal'*)

n_g_stochastic

n_p_unc

n_uncertain_initial_condition

set_initial_condition_as_uncertain(*par*, *mean*, *cov*, *distribution='normal'*)

set_parameter_as_uncertain_parameter(*par*, *mean*, *var*, *distribution='normal'*)

1.1.3.9 yaocptool.modelling.system_model module

Created on Thu Jun 09 10:50:48 2016

@author: marco

class yaocptool.modelling.system_model.**SystemModel**(*name='model'*, *n_x=0*, *n_y=0*, *n_u=0*, *n_p=0*, *n_theta=0*, ***kwargs*)

__init__(*name='model'*, *n_x=0*, *n_y=0*, *n_u=0*, *n_p=0*, *n_theta=0*, ***kwargs*)

x - states y - (internal) algebraic z - external algebraic u - control p - constant parameters theta - parameters dependent of the finite_elements u_par - parametrized control parameters

Note: when vectorizing the parameters order is [p; theta; u_par]

all_sym

connect(*u*, *y*)

Connect an input 'u' to a algebraic variable 'y', u = y. The function will perform the following actions: - include an algebraic equation u - y = 0 - remove 'u' from the input vector (since it is not a free variable anymore) - include 'u' into the algebraic vector, since it is an algebraic variable now.

Parameters

- **u** – input variable
- **y** – algebraic variable

convert_expr_from_tau_to_time(*expr*, *t_k*, *t_kp1*)

convert_expr_from_time_to_tau(*expr*, *t_k*, *t_kp1*)

create_algebraic_variable(*name='y'*, *size=1*)
Create a new algebraic variable with the name "name" and size "size". Size can be an int or a tuple (e.g. (2,2)). However, the new algebraic variable will be vectorized (casadi.vec) to be included in the algebraic vector (model.y_sym).

Parameters

- **name** (*str*) –
- **size** (*int||tuple*) –

Returns

create_control(*name='u'*, *size=1*)
Create a new control variable name "name" and size "size". Size can be an int or a tuple (e.g. (2,2)). However, the new control variable will be vectorized (casadi.vec) to be included in the control vector (model.u_sym).

Parameters

- **name** – str
- **size** – int

Returns

create_parameter(*name='p'*, *size=1*)
Create a new parameter name "name" and size "size"

Parameters

- **name** – str
- **size** – int

Returns

create_state(*name='x'*, *size=1*)
Create a new state with the name "name" and size "size". Size can be an int or a tuple (e.g. (2,2)). However, the new state will be vectorized (casadi.vec) to be included in the state vector (model.x_sym).

Parameters

- **name** – str
- **size** – int|tuple

Returns

create_theta(*name='theta'*, *size=1*)
Create a new parameter name "name" and size "size"

Parameters

- **name** – str
- **size** – int

Returns

find_algebraic_variable(*x*, *u*, *guess=None*, *t=0.0*, *p=None*, *theta_value=None*, *rootfinder_options=None*)

find_equilibrium(*additional_eqs*, *guess=None*, *t_0=0.0*, *rootfinder_options=None*)
Find a equilibrium point for the model. This method solves the root finding problem:

f(x,y,u,t_0) = 0 g(x,y,u,t_0) = 0 additional_eqs (x,y,u,t_0) = 0

Use additional_eqs to specify the additional conditions remembering that dim(additional_eqs) = n_u, so the system can be well defined. If no initial guess is provided ("guess" parameter) a guess of ones will be used (not zero to avoid problems with singularities.

Returns x_0, y_0, u_0

Parameters

- **rootfinder_options** (*dict*) – options to be passed to rootfinder
- **additional_eqs** – SX
- **guess** – DM
- **t_0** – float

Returns (DM, DM, DM)

get_copy()

Get a copy of this model

Return type *SystemModel*

get_dae_system()
Return a DAESystem object with the model equations.

Returns DAESystem

get_hardcopy()

Get a hard copy of this model, differently from "get_copy", the variables of the original copy and the hard copy will not be the same, i.e. model.x_sym != copy.x_sym

Return type *SystemModel*

include_algebraic(*var*, *alg=None*)

include_control(*var*)

include_models(*models*)

include_parameter(*p*)

include_state(*var*, *ode=None*, *x_0_sym=None*)

include_system_equations(*ode=None*, *alg=None*)

Include model equations, (ordinary) differential equation and algebraic equation (ode and alg)

Parameters

- **ode** – (ordinary) differential equation
- **alg** – algebraic equation

include_theta(*theta*)

lamb_sym

linearize (*x_bar*, *u_bar*)
Returns a linearized model at a given points (X_BAR, U_BAR)

merge (*models_list*, *connecting_equations=None*)

n_p

n_theta

n_u

n_x

n_y

p

p_names

print_variables ()
Print list of variable in the model (x, y, u, p, theta)

static put_values_in_all_sym_format (*t=None*, *x=None*, *y=None*, *u=None*, *p=None*, *theta=None*, *u_par=None*)

remove_algebraic (*var*, *eq=None*)

remove_control (*var*)

remove_parameter (*var*)

remove_theta (*var*)

replace_variable (*original*, *replacement*)
Replace a variable or parameter by an variable or expression.

Parameters

- **replacement** –
- **original** – SX: and replacement, and also variable type which describes which type of variable is being remove to it from the counters. Types: 'x', 'y', 'u', 'p', 'ignore'

simulate (*x_0*, *t_f*, *t_0=0.0*, *u=None*, *p=None*, *theta=None*, *y_0=None*, *integrator_type='implicit'*, *integrator_options=None*)

Simulate model. If t_f is a float, then only one simulation will be done. If t_f is a list of times, then a sequence of simulations will be done, that each t_f is the end of a finite element.

Parameters

- **x_0** (`list||DM`) – Initial condition
- **t_f** (`float||list`) – Final time of the simulation, can be a list of final times for sequential simulation
- **t_0** (`float`) – Initial time
- **u** (`list||DM`) – Controls of the system to be simulated
- **p** (`DM||SX||list`) – Simulation parameters
- **theta** (`dict`) – Parameters theta, which varies for each simulation for sequential simulations. If t_f is a list then theta has to have one entry for each k in [0,…,len(t_f)]
- **y_0** – Initial guess for the algebraic variables
- **integrator_type** (`str`) – 'implicit' or 'explicit'

- **integrator_options** (*dict*) – options to be passed to the integrator

Return type *SimulationResult*

system_type

theta

theta_names

u

u_names

x

x_names

x_sys_sym

y

y_names

1.1.3.10 Module contents

Created on Mon Jul 10 14:27:21 2017

@author: marco

1.1.4 yaocptool.mpc package

1.1.4.1 Submodules

1.1.4.2 yaocptool.mpc.mpc module

class yaocptool.mpc.mpc.**MPC**(*plant*, *solution_method*, ***kwargs*)

__init__(*plant*, *solution_method*, ***kwargs*)
Model Predictive control class. Requires a plant and a solution_method.

Parameters

- **plant** (*Plant|PlantSimulation*) –
- **solution_method** (SolutionMethodsBase) –
- **estimator** (EstimatorAbstract) –
- **default_p** (*DM*) – is a default parameter vector that will be used by the 'solution_method'
- **default_theta** (*DM*) – is a default theta parameter that will be used by the 'solution_method'
- **include_cost_in_state_vector** (*bool*) – Typically the optimal control problem has one extra state than the plant, the dynamic cost state. By setting this variable to True, it automatically include an zero the in state vector obtained from the estimator.

- **mean_as_parameter** (*bool*) – The mean estimated by the Estimator as parameter of the OCP. It will be put in the end of parameter vector, before the covariance if covariance_as_parameter=True.
- **covariance_as_parameter** (*bool*) – The covariance estimated by the Estimator as parameter of the OCP. It will be put in the end of parameter vector.
- **mean_p_indices** (*list*) – If mean_as_parameter=True, mean_p_indices is a list of tuples (pairs), where the first element of the tuple is the index in the mean vector and the second is the index in the p vector.
- **cov_p_indices** (*list*) – If covariance_as_parameter=True, cov_p_indices is a list of tuples (pairs), where the first element of the tuple is the index in the vectorized covariance matrix and the second is the index in the p vector.
- **state_rearrangement_function** (*function*) – A function that can be used to rearrange the initial condition in cases where the estimated states is not equal to initial condition vector of the OCP. For instance, when the OCP has multiple representations of the system. The provided function has the estimated stated as input and has to return a initial condition vector.

get_measurement()

Get measurements from the plant. It will return a tuple with the current measurement and the current control

Return type tuple

get_new_control(*x_k*, *u_k*, *p=None*)

Use solution_method to obtain new controls

Parameters

- **x_k** – DM
- **u_k** – DM
- **p** –

get_states(*t_k*, *y_k*, *u_k*)

Get states out of a measurement.

Parameters

- **t_k** (*DM*) – time of the measurement
- **y_k** (*DM*) – measurements
- **u_k** (*DM*) – controls

Returns DM

run(*iterations=0*)

Starts computing control and sending it to the plant.

Parameters **iterations** (*int*) – the number of iterations that the MPC will run. To run it indefinitely use iterations = 0

run_fixed_control(*u*, *iterations*)

Run the plant with a fixed control, can be used for initialization purposes.

Parameters

- **u** (*list | DM | float | int*) – control value
- **iterations** (*int*) – the number of iterations that the MPC will run.

run_fixed_control_with_estimator (*u*, *iterations*)

Run the plant with a fixed control, can be used for initialization purposes.

Parameters

- **u** (*list | DM | float | int*) – control value
- **iterations** (*int*) – the number of iterations that the MPC will run.

send_control (*u*)

Sent controls to the plant.

Parameters **u** – DM

1.1.4.3 yaocptool.mpc.plant module

class yaocptool.mpc.plant.**Plant**

__init__ ()

get_measurement ()

n_x

set_control (*u*)

class yaocptool.mpc.plant.**PlantSimulation** (*model*, *x_0*, ***kwargs*)

Bases: *yaocptool.mpc.plant.Plant*

Simulates a plant using a model.

__init__ (*model*, *x_0*, ***kwargs*)

Plant which uses a SystemModel.simulate to obtain the measurements.

Parameters

- **model** (SystemModel) – simulation model
- **x_0** (*DM*) – initial condition
- **t_s** (*DM*) – (default: 1) sampling time
- **u** (*DM*) – (default: 0) initial control
- **y_guess** (*DM*) – initial guess for algebraic variables for simulation
- **t_0** (*DM*) – (default: 0) initial time
- **integrator_options** (*dict*) – integrator options
- **has_noise** (*bool*) – Turn on/off the process/measurement noise
- **r_n** (*DM*) – Measurement noise covariance matrix
- **r_v** (*DM*) – Process noise covariance matrix
- **noise_seed** – Seed for the random number generator used to create noise. Use the same seed for the repeatability in the experiments.

get_measurement ()

Return the plant measurement of a simulated model and advance time by 't_s'. Return the measurement time, the measurement [x; y], and the controls.

Return type tuple

Returns (timestamp, measuremnt, control)

n_x

set_control(*u*)
set a new control for the plant

Parameters **u** (*DM*) – new control vector

1.1.4.4 Module contents

1.1.5 yaocptool.optimization package

1.1.5.1 Submodules

1.1.5.2 yaocptool.optimization.abstract_optimization_problem module

class yaocptool.optimization.abstract_optimization_problem.**AbstractOptimizationProblem**(***kw*
Bases: object

__init__(***kwargs*)

Abstract Optimization Problem class Optimization problem

$$\min_x f(x,p)$$
$$\text{s.t.:} g_{lb} \leq g(x,p) \leq g_{ub}$$
$$Objectattributes : x- > optimizationvariablesg- > constraint$$

Parameters **n_x** – int

create_parameter(*name*, *size=1*)

create_variable(*name*, *size=1*, *lb=-inf*, *ub=inf*)
Create an optimization variable

Parameters

- **name** (*str*) – Name of the optimization variable.
- **size** (*int*) – Size of the variable (default = 1)
- **lb** – Lower bound of the variable. If the given 'size' is greater than one but a scalar is passed as lower bound, a vector of lb of size 'size' will be used as a lower bound. (default = [-inf]*size)
- **ub** – Upper bound of the variable. If the given 'size' is greater than one but a scalar is passed as upper bound, a vector of ub of size 'size' will be used as a upper bound. (default = [inf]*size)

Returns Return the variable

Return type MX

get_default_call_dict()

get_problem_dict()

get_solver()

include_equality(*expr*, *rhs=None*)
Include a equality with the following form expr = rhs

Parameters

- **`expr`** – expression, this is the only term that should contain symbolic variables
- **`rhs`** – right hand side, by default it is a vector of zeros with same size of expr. If the 'expr' size is greater than one but a scalar is passed as 'rhs', a vector of 'rhs' with size of 'expr' will be used as right hand side. (default = [0]*size)

`include_inequality` (*expr*, *lb=None*, *ub=None*)

Include inequality to the problem with the following form lb <= expr <= ub

Parameters

- **`expr`** – expression for the inequality, this is the only term that should contain symbolic variables
- **`lb`** – Lower bound of the inequality. If the 'expr' size is greater than one but a scalar is passed as lower bound, a vector of lb with size of 'expr' will be used as a lower bound. (default = [-inf]*size)
- **`ub`** – Upper bound of the inequality. If the 'expr' size is greater than one but a scalar is passed as upper bound, a vector of ub with size of 'expr' will be used as a upper bound. (default = [inf]*size)

`include_parameter` (*par*)

`include_variable` (*variable*, *lb=-inf*, *ub=inf*)

Include a symbolic variable in the optimization problem

Parameters

- **`variable`** – variable to be included
- **`lb`** – Lower bound of the variable. If the given variable size is greater than one but a scalar is passed as lower bound, a vector of lb with size of the given variable will be used as a lower bound. (default = [-inf]*size)
- **`ub`** – Upper bound of the variable. If the given variable size is greater than one but a scalar is passed as upper bound, a vector of ub with size of the given variable will be used as a upper bound. (default = [inf]*size)

`set_objective` (*expr*)

`solve` (*initial_guess=None*, *call_dict=None*, *p=None*, *lam_x=None*, *lam_g=None*)

Parameters

- **`initial_guess`** – Initial guess
- **`call_dict`** – a dictionary containing 'lbx', 'ubx', 'lbg', 'ubg'. If not given, the one obtained with self.get_default_call_dict will be used.
- **`p`** – parameters
- **`lam_x`** –
- **`lam_g`** –

Returns dictionary with solution

1.1.5.3 yaocptool.optimization.nonlinear_problem module

class yaocptool.optimization.nonlinear_problem.**NonlinearOptimizationProblem**(***kwargs*)

Bases: *yaocptool.optimization.abstract_optimization_problem.AbstractOptimizationProblem*

__init__(***kwargs*)

Nonlinear Optimization Problem class Optimization problem

$$\min_x f(x,p)$$
$$\text{s.t.:} g_{lb} \leq g(x,p) \leq g_{ub}$$

Object attributes: x -> optimization variables g -> constraint

1.1.5.4 yaocptool.optimization.quadratic_problem module

class yaocptool.optimization.quadratic_problem.**QuadraticOptimizationProblem**(***kwargs*)

Bases: *yaocptool.optimization.abstract_optimization_problem.AbstractOptimizationProblem*

__init__(***kwargs*)

Quadratic Optimization Problem class Optimization problem

$$\min_x f(x,p)$$
$$\text{s.t.:} g_{lb} \leq g(x,p) \leq g_{ub}$$

Object attributes: x -> optimization variables g -> constraint

1.1.5.5 Module contents

1.1.6 yaocptool.parallel package

1.1.6.1 Submodules

1.1.6.2 yaocptool.parallel.worker module

class yaocptool.parallel.worker.**Worker**(*obj_class*, *obj_arg*, *function_name*, *queue_in*, *queue_out*)

Bases: `multiprocessing.process.Process`

Creates new process that creates and object of class 'obj_class' with 'obj_arg' argument. It will consume one element from each queue_in and call function 'function_name' the consumed elements as argument. It will put the return of the 'function_name' call in all Queues in queue_out

__init__(*obj_class*, *obj_arg*, *function_name*, *queue_in*, *queue_out*)

x.__init__(...) initializes x; see help(type(x)) for signature

run()

Method to be run in sub-process; can be overridden in sub-class

1.1.6.3 Module contents

1.1.7 yaocptool.stochastic package

1.1.7.1 Submodules

1.1.7.2 yaocptool.stochastic.pce module

class yaocptool.stochastic.pce.**PCEConverter**(*socp*, ***kwargs*)

__init__(*socp*, ***kwargs*)

Parameters

- **socp** (StochasticOCP) – Stochastic Optimal Control Problem
- **n_samples** (*int*) – number of samples of the parameters. If none is provided, the minimum number of samples will be used, depending on the number of uncertain parameters and polynomial order
- **pc_order** (*int*) – order of the polynomial, for the polynomial approximation. (default: 3)

convert_socp_to_ocp_with_pce()

n_pol_parameters

n_uncertain

yaocptool.stochastic.pce.**get_ls_factor**(*n_uncertain*, *n_samples*, *pc_order*, *lamb=0.0*)

1.1.7.3 yaocptool.stochastic.util module

yaocptool.stochastic.util.**sample_parameter_log_normal_distribution_with_sobol**(*mean*, *covariance*, *n_samples=1*)

Sample parameter using Sobol sampling with a log-normal distribution.

Parameters

- **mean** –
- **covariance** –
- **n_samples** –

Returns

yaocptool.stochastic.util.**sample_parameter_normal_distribution_with_sobol**(*mean*, *covariance*, *n_samples=1*)

Sample parameter using Sobol sampling with a normal distribution.

Parameters

- **mean** –

- **covariance** –
- **n_samples** –

Returns

1.1.7.4 Module contents

1.1.8 yaocptool.util package

1.1.8.1 Submodules

1.1.8.2 yaocptool.util.util module

`yaocptool.util.util.`**`blockdiag`**(*matrices_list*)

Receives a list of matrices and return a block diagonal.

Parameters **matrices_list** (*list*) – list of matrices

`yaocptool.util.util.`**`convert_expr_from_tau_to_time`**(*expr*, *t_sym*, *tau_sym*, *t_k*, *t_kp1*)

`yaocptool.util.util.`**`create_constant_theta`**(*constant*, *dimension*, *finite_elements*)

`yaocptool.util.util.`**`expm`**(*a_matrix*)

Since casadi does not have native support for matrix exponential, this is a trick to computing it. It can be quite expensive, specially for large matrices. THIS ONLY SUPPORT NUMERIC MATRICES AND MX VARIABLES, DOES NOT SUPPORT SX SYMBOLIC VARIABLES.

Parameters **a_matrix** (*DM*) – matrix

Returns

`yaocptool.util.util.`**`find_variables_indices_in_vector`**(*var*, *vector*)

`yaocptool.util.util.`**`join_thetas`**(**args*)

`yaocptool.util.util.`**`remove_variables_from_vector`**(*var*, *vector*)

Returns a vector with items removed

Parameters

- **var** – items to be removed
- **vector** – vector which will have items removed

Returns

`yaocptool.util.util.`**`remove_variables_from_vector_by_indices`**(*vector*, *indices*)

Returns a vector with items removed

Parameters

- **vector** – vector which will have items removed
- **indices** (*list*) – list of indices for which the variables need to be removed.

Returns

1.1.8.3 Module contents

1.2 Submodules

1.3 yaocptool.config module

Created on Wed Nov 16 15:27:37 2016

@author: marco

1.4 Module contents

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

O

P

Q

R

S

T

U

V

W

X

Y