mpopt

Release 0.1.2

Jan 14, 2023

Contents:

CHAPTER 1

MPOPT

MPOPT is a collection of modules to solve multi-stage optimal control problems(OCPs) using pseudo-spectral collocation method. This module creates Nonlinear programming problem (NLP) from the given OCP description, which is then solved by CasADi nlpsolver using various available plugins such as *ipopt*, *snopt* etc.

Main features of the solver are :

- Customizable collocation approximation, compatable with Legendre-Gauss-Radau, Legendre-Gauss-Lobatto, Chebyshev-Gauss-Lobatto roots.
- Intuitive definition of OCP/multi-phase OCP
- Single-phase as well as multi-phase OCP solving capability using user defined collocation approximation
- Adaptive grid refinement schemes for robust solutions
- NLP solution using algorithmic differentiation capability offered by CasADi, multiple NLP solver compatibility 'ipopt', 'snopt', 'sqpmethod' etc.
- Sophisticated post-processing module for interactive data visualization

CHAPTER 2

Getting started

A brief overview of the package and capabilities are demonstrated with simple moon-lander OCP example in Jupyter notebook.

- Get started with MPOPT

CHAPTER 3

Installation

Install and try the package using

```
$ pip install mpopt
$ wget https://raw.githubusercontent.com/mpopt/mpopt/master/examples/moon_lander.py
$ python3 moon_lander.py
```

If you want to downloaded it from source, you may do so either by:

- Downloading it from GitHub page
  - Unzip the folder and you are ready to go
- Or cloning it to a desired directory using git:
  - `$ git clone https://github.com/mpopt/mpopt.git --branch master`
- Install package using
  - `$ make install`
- Test installation using
  - `$ make test`
- Try moon-lander example using
  - `$ make run`

CHAPTER 4

Documentation

- Refer Documentation

CHAPTER 5

A sample code to solve moon-lander OCP (2D)

```
# Moon lander OCP direct collocation/multi-segment collocation
from mpopt import mp

# Define OCP
ocp = mp.OCP(n_states=2, n_controls=1)
ocp.dynamics[0] = lambda x, u, t: [x[1], u[0] - 1.5]
ocp.running_costs[0] = lambda x, u, t: u[0]
ocp.terminal_constraints[0] = lambda xf, tf, x0, t0: [xf[0], xf[1]]
ocp.x00[0] = [10.0, -2.0]
ocp.lbu[0], ocp.ubu[0] = 0, 3

# Create optimizer(mpo), solve and post process(post) the solution
mpo, post = mp.solve(ocp, n_segments=20, poly_orders=3, scheme="LGR", plot=True)
```

CHAPTER 6

Authors

- **Devakumar THAMMISETTY**
- **Prof. Colin Jones** (Co-author)

CHAPTER 7

License

CHAPTER 8

Acknowledgements

- **Petr Listov**

CHAPTER 9

MPOPT package implementation

9.1 mpopt base class

class mpopt.mpopt.**mpopt**(*problem: mpopt.mpopt.OCP, n_segments: int = 1, poly_orders: List[int] = [9], scheme: str = 'LGR', **kwargs*)

Bases: object

Multiphase Optimal Control Problem Solver

This is the base class, implementing the OCP discretization, transcription and calls to NLP solver

Examples :

```
>>> # Moon lander problem
>>> from mpopt import mp
>>> ocp = mp.OCP(n_states=2, n_controls=1, n_phases=1)
>>> ocp.dynamics[0] = lambda x, u, t: [x[1], u[0] - 1.5]
>>> ocp.running_costs[0] = lambda x, u, t: u[0]
>>> ocp.terminal_constraints[0] = lambda xf, tf, x0, t0: [xf[0], xf[1]]
>>> ocp.x00[0] = [10, -2]
>>> ocp.lbu[0] = 0; ocp.ubu[0] = 3
>>> ocp.lbtf[0] = 3; ocp.ubtf[0] = 5
>>> opt = mp.mpopt(ocp, n_segments=20, poly_orders=[3]*20)
>>> solution = opt.solve()
>>> post = opt.process_results(solution, plot=True)
```

static compute_interpolation_taus_corresponding_to_original_grid(*nodes_req, seg_widths*)

Compute the taus on original solution grid corresponding to the required interpolation nodes

:param : nodes_req: target_nodes :param : seg_widths: width of the segments whose sum equal 1

Returns taus: List of taus in each segment corresponding to nodes_req on target_grid

compute_states_from_solution_dynamics(*solution, phase: int = 0, nodes=None*)

solution : NLP solution

create_nlp() → Tuple

Create Nonlinear Programming problem for the given OCP

Parameters **None** –

Returns

(nlp_problem, nlp_bounds)

nlp_problem Dictionary (f, x, g, p) f - Objective function x - Optimization variables vector g - constraint vector p - parameter vector

nlp_bounds Dictionary (lbx, ubx, lbg, ubg) lbx - Lower bound for the optimization variables (x) ubx - Upper bound for the optimization variables (x) lbu - Lower bound for the constraints vector (g) ubu - Upper bound for the constraints vector (g)

Return type Tuple

create_solver(*solver: str = 'ipopt', options: Dict[KT, VT] = {}*) → None

Create NLP solver

:param [solver: Optimization method to be used in nlp_solver (List of plugins] avaiable at http://casadi.sourceforge.net/v2.0.0/api/html/d6/d07/classcasadi_1_1NlpSolver.html)

:param [options: Dictionary] List of options for the optimizer (Based on CasADi documentation)

Returns None

Updates the nlpsolver object in the present optimizer class

create_variables() → None

Create casadi variables for states, controls, time and segment widths which are used in NLP transcription

Initialized casadi varaibles for optimization.

args: None returns : None

discretize_phase(*phase: int*) → Tuple

Discretize single phase of the Optimal Control Problem

Parameters **phase** – index of the phase (starting from 0)

returns : Tuple : Constraint vector (G, Gmin, Gmax) and objective function (J)

get_discretized_dynamics_constraints_and_cost_matrices(*phase: int = 0*) → Tuple

Get discretized dynamics, path constraints and running cost at each collocation node in a list

:param : phase: index of phase

Returns

(f, c, q) f - List of constraints for discretized dynamics c - List of constraints for discretized path constraints q - List of constraints for discretized running costs

Return type Tuple

get_dynamics_residuals(*solution, nodes=None, grid_type=None, residual_type=None, plot=False, fig=None, axs=None*)

Compute residual of the system dynamics at given taus (Normalized [0, 1]) by interpolating the given solution onto a fixed grid consisting of single segment per phase with

roots at given target_nodes.

:param : grid_type: target grid type (normalized between 0 and 1) :param : solution: solution of the NLP as reported by the solver :param : nodes: grid where the residual is computed (between tau0, tau1) in a list (nodes[0] -> Nodes for phase0) :param : options: Options for the target grid :param : residual_type:

None - Actual residual values "relative" - Scaled residual between -1 and 1

Returns residuals: residual vector for the dynamics at the given taus

get_dynamics_residuals_single_phase(*solution*, *phase: int = 0*, *target_nodes: List[T] = None*)

Compute residual of the system dynamics at given taus (Normalized [0, 1]) by interpolating the given solution onto a fixed grid consisting of single segment per phase with

roots at given target_nodes.

:param : target_nodes: target grid nodes (normalized between 0 and 1) :param : solution: solution of the NLP as reported by the solver

Returns residuals: residual vector for the dynamics at the given taus

get_event_constraints() → Tuple

Estimate the constraint vectors for linking the phases

Parameters **None** –

Returns

Constraint vectors (E, Emin, Emax) containing phase linking constraints, discontinuities across states, controls and time variables.

Return type Tuple

static get_interpolated_time_grid(*t_orig*, *taus: numpy.ndarray*, *poly_orders: numpy.ndarray*, *tau0: float*, *tau1: float*)

Update the time vector with the interpolated grid across each segment of the original optimization problem

:param : t_orig: Time grid of the original optimization problem (unscaled/scaled) :param : taus: grid of the interpolation taus across each segment of the original OCP :param : poly_orders: Order of the polynomials across each segment used in solving OCP

Returns time: Interpolated time grid

get_nlp_constrains_for_control_input_at_mid_colloc_points(*phase: int = 0*) → Tuple

Get NLP constrains on control input at mid points of the collocation nodes

Box constraints on control input

:param : phase: index of the corresponding phase

returns:

Tuple [(DU, DUmin, DUmax)] mU - CasADi vector of constraints for the control input at mid colloc points mUmin - Respective lowever bound vector mUmax - Respective upper bound vector

get_nlp_constrains_for_control_slope_continuity_across_segments(*phase: int = 0*) → Tuple

Get NLP constrains to maintain control input slope continuity across segments

:param : phase: index of the corresponding phase

Returns

(DU, DUmin, DUmax)

DU - CasADi vector of constraints for the slope of control input across segments

DUmin - Respective lowever bound vector DUmax - Respective upper bound vector

Return type Tuple

get_nlp_constraints_for_control_input_slope(*phase: int = 0*) → Tuple

Get NLP constraints slope on control input (U)

:param : phase: index of the corresponding phase

returns:

Tuple [(DU, DUmin, DUmax)] DU - CasADi vector of constraints for the slope of control input DUmin - Respective lowever bound vector DUmax - Respective upper bound vector

get_nlp_constraints_for_dynamics(*f: List[T] = [], phase: int = 0*) → Tuple

Get NLP constraints for discretized dynamics

:param : f: Discretized vector of dynamics function evaluated at collocation nodes :param : phase: index of the phase corresponding to the given dynamics

returns:

Tuple [(F, Fmin, Fmax)] F - CasADi vector of constraints for the dynamics Fmin - Respective lowever bound vector Fmax - Respective upper bound vector

get_nlp_constraints_for_path_contraints(*c: List[T] = [], phase: int = 0*) → Tuple

Get NLP constraints for discretized path constraints

:param : c: Discretized vector of path constraints evaluated at collocation nodes :param : phase: index of the corresponding phase

returns:

Tuple [(C, Cmin, Cmax)] C - CasADi vector of constraints for the path constraints Cmin - Respective lowever bound vector Cmax - Respective upper bound vector

get_nlp_constraints_for_terminal_contraints(*phase: int = 0*) → Tuple

Get NLP constraints for discretized terminal constraints

:param : phase: index of the corresponding phase

returns:

Tuple [(TC, TCmin, TCmax, J)] TC - CasADi vector of constraints for the terminal constraints TCmin - Respective lowever bound vector TCmax - Respective upper bound vector J - Terminal cost

get_nlp_variables(*phase: int*) → Tuple

Retrieve optimization variables and their bounds for a given phase

:param : phase: index of the phase (starting from 0)

Returns

(Z, Zmin, Zmax) Z - Casadi SX vector containing optimization variables for the given phase (X, U, t0, tf) Zmin - Lower bound for the variables in 'Z' Zmax - Upper bound for the variables in 'Z'

Return type Tuple

get_residual_grid_taus(*phase: int = 0, grid_type: str = None*)
Select the non-collocation nodes in a given phase

This is often useful in estimation of residual once the OCP is solved. Starting and end nodes are not included.

:param : phase: Index of the phase :param : grid_type: Type of non-collocation nodes (fixed, mid-points, spectral)

Returns points: List of normalized collocation points in each segment of the phase

get_segment_width_parameters(*solution: Dict[KT, VT]*) → List[T]
Get segment widths in all phases

All segment widths are considered equal

:param [solution: Solution to the nlp from wich the seg_width parameters are] computed (if Adaptive)

Returns

seg_widths: numerical values for the fractions of the segment widths that equal 1 in each phase

get_solver_warm_start_input_parameters(*solution: Dict[KT, VT] = None*)
Create dictionary of objects for warm starting the solver using results in 'solution'

:param : solution: Solution of nlp_solver

Returns dict: (x0, lam_x0, lam_g0)

get_state_second_derivative(*solution, grid_type='spectral', nodes=None, plot=False, fig=None, axs=None*)
Compute residual of the system states at given taus (Normalized [0, 1]) by interpolating the given solution onto a fixed grid consisting of single segment per phase with

roots at given target_nodes.

:param : grid_type: target grid type (normalized between 0 and 1) :param : solution: solution of the NLP as reported by the solver :param : options: Options for the target grid

Returns residuals: residual vector for the states at the given taus

get_state_second_derivative_single_phase(*solution, phase: int = 0, nodes: List[T] = None, grid_type: str = None, residual_type: str = None*)
Compute residual of the system dynamics at given taus (Normalized [0, 1]) by interpolating the given solution onto a fixed grid consisting of single segment per phase with

roots at given target_nodes.

:param : target_nodes: target grid nodes (normalized between 0 and 1) :param : solution: solution of the NLP as reported by the solver :param : residual_type: 'relative' if relative is req.

Returns residuals: residual vector for the dynamics at the given taus

get_states_residuals(*solution, nodes=None, grid_type='spectral', residual_type=None, plot=False, fig=None, axs=None*)
Compute residual of the system dynamics at given taus (Normalized [0, 1]) by interpolating the given solution onto a fixed grid consisting of single segment per phase with

roots at given target_nodes.

:param : grid_type: target grid type (normalized between 0 and 1) :param : solution: solution of the NLP as reported by the solver :param : nodes: grid where the residual is computed (between tau0, tau1) in a list (nodes[0] -> Nodes for phase0) :param : options: Options for the target grid

Returns residuals: residual vector for the dynamics at the given taus

init_segment_width() → None
Initialize segment width in each phase

Segment width is normalized so that sum of all the segment widths equal 1

args: None returns: None

init_solution_per_phase(*phase: int*) → numpy.ndarray
Initialize solution vector at all collocation nodes of a given phase.

The initial solution for a given phase is estimated from the initial and terminal conditions defined in the OCP. Simple linear interpolation between initial and terminal conditions is used to estimate solution at interior collocation nodes.

:param : phase: index of phase

Returns initialized solution for given phase

Return type solution

init_trajectories(*phase: int = 0*) → casadi.casadi.Function
Initialize trajectories of states, constrols and time variables

:param : phase: index of the phase

Returns

trajectories: CasADi function which returns states, controls and time variable for the given phase when ca
t0, tf - unscaled AND x, u, t - scaled trajectories

initialize_solution() → numpy.ndarray
Initialize solution for the NLP from given OCP description

Parameters **None** –

Returns Initialized solution for the NLP

Return type solution

interpolate_single_phase(*solution, phase: int = 0, target_nodes: numpy.ndarray = None, grid_type=None, options: Set[T] = {}*)
Interpolate the solution at given taus

:param : solution: solution as reported by nlp solver :param : phase: index of the phase :param : target_nodes: List of nodes at which interpolation is performed

Returns

Tuple - (X, DX, DU) X - Interpolated states DX - Derivative of the interpolated states based on PS polynomials DU - Derivative of the interpolated controls based on PS polynomials

process_results(*solution, plot: bool = True, scaling: bool = False*)
Post process the solution of the NLP

:param : solution: NLP solution as reported by the solver :param : plot: bool

True - Plot states and variables in a single plot with states in a subplot and controls in another.
False - No plot

:param [scaling: bool] True - Plot the scaled variables False - Plot unscaled variables meaning, original solution to the problem

Returns post: Object of post_process class (Initialized)

solve(*initial_solution: Dict[KT, VT] = None, reinitialize_nlp: bool = False, solver: str = 'ipopt', nlp_solver_options: Dict[KT, VT] = {}, mpopt_options: Dict[KT, VT] = {}, **kwargs*) → Dict[KT, VT]

Solve the Nonlinear Programming problem

:param [init_solution: Dictionary containing initial solution with keys] x or x0 - Initional solution for the nlp variables

:param [reinitialize_nlp: (True, False)] True - Reinitialize NLP solver object False - Use already created object if available else create new one

:param [nlp_solver_options: Options to be passed to the nlp_solver while creating] the solver object, not while solving (like initial conditions)

:param : mpopt_options: Options dict for the optimizer

Returns solution: Solution as reported by the given nlp_solver object

validate()

Validate initialization of the optimizer object

9.2 Collocation

9.2.1 Collocation class

class mpopt.mpopt.**Collocation**(*poly_orders: List[T] = [], scheme: str = 'LGR', polynomial_type: str = 'lagrange'*)

Bases: object

Collocation functionality for optimizer

Functionality related to polynomial basis, respective differential and integral matrices calculation is implemented here.

Numpy polynomial modules is used for calculating differentiation and quadrature weights. Hence, computer precision can affect these derivative calculations

D_MATRIX_METHOD = 'symbolic'

TVAR = SX(t)

get_composite_differentiation_matrix(*poly_orders: List[T] = None, order: int = 1*)

Get composite differentiation matrix for given collocation approximation

:param [poly_orders: order of the polynomials used in collocation with each] element representing one segment

get_composite_interpolation_Dmatrix_at(*taus, poly_orders: List[T] = None, order: int = 1*)

Get differentiation matrix corresponding to given basis polynomial degree at nodes different from collocation nodes

:param [taus: List of scaled taus (between 0 and 1) with length of list equal] to length of poly_orders (= number of segments)

:param [poly_orders: order of the polynomials used in collocation with each] element representing one segment

Returns D: Composite differentiation matrix

get_composite_interpolation_matrix(*taus*, *poly_orders: List[T] = None*)
Get differentiation matrix corresponding to given basis polynomial degree

:param [taus: List of scaled taus (between 0 and 1) with length of list equal] to length of poly_orders (= number of segments). Note- taus are not assumed to have overlap between segments(end element != start of next phase)

:param [poly_orders: order of the polynomials used in collocation with each] element representing one segment

Returns I: composite interpolation matrix

get_composite_quadrature_weights(*poly_orders: List[T] = None*, *tau0=None*, *tau1=None*)
Get composite quadrature weights for given collocation approximation

:param [poly_orders: order of the polynomials used in collocation with each] element representing one segment

get_diff_matrices(*poly_orders: List[T] = None*, *order: int = 1*)
Get differentiation matrices for given collocation approximation

:param : poly_orders: order of the polynomials used in collocation with each element representing one segment

get_diff_matrix(*key*, *taus: numpy.ndarray = None*, *order: int = 1*)
Get differentiation matrix corresponding to given basis polynomial degree

:param : degree: order of the polynomial used in collocation :param : taus: Diff matrix computed at these nodes if not None.

Returns D: Differentiation matrix

classmethod get_diff_matrix_fn(*polynomial_type: str = 'lagrange'*)
Return a function that returns differentiation matrix

:param : polynomial_type: (lagrange)

Returns Diff matrix function with arguments (degree, taus_at=None)

get_interpolation_Dmatrices_at(*taus*, *keys: List[T] = None*, *order: int = 1*)
Get differentiation matrices at the interpolated nodes (taus), different from the collocation nodes.

:param [taus: List of scaled taus (between 0 and 1) with length of list equal] to length of poly_orders (= number of segments)

:param : keys: keys of the roots and polys Dict element containing roots of the legendre polynomials and polynomials themselves

Returns

Dict [(key, value)] key - segment number (starting from 0) value - Differentiation matrix(C) such that DX_tau = D*X_colloc where

X_colloc is the values of states at the collocation nodes

get_interpolation_matrices(*taus*, *poly_orders: List[T] = None*)
Get interpolation matrices corresponding to each poly_order at respective element in list of taus

:param [taus: List of scaled taus (between 0 and 1) with length of list equal] to length of poly_orders (= number of segments).

:param [poly_orders: order of the polynomials used in collocation with each] element representing one segment

Returns

(key, value) key - segment number (starting from 0) value - interpolation matrix(C) such that X_new = C*X

Return type Dict

get_interpolation_matrix(*taus*, *degree*)

Get interpolation matrix corresponsing nodes (taus) where the segment is approximated with polynomials of degree (degree)

:param : taus: Points where interpolation is performed :param : degree: Order of the collocation polynomial

classmethod get_lagrange_polynomials(*roots*)

Get basis polynomials given the collocation nodes

:param : roots: Collocation points

classmethod get_polynomial_function(*polynomial_type: str = 'lagrange'*)

Get function which returns basis polynomials for the collocation given polynomial degree

:param : polynomial_type: str, 'lagrange'

Returns poly_basis_fn: Function which returns basis polynomials

get_quad_weight_matrices(*keys: List[T] = None*, *tau0=None*, *tau1=None*)

Get quadrature weights for given collocation approximation

:param : keys: keys of the Dict element (roots and polys), Normally these keys are equal to the order of the polynomial

get_quadrature_weights(*key*, *tau0=None*, *tau1=None*)

Get quadrature weights corresponding to given basis polynomial degree

:param : degree: order of the polynomial used in collocation

classmethod get_quadrature_weights_fn(*polynomial_type: str = 'lagrange'*)

Return a function that returns quadrature weights for the cost function approx.

:param : polynomial_type: (lagrange)

Returns quadrature weights function with arguments (degree)

init_polynomials(*poly_orders*) → None

Initialize roots of the polynomial and basis polynomials

:param : poly_orders: List of polynomial degrees used in collocation

init_polynomials_with_customized_roots(*roots_dict: Dict[KT, VT] = None*) → None

Initialize polynomials with predefined roots

:param : roots_dict: Dictionary with a key for the roots and polys (Ideally not numbers as they are already taken by the regular polynominals)

9.2.2 Collocation Roots class

class mpopt.mpopt.**CollocationRoots**(*scheme: str = 'LGR'*)

Bases: object

Functionality related to commonly used gauss quadrature schemes such as

Legendre-Gauss (LG) Legendre-Gauss-Radau (LGR) Legendre-Gauss-Lobatto (LGL) Chebyshev-Gauss-Lobatto (CGL)

classmethod get_collocation_points(*scheme: str*)

Get function that returns collocation points for the given scheme

:param : scheme: quadrature scheme to find the collocation points

returns: Function, that retuns collocation points when called with polynomial degree

static roots_chebyshev_gauss_lobatto(*tau_min=-1*, *tau_max=1*)

Get Chebyshev-gauss-lobatto collocation points in the interval [_TAU_MIN, _TAU_MAX]

args: None

returns: a function that returns collocation points given polynomial degree

static roots_legendre_gauss(*tau_min=-1*, *tau_max=1*)

Get legendre-gauss-radau collocation points in the interval [_TAU_MIN, _TAU_MAX)

args: None

returns: a function that returns collocation points given polynomial degree

static roots_legendre_gauss_lobatto(*tau_min=-1*, *tau_max=1*)

Get legendre-gauss-lobatto collocation points in the interval [_TAU_MIN, _TAU_MAX]

args: None

returns: a function that returns collocation points given polynomial degree

static roots_legendre_gauss_radau(*tau_min=-1*, *tau_max=1*)

Get legendre-gauss-radau (Left aligned) collocation points in the interval [_TAU_MIN, _TAU_MAX]

args: None

returns: a function that returns collocation points, given polynomial degree

9.3 OCP definition

9.3.1 OCP definition class

class mpopt.mpopt.**OCP**(*n_states: int = 1*, *n_controls: int = 1*, *n_phases: int = 1*, *n_params=0*, ***kwargs*)

Bases: object

Define Optimal Control Problem

Optimal control problem definition in standard Bolza form.

Examples of usage:

```
>>> ocp = OCP(n_states=1, n_controls=1, n_phases=1)
>>> ocp.dynamics[0] = lambda x, u, t, a: [u[0]]
>>> ocp.path_constraints[0] = lambda x, u, t, a: [x[0] + u[0]]
>>> ocp.running_costs[0] = lambda x, u, t, a: x[0]
>>> ocp.terminal_costs[0] = lambda xf, tf, x0, t0, a: xf[0]
>>> ocp.terminal_constraints[0] = lambda xf, tf, x0, t0, a: [xf[0] + 2]
```

LB_DYNAMICS = 0

LB_PATH_CONSTRAINTS = -inf

LB_TERMINAL_CONSTRAINTS = 0

UB_DYNAMICS = 0

UB_PATH_CONSTRAINTS = 0

UB_TERMINAL_CONSTRAINTS = 0

get_dynamics (*phase: int = 0*)

Get dynamics function for the given phase

:param : phase: index of the phase (starting from 0)

Returns dynamics: system dynamics function with arguments x, u, t, a

get_path_constraints (*phase: int = 0*)

Get path constraints function for the given phase

:param : phase: index of the phase (starting from 0)

Returns path_constraints: path constraints function with arguments x, u, t, a

get_running_costs (*phase: int = 0*)

Get running_costs function for the given phase

:param : phase: index of the phase (starting from 0)

Returns running_costs: system running_costs function with arguments x, u, t, a

get_terminal_constraints (*phase: int = 0*)

Get terminal_constraints function for the given phase

:param : phase: index of the phase (starting from 0)

Returns terminal_constraints: system terminal_constraints function with arguments x, u, t, a

get_terminal_costs (*phase: int = 0*)

Get terminal_costs function for the given phase

:param : phase: index of the phase (starting from 0)

Returns terminal_costs: system terminal_costs function with arguments x, u, t, a

has_path_constraints (*phase: int = 0*) → bool

Check if given phase has path constraints in given OCP

:param : phase: index of phase

Returns status: bool (True/False)

has_terminal_constraints (*phase: int = 0*) → bool

Check if given phase has terminal equality constraints in given OCP

:param : phase: index of phase

Returns status: bool (True/False)

validate() → None
Validate dimensions and initialization of attributes

9.4 Adaptive grid refinement scheme-I & II

9.4.1 mpopt h-adaptive class

class mpopt.mpopt.**mpopt_h_adaptive**(*problem: mpopt.mpopt.OCP, n_segments: int = 1, poly_orders: List[int] = [9], scheme: str = 'LGR', **kwargs*)

Bases: *mpopt.mpopt.mpopt*

Multi-stage Optimal control problem (OCP) solver which implements iterative procedure to refine the segment width in each phase adaptively while keeping the same number of segments

Examples :

```
>>> # Moon lander problem
>>> from mpopt import mp
>>> ocp = mp.OCP(n_states=2, n_controls=1, n_params=0, n_phases=1)
>>> ocp.dynamics[0] = lambda x, u, t, a: [x[1], u[0] - 1.5]
>>> ocp.running_costs[0] = lambda x, u, t, a: u[0]
>>> ocp.terminal_constraints[0] = lambda xf, tf, x0, t0, a: [xf[0], xf[1]]
>>> ocp.x00[0] = [10, -2]
>>> ocp.lbu[0] = 0; ocp.ubu[0] = 3
>>> ocp.lbtf[0] = 3; ocp.ubtf[0] = 5
>>> opt = mp.mpopt_h_adaptive(ocp, n_segments=3, poly_orders=[2]*3)
>>> solution = opt.solve()
>>> post = opt.process_results(solution, plot=True)
```

compute_seg_width_based_on_input_slope(*solution*)
Compute the optimum segment widths based on slope of the control signal.

:param : solution: nlp solution as reported by the solver

Returns segment_widths: optimized segment widths based on present solution

compute_seg_width_based_on_residuals(*solution, method: str = 'merge_split'*)
Compute the optimum segment widths based on residual of the dynamics in each segment.

:param : solution: nlp solution as reported by the solver

Returns segment_widths: optimized segment widths based on present solution

static compute_segment_widths_at_times(*times, n_segments, t0, tf*)
Compute seg_width fractions corresponding to given times and number of segments

static compute_time_at_max_values(*t_grid, t_orig, du_orig, threshold: float = 0*)
Compute the times corresponding to max value of the given variable (du_orig)

:param : t_grid: Fixed grid :param : t_orig: time corresponding to collocation nodes and variable (du_orig) :param : du_orig: Variable to decide the output times

Returns time: Time corresponding to max, slope of given variable

static get_roots_wrt_equal_area(*residuals, n_segments*)

get_segment_width_parameters(*solution, options: Dict[KT, VT] = {'method': 'residual', 'sub_method': 'merge_split'}*)
Compute optimum segment widths in every phase based on the given solution to the NLP

:param : solution: solution to the NLP :param : options: Dictionary of options if required (Computation method etc.)

method Method used to refine the grid 'residual'

'merge_split' 'equal_area'

'control_slope'

Returns seg_widths: Computed segment widths in a 1-D list (each phase followed by previous)

static merge_split_segments_based_on_residuals(*max_residuals*, *segment_widths*, *ERR_TOL: float = 0.001*)

Merge/Split existing segments based on residual errors

Merge consecutive segments with residual below tolerance

:param : max_residuals: max residual in dynamics of each segment :param : segment_widths: Segment width corresponding to the residual

Returns segment_widths: Updated segment widths after merging/splitting

refine_segment_widths_based_on_residuals(*residuals*, *segment_widths*, *ERR_TOL: float = 0.001*, *method: str = 'merge_split'*)

Refine segment widths based on residuals of dynamics

:param : residuals: residual matrix of dynamics of each segment :param : segment_widths: Segment width corresponding to the residual

Returns segment_widths: Updated segment widths after refining

solve(*initial_solution: Dict[KT, VT] = None*, *reinitialize_nlp: bool = False*, *solver: str = 'ipopt'*, *nlp_solver_options: Dict[KT, VT] = {}*, *mpopt_options: Dict[KT, VT] = {}*, *max_iter: int = 10*, ***kwargs*) → Dict[KT, VT]

Solve the Nonlinear Programming problem

:param [init_solution: Dictionary containing initial solution with keys] x or x0 - Initional solution for the nlp variables

:param [reinitialize_nlp: (True, False)] True - Reinitialize NLP solver object False - Use already created object if available else create new one

:param [nlp_solver_options: Options to be passed to the nlp_solver while creating] the solver object, not while solving (like initial conditions)

:param [mpopt_options: Options dict for the optimizer] 'method': 'residual' or 'control_slope' 'sub_method': (if method is residual)

'merge_split' 'equal_area'

Returns solution: Solution as reported by the given nlp_solver object

9.5 Adaptive grid refinement scheme-III

9.5.1 mpopt-adaptive class

class mpopt.mpopt.**mpopt_adaptive**(*problem: mpopt.mpopt.OCP, n_segments: int = 1, poly_orders: List[int] = [9], scheme: str = 'LGR', **kwargs*)

Bases: *mpopt.mpopt.mpopt*

Multi-stage Optimal control problem (OCP) solver which implements seg-widths as optimization variables and solves for them along with the optimization problem.

Examples :

```
>>> # Moon lander problem
>>> from mpopt import mp
>>> ocp = mp.OCP(n_states=2, n_controls=1, n_phases=1)
>>> ocp.dynamics[0] = lambda x, u, t: [x[1], u[0] - 1.5]
>>> ocp.running_costs[0] = lambda x, u, t: u[0]
>>> ocp.terminal_constraints[0] = lambda xf, tf, x0, t0: [xf[0], xf[1]]
>>> ocp.x00[0] = [10, -2]
>>> ocp.lbu[0] = 0; ocp.ubu[0] = 3
>>> ocp.lbtf[0] = 3; ocp.ubtf[0] = 5
>>> opt = mp.mpopt_adaptive(ocp, n_segments=3, poly_orders=[2]*3)
>>> solution = opt.solve()
>>> post = opt.process_results(solution, plot=True)
```

create_solver(*solver: str = 'ipopt', options: Dict[KT, VT] = {}*) → None

Create NLP solver

:param [solver: Optimization method to be used in nlp_solver (List of plugins] avaiable at http://casadi.sourceforge.net/v2.0.0/api/html/d6/d07/classcasadi_1_1NlpSolver.html)

:param [options: Dictionary] List of options for the optimizer (Based on CasADi documentation)

Returns None

Updates the nlpsolver object in the present optimizer class

discretize_phase(*phase: int*) → Tuple

Discretize single phase of the Optimal Control Problem

Parameters **phase** – index of the phase (starting from 0)

returns : Tuple : Constraint vector (G, Gmin, Gmax) and objective function (J)

get_nlp_constrains_for_segment_widths(*phase: int = 0*) → Tuple

Add additional constraints on segment widths to the original NLP

get_nlp_variables(*phase: int = 0*)

Retrieve optimization variables and their bounds for a given phase

:param : phase: index of the phase (starting from 0)

Returns

(Z, Zmin, Zmax) Z - Casadi SX vector containing optimization variables for the given phase (X, U, t0, tf) Zmin - Lower bound for the variables in 'Z' Zmax - Upper bound for the variables in 'Z'

Return type Tuple

init_solution_per_phase(*phase: int*) → numpy.ndarray

Initialize solution vector at all collocation nodes of a given phase.

The initial solution for a given phase is estimated from the initial and terminal conditions defined in the OCP. Simple linear interpolation between initial and terminal conditions is used to estimate solution at interior collocation nodes.

:param : phase: index of phase

Returns initialized solution for given phase

Return type solution

init_trajectories(*phase: int = 0*) → casadi.casadi.Function

Initialize trajectories of states, constrols and time variables

:param : phase: index of the phase

Returns

trajectories: CasADi function which returns states, controls and time

variable for the given phase when called with NLP solution vector of all phases

t0, tf - unscaled AND x, u, t - scaled trajectories

process_results(*solution*, *plot: bool = True*, *scaling: bool = False*)

Post process the solution of the NLP

:param : solution: NLP solution as reported by the solver :param : plot: bool

True - Plot states and variables in a single plot with states in a subplot and controls in another.
False - No plot

:param [scaling: bool] True - Plot the scaled variables False - Plot unscaled variables meaning, original solution to the problem

Returns post: Object of post_process class (Initialized)

solve(*initial_solution: Dict[KT, VT] = None*, *reinitialize_nlp: bool = False*, *solver: str = 'ipopt'*, *nlp_solver_options: Dict[KT, VT] = {}*, *mpopt_options: Dict[KT, VT] = {}*, ***kwargs*) → Dict[KT, VT]

Solve the Nonlinear Programming problem

:param [init_solution: Dictionary containing initial solution with keys] x or x0 - Initional solution for the nlp variables

:param [reinitialize_nlp: (True, False)] True - Reinitialize NLP solver object False - Use already created object if available else create new one

:param [nlp_solver_options: Options to be passed to the nlp_solver while creating] the solver object, not while solving (like initial conditions)

:param : mpopt_options: Options dict for the optimizer

Returns solution: Solution as reported by the given nlp_solver object

9.6 Processing results

9.6.1 Post-processing class

class mpopt.mpopt.**post_process**(*solution: Dict[KT, VT] = {}, trajectories: List[T] = None, options: Dict[KT, VT] = {}*)

Bases: object

Process the results of mpopt optimizer for further processing and interactive visualization

This class contains various methods for processing the solution of OCP

Examples

```
>>> post = post_process(solution, trajectories, options)
```

get_data(*phases: List[T] = [], interpolate: bool = False*)

Get solution corresponding to given phases (original/interpolated)

:param : phases: List of phase indices :param : interpolate: bool

True - Interpolate the original grid (Refine the grid for smooth plot) False - Return original data

Returns

(x, u, t, a) x - interpolated states u - interpolated controls t - interpolated time grid

Return type Tuple

get_interpolated_data(*phases, taus: List[T] = []*)

Interpolate the original solution across given phases

:param : phases: List of phase indices :param : taus: collocation grid points across which interpolation is performed

Returns

(x, u, t, a) x - interpolated states u - interpolated controls t - interpolated time grid

Return type Tuple

static get_interpolated_time_grid(*t_orig, taus: numpy.ndarray, poly_orders: numpy.ndarray, tau0: float, tau1: float*)

Update the time vector with the interpolated grid across each segment of the original optimization problem

:param : t_orig: Time grid of the original optimization problem (unscaled/scaled) :param : taus: grid of the interpolation taus across each segment of the original OCP :param : poly_orders: Order of the polynomials across each segment used in solving OCP

Returns time: Interpolated time grid

get_interpolation_taus(*n: int = 75, taus_orig: numpy.ndarray = None, method: str = 'uniform'*)

Nodes across the normalized range [0, 1] or [-1, 1], to interpolate the data for post processing such as plotting

:param : n: number of interpolation nodes :param : taus_orig: original grid across which interpolation is to be performed :param : method: ("uniform", "other")

"uniform" : returns equally spaced grid points "other": returns mid points of the original grid recursively until 'n' elements

Returns taus: interpolation grid

static get_non_uniform_interpolation_grid(*taus_orig*, *n: int = 75*)

Increase the resolution of the given taus preserving the sparsity of the given grid

:param : taus_orig: original grid to be refined :param : n: max number of points in the refined grid.

Returns taus: refined grid

get_original_data(*phases: List[T] = []*)

Get optimized result for multiple phases

:param : phases: Optional, List of phases to retrieve the data from.

Returns

(x, u, t, a) x - states u - controls t - corresponding time vector

Return type Tuple

get_trajectories(*phase: int = 0*)

Get trajectories of states, controls and time vector for single phase

:param : phase: index of the phase

Returns

(x, u, t, a) x - states u - controls t - corresponding time vector

Return type Tuple

classmethod plot_all(*x*, *u*, *t*, *tics: str = None*, *fig=None*, *axs=None*, *legend: bool = True*, *name: str = ''*)

Plot states and controls

:param : x: states data :param : u: controls data :param : t: time grid

Returns

(fig, axs) fig - handle to the figure obj. of plot (matplotlib figure object) axs - handle to the axis of plot (matplotlib figure object)

Return type Tuple

static plot_curve(*ax, x, t, name=None, ylabel='', tics=['-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-'], legend_index=None*)

2D Plot given data (x, y)

:param : ax: Axis handle of matplotlib plot :param : x: y-axis data - numpy ndarray with first dimention having data :param : t: x-axis data

plot_phase(*phase: int = 0*, *interpolate: bool = True*, *fig=None*, *axs=None*)

Plot states and controls across given phase

:param : phase: index of phase :param : interpolate: bool

True - Plot refined data False - Plot original data

Returns

(fig, axs) fig - handle to the figure obj. of plot (matplotlib figure object) axs - handle to the axis of plot (matplotlib figure object)

Return type Tuple

plot_phases(*phases: List[T] = None, interpolate: bool = True, fig=None, axs=None, tics: List[T] = ['-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-'], name: str = ''*)

Plot states and controls across given phases

:param : phases: List of indices corresponding to phases :param : interpolate: bool

True - Plot refined data False - Plot original data

Returns

(fig, axs) fig - handle to the figure obj. of plot (matplotlib figure object) axs - handle to the axis of plot (matplotlib figure object)

Return type Tuple

classmethod plot_residuals(*time, residuals: numpy.ndarray = None, phases: List[T] = [0], name=None, fig=None, axs=None, tics=None*)

Plot residual in dynamics

classmethod plot_single_variable(*var_data, t, dims, name: str = None, ylabel: str = None, axis=1, fig=None, axs=None, tics=['-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-']*)

Plot given numpy array in various subplots

:param : var_data: input data to be plotted :param : t: xdata of the plot :param : dims: List of dimentions of the data to be plotted (List of list indicates

each internal list plotted in respective subplot)

:param : name: Variable name :param : ylabel: ylabel for the plot :param : axis: int, 0 or 1

0 - subplots are stacked horizintally 1 - subplots are stacked vertically

Returns

(fig, axs) fig - handle to the figure obj. of plot (matplotlib figure object) axs - handle to the axis of plot (matplotlib figure object)

Return type Tuple

plot_u(*dims: List[T] = None, phases: List[T] = None, axis: int = 1, interpolate: bool = True, fig=None, axs=None, tics=None, name='control', ylabel='Control variables'*)

Plot given dimenstions of controls across given phases stacked vertically or horizontally

:param [dims: List of dimentions of the control to be plotted (List of list indicates] each internal list plotted in respective subplot)

:param : phases: List of phases to plot :param : interpolate: bool

True - Plot refined data False - Plot original data

Returns

(fig, axs) fig - handle to the figure obj. of plot (matplotlib figure object) axs - handle to the axis of plot (matplotlib figure object)

Return type Tuple

plot_x(*dims: List[T] = None, phases: List[T] = None, axis: int = 1, interpolate: bool = True, fig=None, axs=None, tics=['-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-', '-']*)

Plot given dimenstions of states across given phases stacked vertically or horizontally

:param [dims: List of dimentions of the state to be plotted (List of list indicates] each internal list plotted in respective subplot)

:param : phases: List of phases to plot :param : interpolate: bool

True - Plot refined data False - Plot original data

Returns

(fig, axs) fig - handle to the figure obj. of plot (matplotlib figure object) axs - handle to the axis of plot (matplotlib figure object)

Return type Tuple

static sort_residual_data(*time, residuals, phases: List[T] = [0]*)

Sort the given data corresponding to plases

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Index

H

I

L

M

O

P

R

S

T

U

V