MPO

Release 1.0.2

Dustin Kenefake

Feb 17, 2021

TUROTIAL

MPO is a multiparametric programming solver written in Python, meant for solving general mpQPs and mpLPs with support for mixed integer and Quadratically constrained problems prospectively in the future. Optimized implementations of combinatorial algorithms and graph-based algorithms have been implemented. A focus of this solver is to implement parallel and scalable algorithms for multithreading compute.

CHAPTER

ONE

INSTALLATION

All you need to do is the following pip command in the relevant console.

```
pip install git+https://github.com/dkenefake/mpo.git
```

CHAPTER

TWO

TUTORIAL

2.1 Solving a MPQP Program

Here we are going to solve a classic transportation problem with multiparametric uncertainty. We have a set of plants and a set of markets with corresponding supplies and demand, and we want to minimize the transport cost between the plants and ensuring we satisfy all market demand. The multiparametric formulation is fleshed out in more detail in Multiparametric Optimization and Control by Pistikopolous et al.

This optimization problem leads to the following multiparametric optimization problem, with representing the markets' uncertain demands.

$$\min_{x} \frac{1}{2} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}^T \begin{bmatrix} 306.0 & 0 & 0 & 0 \\ 0 & 324.0 & 0 & 0 \\ 0 & 0 & 324.0 & 0 \\ 0 & 0 & 0 & 252.0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 25.0 \\ 25.0 \\ 25.0 \\ 25.0 \end{bmatrix}^T \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\text{s.t.} \begin{bmatrix} 1.0 & 1.0 & 0 & 0 \\ 0 & 0 & 1.0 & 1.0 \\ -1.0 & 0 & -1.0 & 0 \\ 0 & -1.0 & 0 & -1.0 \\ -1.0 & 0 & 0 & 0 \\ 0 & -1.0 & 0 & 0 \\ 0 & 0 & -1.0 & 0 \\ 0 & 0 & 0 & -1.0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \leq \begin{bmatrix} 350.0 \\ 600.0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ -1.0 & 0 \\ 0 & -1.0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$$

$$\begin{bmatrix} 1.0 & 0 \\ 0 & 1.0 \\ -1.0 & 0 \\ 0 & -1.0 \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} \leq \begin{bmatrix} 1e+03 \\ 1e+03 \\ 0 \\ 0 \end{bmatrix}$$

1.0
00 0
1.0 0
0 1.0 − 1.0
00 −1.0
0 −1.0
0 −1.0 − 1.0
00 0
0 −1.0
0 00
00 −1.0
0 0
−1.0

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \leq \begin{bmatrix} 350.0 \\ 600.0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ -1.0 & 0 \\ 0 & -1.0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$$

$$\begin{bmatrix} 1.0 & 0 \\ 0 & 1.0 \\ -1.0 & 0 \\ 0 & -1.0 \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} \leq \begin{bmatrix} 1e+03 \\ 1e+03 \\ 0 \\ 0 \end{bmatrix}$$

Using MPO, this is translated as the following python code. (The latex above was generated for me with `prog.latex()` if you were wondering if I typed that all out by hand.)

```
A = numpy.array([[1, 1, 0, 0], [0, 0, 1, 1], [-1, 0, -1, 0], [0, -1, 0, -1], [-1, 0,␣
↪0, 0], [0, -1, 0, 0], [0, 0, -1, 0], [0, 0, 0, -1]])
b = numpy.array([350, 600, 0, 0, 0, 0, 0, 0]).reshape(8, 1)
c = 25 * make_column([1, 1, 1, 1])
F = numpy.array([[0, 0], [0, 0], [-1, 0], [0, -1], [0, 0], [0, 0], [0, 0], [0, 0]])
Q = 2.0 * numpy.diag([153, 162, 162, 126])

CRa = numpy.vstack((numpy.eye(2), -numpy.eye(2)))
CRb = numpy.array([1000, 1000, 0, 0]).reshape(4, 1)
H = numpy.zeros((F.shape[1], Q.shape[0]))

prog = MPQP_Program(A, b, c, H, Q, CRa, CRb, F)
```

But before you go forward and solve this, I would always recommend processing the constraints. Removing all strongly and weakly redundant constraints and rescaling them leads to significant performance increases and robustifying the numerical stability. In MPO, processing the constraints is a simple task.

```
prog.process_constraints()
```

This results in the following (identical) multiparametric optimization problem. We were able to remove 2 constraints! And we reduced the condition number of the constraints.

$$\min_{x} \frac{1}{2} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}^T \begin{bmatrix} 306.0 & 0 & 0 & 0 \\ 0 & 324.0 & 0 & 0 \\ 0 & 0 & 324.0 & 0 \\ 0 & 0 & 0 & 252.0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 25.0 \\ 25.0 \\ 25.0 \\ 25.0 \end{bmatrix}^T \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\text{s.t.} \begin{bmatrix} 0.7071 & 0.7071 & 0 & 0 \\ 0 & 0 & 0.7071 & 0.7071 \\ -0.5774 & 0 & -0.5774 & 0 \\ 0 & -0.5774 & 0 & -0.5774 \\ -1.0 & 0 & 0 & 0 \\ 0 & -1.0 & 0 & 0 \\ 0 & 0 & -1.0 & 0 \\ 0 & 0 & 0 & -1.0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \leq \begin{bmatrix} 247.5 \\ 424.3 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ -0.5774 & 0 \\ 0 & -0.5774 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$$

$$\begin{bmatrix} 1.0 & 0 \\ 0 & 1.0 \\ -1.0 & 0 \\ 0 & -1.0 \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} \leq \begin{bmatrix} 1e+03 \\ 1e+03 \\ 0 \\ 0 \end{bmatrix}$$

0.7071
00 0
0.7071 0
0 0.7071 − 0.5774
00 −0.5774
0 −0.5774
0 −0.5774 − 1.0
00 0
0 −1.0
0 00
00 −1.0
0 0
−1.0

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \leq \begin{bmatrix} 247.5 \\ 424.3 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ -0.5774 & 0 \\ 0 & -0.5774 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$$

$$\begin{bmatrix} 1.0 & 0 \\ 0 & 1.0 \\ -1.0 & 0 \\ 0 & -1.0 \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} \leq \begin{bmatrix} 1e+03 \\ 1e+03 \\ 0 \\ 0 \end{bmatrix}$$

That wasn't that bad, and we were able to cut away some constraints that didn't matter in the process! Now we are ready to solve it.

```
solution = mpo.solve(prog)
```

Now we have the solution, we can either export the solution via the micropop module, or we can plot it. Let's plot it here. The extra arguments mean we are saving a picture of the plot and displaying it to the user (you can give a file path, so it saves somewhere that is not the current working directory).

```
parametric_plot(solution, 'transport.png' , show = True)
```

CHAPTER

THREE

API

3.1 mpo package

3.1.1 Subpackages

mpo.geometry package

Submodules

mpo.geometry.polytope module

mpo.geometry.polytope_operations module

mpo.geometry.polytope_operations.**get_chebyshev_information**(*region:* mpo.critical_region.CriticalRegion, *deterministic_solver='glpk'*)

mpo.geometry.polytope_operations.**get_vertices**(*region:* mpo.critical_region.CriticalRegion, *deterministic_solver='glpk'*)

mpo.geometry.polytope_operations.**sample_region_convex_combination**(*region:* mpo.critical_region.CriticalRegion, *dispersion=0*, *num_samples: int = 100*, *deterministic_solver='glpk'*)

Module contents

mpo.mp_solvers package

Submodules

mpo.mp_solvers.mpqp_ahmadi module

mpo.mp_solvers.mpqp_combinatorial module

class mpo.mp_solvers.mpqp_combinatorial.**CombinationTester**(*combos: List[List[int]] = <factory>*)

Bases: object

This keeps track of all of the infeasible active set combinations and filters prospective active set combinations

add(*active_set: List[int]*) → None

Added an infeasible active set to keep track of so we can cull later

check(*active_set: List[int]*) → bool

Checks if the provided active set combination is a superset of a previously tested infeasible active set :param active_set: :return: False if it should be culled and not tested any further, True if the set could be feasible

combos: List[List[int]]

mpo.mp_solvers.mpqp_combinatorial.**check_child_feasibility**(*program: mpo.mp_program.MPQP_Program, set_list: List[List[int]], combination_checker: mpo.mp_solvers.mpqp_combinatorial.Combinat*) → List[List[int]]

Checks the feasibility of a list of active set combinations, if infeasible add to the combination checker and returns all feasible active set combinations

Parameters

- **program** – An MPQP Program
- **set_list** – The list of active sets
- **combination_checker** – The combination checker that prunes

Returns The list of all feasible active sets

mpo.mp_solvers.mpqp_combinatorial.**generate_children**(*program_active_set: List[int], num_constraints: int, super_set_checker: mpo.mp_solvers.mpqp_combinatorial.CombinationTester, root: Union[List[int], numpy.ndarray] = (), is_root: bool = False*) → List

Takes a child node and then finds all of the feasible (w.r.t. pruning list) active set combinations from this branch :param program_active_set: base active set combinations from the multiparametric program :param num_constraints: number of constraints from the multi parametric program :param super_set_checker: the pruning list that will check and remove all provably infeasible child sets :param root: the base active set of the branch :param is_root: if this root is not active :return: returns all the possibly feasible children of this active set

mpo.mp_solvers.mpqp_combinatorial.**solve**(*program: mpo.mp_program.MPQP_Program, solver='gurobi'*) → *mpo.solution.Solution*

Solves the MPQP program with a modified algorithm described in Gupta et. al. 2011

url: https://www.sciencedirect.com/science/article/pii/S0005109811003190

Parameters

- **program** – MPQP to be solved
- **solver** – Deterministic solver to use

Returns the solution of the MPQP

mpo.mp_solvers.mpqp_geometric module

`mpo.mp_solvers.mpqp_geometric.`**`solve`**(*program: mpo.mp_program.MPQP_Program*) → *mpo.solution.Solution*

mpo.mp_solvers.mpqp_graph module

`mpo.mp_solvers.mpqp_graph.`**`generate_extra`**(*candidate: tuple, expansion_set, attempted: set, murder_list: settrie.SetTrie*) → list

`mpo.mp_solvers.mpqp_graph.`**`generate_reduce`**(*candidate: tuple, attempted: set, murder_list: settrie.SetTrie*) → list

`mpo.mp_solvers.mpqp_graph.`**`solve`**(*program: mpo.mp_program.MPQP_Program*) → *mpo.solution.Solution*

Solves the MPQP program with a modified algorithm described in Oberdieck et. al. 2016

url: https://www.sciencedirect.com/science/article/pii/S0005109816303971

Parameters **`program`** – MPQP to be solved

Returns the solution of the MPQP

mpo.mp_solvers.mpqp_parrallel_combinatorial module

`class` `mpo.mp_solvers.mpqp_parrallel_combinatorial.`**`CombinationTester`**

Bases: `object`

This keeps track of all of the infeasible active set combinations and filters prospective active set combinations

`add_combos`(*set_list: Set[Tuple[int]]*) → None

`check`(*active_set: Set[int]*) → bool

Checks if the provided active set combination is a superset of a previously tested infeasible active set :param active_set: :return: False if it should be culled and not tested any further, True if the set could be feasible

`mpo.mp_solvers.mpqp_parrallel_combinatorial.`**`full_process`**(*program: mpo.mp_program.MPQP_Program, active_set: List[int], murder_list, gen_children*)

`mpo.mp_solvers.mpqp_parrallel_combinatorial.`**`generate_children`**(*program_active_set: List[int], num_constraints: int, super_set_checker: mpo.mp_solvers.mpqp_parrallel_combina root: Union[List[int], numpy.ndarray] = (), is_root: bool = False*) → List

Takes a child node and then finds all of the feasible (w.r.t. pruning list) active set combinations from this branch :param program_active_set: base active set combinations from the multiparametric program :param num_constraints: number of constraints from the multi parametric program :param super_set_checker: the pruning list that will check and remove all provably infeasible child sets :param root: the base active set of the branch :param is_root: if this root is not active :return: returns all the possibly feasible children of this active set

`mpo.mp_solvers.mpqp_parrallel_combinatorial.`**`is_feasible`**(*program: mpo.mp_program.MPQP_Program, active_set: List[int]*) → bool

`mpo.mp_solvers.mpqp_parrallel_combinatorial.`**`is_optimal`**(*program: mpo.mp_program.MPQP_Program, active_set: List[int]*) → bool

`mpo.mp_solvers.mpqp_parrallel_combinatorial.`**`solve`**(*program: mpo.mp_program.MPQP_Program, num_cores=- 1*) → *mpo.solution.Solution*

Solves the MPQP program with a modified algorithm described in Gupta et. al. 2011

This is the parallel version of the combinatorial.

url: https://www.sciencedirect.com/science/article/pii/S0005109811003190

Parameters

- **num_cores** – Sets the number of cores that are allocated to run this algorithm
- **program** – MPQP to be solved

Returns the solution of the MPQP

mpo.mp_solvers.mpqp_parrallel_combinatorial_exp module

mpo.mp_solvers.mpqp_parrallel_graph module

`mpo.mp_solvers.mpqp_parrallel_graph.`**`solve`**(*program: mpo.mp_program.MPQP_Program, num_cores=- 1*) → *mpo.solution.Solution*

Solves the MPQP program with a modified algorithm described in Oberdieck et. al. 2016

url: https://www.sciencedirect.com/science/article/pii/S0005109816303971

Parameters

- **program** – MPQP to be solved
- **num_cores** – specifies numbers of cores to run, default is set to run on all available cores

Returns the solution of the MPQP

mpo.mp_solvers.solve_mplp module

class `mpo.mp_solvers.solve_mplp.`**`mplp_solver`**(*value*)

Bases: `enum.Enum`

An enumeration.

`Dustin = '1'`

`mpo.mp_solvers.solve_mplp.`**`solve_mplp`**(*problem: mpo.mp_program.MPLP_Program, algorithm: mpo.mp_solvers.solve_mplp.mplp_solver = <mplp_solver.Dustin: '1'>*)

mpo.mp_solvers.solve_mpqp module

`mpo.mp_solvers.solve_mpqp.`**`filter_solution`**(*solution:* mpo.solution.Solution) → *mpo.solution.Solution*

class `mpo.mp_solvers.solve_mpqp.`**`mpqp_algorithm`**(*value*)

Bases: `enum.Enum`

Enum that selects algorithm to be used

graph = '5'

gupta = '1'

gupta_parallel = '2'

space = '6'

step = '7'

`mpo.mp_solvers.solve_mpqp.`**`solve_mpqp`**(*problem: mpo.mp_program.MPQP_Program, algorithm: mpo.mp_solvers.solve_mpqp.mpqp_algorithm = <mpqp_algorithm.gupta: '1'>*) → *mpo.solution.Solution*

Takes a mpqp programming problem and solves it in a specified manner

default behavior is the algorithm from Gupta et al.

Using mpqp_algorithm as the algorithm selector

mpqp_algorithm.gupta => Gupta et al. Algorithm

Parameters

- **algorithm** –
- **problem** – MPQP to be solved

Returns the solution of the MPQP

mpo.mp_solvers.solver_utils module

Module contents

mpo.solver_interface package

Submodules

mpo.solver_interface.cvxopt_interface module

`mpo.solver_interface.cvxopt_interface.`**`solve_fully_constraints`**(*c: numpy.ndarray, A: numpy.ndarray, b: numpy.ndarray, equality_constraints=()*) → Optional[*mpo.solver_interface.solver_utils.So* mpo.solver_interface.cvxopt_interface.**solve_lp_cvxopt**(*c: numpy.ndarray*, *A: numpy.ndarray*, *b: numpy.ndarray*, *equality_constraints=None*, *verbose=False*, *get_duals=True*, *cvx_solver='glpk'*) → Optional[*mpo.solver_interface.solver_utils.SolverOutput*]

mpo.solver_interface.gurobi_solver_interface module

mpo.solver_interface.gurobi_solver_interface.**solve_lp_gurobi**(*c: numpy.ndarray*, *A: numpy.ndarray*, *b: numpy.ndarray*, *equality_constraints=None*, *verbose=False*, *get_duals=True*) → Optional[*mpo.solver_interface.solver_utils.Sol*

This is the breakout for solving mixed integer linear programs with gruobi, This is feed directly into the MIQP solver that is defined in the same file.

The Mixed Integer Linear program programming problem min_{xy} c^T*[xy]

s.t. A[xy] <= b Aeq*[xy] = beq

xy is the parameter vector of mixed real and binary inputs

Parameters

- **c** – Column Vector, can be None
- **A** – Constraint LHS matrix, can be None
- **b** – Constraint RHS matrix, can be None
- **equality_constraints** – List of Equality constraints
- **verbose** – Flag for output of underlying solver, default False
- **get_duals** – Flag for returning dual variable of problem, default True

Returns A dictionary of the solver outputs, or none if infeasible or unbounded. output['sol'] = primal variables, output['dual'] = dual variables, output['obj'] = objective value, output['const'] = slacks, output['active'] = active constraints.

`mpo.solver_interface.gurobi_solver_interface.`**`solve_milp_gurobi`**`(`*c: numpy.ndarray, A: numpy.ndarray, b: numpy.ndarray, equality_constraints: Optional[Iterable[int]] = None, bin_vars: Optional[Iterable[int]] = None, verbose=False, get_duals=True*`)` → Optional[*mpo.solver_interface.solver_utils.S*

This is the breakout for solving mixed integer linear programs with gruobi, This is feed directly into the MIQP solver that is defined in the same file.

The Mixed Integer Linear program programming problem min_{xy} c^T*[xy]

s.t. A[xy] <= b Aeq*[xy] = beq

xy is the parameter vector of mixed real and binary inputs

Parameters

- **c** – Column Vector, can be None
- **A** – Constraint LHS matrix, can be None
- **b** – Constraint RHS matrix, can be None
- **equality_constraints** – List of Equality constraints
- **bin_vars** – List of binary variable indices
- **verbose** – Flag for output of underlying solver, default False
- **get_duals** – Flag for returning dual variable of problem, default True (false for all mixed integer models)

Returns A dictionary of the solver outputs, or none if infeasible or unbounded. output['sol'] = primal variables, output['dual'] = dual variables, output['obj'] = objective value, output['const'] = slacks, output['active'] = active constraints.

`mpo.solver_interface.gurobi_solver_interface.`**`solve_miqp_gurobi`**`(`*Q: numpy.ndarray*, *c: numpy.ndarray*, *A: numpy.ndarray*, *b: numpy.ndarray*, *equality_constraints: Optional[Iterable[int]] = None*, *bin_vars: Optional[Iterable[int]] = None*, *verbose: bool = False*, *get_duals: bool = True*`)` → Optional[*mpo.solver_interface.solver_utils.S*

This is the breakout for solving mixed integer quadratic programs with gruobi

The Mixed Integer Quadratic program programming problem min_{xy} 1/2 [xy]^T*Q*[xy] + c^T*[xy]

s.t. A[xy] <= b Aeq*[xy] = beq

xy is the parameter vector of mixed real and binary inputs

Parameters

- **Q** – Square matrix, can be None
- **c** – Column Vector, can be None
- **A** – Constraint LHS matrix, can be None
- **b** – Constraint RHS matrix, can be None
- **equality_constraints** – List of Equality constraints
- **bin_vars** – List of binary variable indices
- **verbose** – Flag for output of underlying solver, default False
- **get_duals** – Flag for returning dual variable of problem, default True (false for all mixed integer models)

Returns A dictionary of the solver outputs, or none if infeasible or unbounded. n output['sol'] = primal

variables, output['dual'] = dual variables, output['obj'] = objective value, output['const'] = slacks, output['active'] = active constraints.

`mpo.solver_interface.gurobi_solver_interface.`**`solve_qp_gurobi`**(*Q: numpy.ndarray, c: numpy.ndarray, A: numpy.ndarray, b: numpy.ndarray, equality_constraints: Optional[Iterable[int]] = None, verbose=False, get_duals=True*) → Optional[*mpo.solver_interface.solver_utils.Sol*

This is the breakout for solving mixed integer quadratic programs with gruobi

The Mixed Integer Quadratic program programming problem min_{xy} 1/2 [xy]^T*Q*[xy] + c^T*[xy]

s.t. A[xy] <= b Aeq*[xy] = beq

xy is the parameter vector of mixed real and binary inputs

Parameters

- **Q** – Square matrix, can be None
- **c** – Column Vector, can be None
- **A** – Constraint LHS matrix, can be None
- **b** – Constraint RHS matrix, can be None
- **equality_constraints** – List of Equality constraints
- **verbose** – Flag for output of underlying solver, default False
- **get_duals** – Flag for returning dual variable of problem, default True (false for all mixed integer models)

Returns A dictionary of the solver outputs, or none if infeasible or unbounded. n output['sol'] = primal variables, output['dual'] = dual variables, output['obj'] = objective value, output['const'] = slacks, output['active'] = active constraints.

mpo.solver_interface.solver_interface module

`mpo.solver_interface.solver_interface.`**`solve_lp`**(*c: Optional[numpy.ndarray], A: Optional[numpy.ndarray], b: Optional[numpy.ndarray], equality_constraints=None, verbose=False, get_duals=True, deterministic_solver='glpk'*) → Optional[*mpo.solver_interface.solver_utils.SolverOutput*]

This is the breakout for solving mixed integer linear programs with gruobi, This is feed directly into the MIQP solver that is defined in the same file.

The Mixed Integer Linear program programming problem min_{xy} c^T*[xy]

s.t. A[xy] <= b Aeq*[xy] = beq

xy is the parameter vector of mixed real and binary inputs

Parameters

- **c** – Column Vector, can be None
- **A** – Constraint LHS matrix, can be None
- **b** – Constraint RHS matrix, can be None
- **equality_constraints** – List of Equality constraints
- **verbose** – Flag for output of underlying solver, default False
- **get_duals** – Flag for returning dual variable of problem, default True
- **deterministic_solver** – The underlying solver to use, eg. gurobi, ect

Returns A dictionary of the solver outputs, or none if infeasible or unbounded. output['sol'] = primal variables, output['dual'] = dual variables, output['obj'] = objective value, output['const'] = slacks, output['active'] = active constraints.

`mpo.solver_interface.solver_interface.`**`solve_milp`**(*c: Optional[numpy.ndarray], A: Optional[numpy.ndarray], b: Optional[numpy.ndarray], equality_constraints: Optional[Iterable[int]] = None, bin_vars: Optional[Iterable[int]] = None, verbose=False, get_duals=True, deterministic_solver='gurobi'*) → Optional[*mpo.solver_interface.solver_utils.SolverOutput*]

This is the breakout for solving mixed integer linear programs with gruobi, This is feed directly into the MIQP solver that is defined in the same file.

The Mixed Integer Linear program programming problem min_{xy} c^T*[xy]

s.t. A[xy] <= b Aeq*[xy] = beq

xy is the parameter vector of mixed real and binary inputs

Parameters

- **c** – Column Vector, can be None
- **A** – Constraint LHS matrix, can be None
- **b** – Constraint RHS matrix, can be None
- **equality_constraints** – List of Equality constraints
- **bin_vars** – List of binary variable indices
- **verbose** – Flag for output of underlying solver, default False
- **get_duals** – Flag for returning dual variable of problem, default True (false for all mixed integer models)
- **deterministic_solver** – The underlying solver to use, eg. gurobi, ect

Returns A dictionary of the solver outputs, or none if infeasible or unbounded. output['sol'] = primal variables, output['dual'] = dual variables, output['obj'] = objective value, output['const'] = slacks, output['active'] = active constraints.

mpo.solver_interface.solver_interface.**solve_miqp**(*Q: Optional[numpy.ndarray], c: Optional[numpy.ndarray], A: Optional[numpy.ndarray], b: Optional[numpy.ndarray], equality_constraints: Iterable[int] = (), bin_vars: Iterable[int] = (), verbose: bool = False, get_duals: bool = True, deterministic_solver='gurobi'*) → Optional[*mpo.solver_interface.solver_utils.SolverOutput*]

This is the breakout for solving mixed integer quadratic programs with gruobi

The Mixed Integer Quadratic program programming problem min_{xy} 1/2 [xy]^T*Q*[xy] + c^T*[xy]

s.t. A[xy] <= b Aeq*[xy] = beq

xy is the parameter vector of mixed real and binary inputs

Parameters

- **Q** – Square matrix, can be None
- **c** – Column Vector, can be None
- **A** – Constraint LHS matrix, can be None
- **b** – Constraint RHS matrix, can be None
- **equality_constraints** – List of Equality constraints
- **bin_vars** – List of binary variable indices
- **verbose** – Flag for output of underlying solver, default False
- **get_duals** – Flag for returning dual variable of problem, default True (false for all mixed integer models)
- **deterministic_solver** – The underlying solver to use, eg. gurobi, ect

Returns A dictionary of the solver outputs, or none if infeasible or unbounded. n output['sol'] = primal

variables, output['dual'] = dual variables, output['obj'] = objective value, output['const'] = slacks, output['active'] = active constraints.

mpo.solver_interface.solver_interface.**solve_qp**(*Q: Optional[numpy.ndarray], c: Optional[numpy.ndarray], A: Optional[numpy.ndarray], b: Optional[numpy.ndarray], equality_constraints: Optional[Iterable[int]] = None, verbose=False, get_duals=True, deterministic_solver='gurobi'*) → Optional[*mpo.solver_interface.solver_utils.SolverOutput*]

This is the breakout for solving mixed integer quadratic programs with gruobi

The Mixed Integer Quadratic program programming problem min_{xy} 1/2 [xy]^T*Q*[xy] + c^T*[xy]

s.t. A[xy] <= b Aeq*[xy] = beq

xy is the parameter vector of mixed real and binary inputs

Parameters

- **Q** – Square matrix, can be None
- **c** – Column Vector, can be None
- **A** – Constraint LHS matrix, can be None
- **b** – Constraint RHS matrix, can be None
- **equality_constraints** – List of Equality constraints
- **verbose** – Flag for output of underlying solver, default False
- **get_duals** – Flag for returning dual variable of problem, default True (false for all mixed integer models)
- **deterministic_solver** – The underlying solver to use, eg. gurobi, ect

Returns A dictionary of the solver outputs, or none if infeasible or unbounded. n output['sol'] = primal variables, output['dual'] = dual variables, output['obj'] = objective value, output['const'] = slacks, output['active'] = active constraints.

mpo.solver_interface.solver_interface.**solver_not_supported**(*solver_name: str*) → None

This is an internal method that throws an error and prompts the user when they use an unsupported solver

mpo.solver_interface.solver_utils module

class mpo.solver_interface.solver_utils.**SolverOutput**(*obj: float, sol: numpy.ndarray, slack: Optional[numpy.ndarray], active_set: Optional[numpy.ndarray], dual: Optional[numpy.ndarray]*)

Bases: object

Solver information object

Members: obj: objective value of the optimal solution

sol: x*, numpy.ndarray

Optional Parameters -> None or numpy.ndarray type

slack: the slacks associated with every constraint

active_set: the active set of the solution, including strongly and weakly active constraints

dual: the lagrange multipliers associated with the problem

active_set: Optional[numpy.ndarray]

dual: Optional[numpy.ndarray]

obj: float

slack: Optional[numpy.ndarray]

sol: numpy.ndarray

`mpo.solver_interface.solver_utils.`**`get_program_parameters`**(*Q: Optional[numpy.ndarray]*, *c: Optional[numpy.ndarray]*, *A: Optional[numpy.ndarray]*, *b: Optional[numpy.ndarray]*)

Given a set of possibly None optimization parameters determine the number of variables and constraints

Module contents

mpo.upop package

Submodules

mpo.upop.language_generation module

`mpo.upop.language_generation.`**`gen_array`**(*data: list*, *name: str*, *vartype: str*, *options=('const')*, *lang='cpp'*) → str

`mpo.upop.language_generation.`**`gen_cpp_array`**(*data: list*, *name: str*, *vartype: str*, *options: list = ('const')*) → str

`mpo.upop.language_generation.`**`gen_cpp_variable`**(*data*, *name: str*, *vartype: str*, *options: list = ('const')*) → str

`mpo.upop.language_generation.`**`gen_js_array`**(*data: list*, *name: str*, *vartype: str*, *options: list = ('const')*) → str

`mpo.upop.language_generation.`**`gen_js_variable`**(*data*, *name: str*, *vartype: str*, *options: list = ('const')*) → str

`mpo.upop.language_generation.`**`gen_python_array`**(*data: list*, *name: str*, *vartype: str*, *options: list = ('const')*) → str

`mpo.upop.language_generation.`**`gen_python_variable`**(*data*, *name: str*, *vartype: str*, *options: list = ('const')*) → str

`mpo.upop.language_generation.`**`gen_variable`**(*data*, *name: str*, *vartype: str*, *options=('const')*, *lang='cpp'*) → str

mpo.upop.linear_code_gen module

mpo.upop.point_location module

`class` `mpo.upop.point_location.`**`PointLocation`**(*solution:* mpo.solution.Solution)

Bases: `object`

`evaluate`(*theta: numpy.ndarray*) → Optional[numpy.ndarray]

Evaluates the value of x(theta), of the

Parameters **`theta`** –

Returns

`is_inside`(*theta: numpy.ndarray*) → bool

Determines if the theta point in inside of the feasible space

param theta A point in the theta space

return True, if theta in region

False, if theta not in region

locate(*theta: numpy.ndarray*) → int

Finds the index of the critical region that theta is inside

Parameters **theta** –

Returns

mpo.upop.ucontroller module

class mpo.upop.ucontroller.**BVH**(*parent*, *fundamental_list*, *region_list*, *depth*, *index*)

Bases: object

mpo.upop.ucontroller.**classify_polytope**(*region:* mpo.critical_region.CriticalRegion, *hyper_plane: numpy.ndarray*) → int

We are going to classify the polytopic critical region by solving 2 LPS

max ||<x,A>||-d for x in Critical region

min ||<x,A>||-d for x in Critical region

The result of the objective function will tell us the side of the hyper plane the point is on

Parameters

- **region** – Critical region
- **hyper_plane** – A fundamental hyperplane

Returns -1 if completely not in support, 0 if intersected, 1 if completely in support

mpo.upop.ucontroller.**determine_hyperplane**(*regions: List[*mpo.critical_region.CriticalRegion*]*, *hyper_planes: numpy.ndarray*)

Finds the 'best' splitting hyper plane for this task

In this case best means minimizing the number of intersected regions while also maximizing the difference between supported and not supported regions

Parameters

- **regions** –
- **hyper_planes** –

Returns []

mpo.upop.ucontroller.**generate_code**(*solution:* mpo.solution.Solution) → List[str]

Generates C++17 code for point location and function evaluation on microcontrollers

This forms a BVH to accelerate solution times

WARNING: This breaks down at high dimensions

Parameters **solution** – a solution to a MPLP or MPQP solution

Returns List of the strings of the C++17 datafiles that integrate with uPOP

mpo.upop.upop_utils module

`mpo.upop.upop_utils.`**`find_unique_hyperplanes`**`(`*overall: numpy.ndarray*`)` → Tuple[List[int], List[int], List[int]]

Generates the list of indices of the fundamental hyperplanes of the solution, as well as the indices of the associated hyperplanes from the original solution and the parity of the constraint

This is linear w.r.t. number of hyper planes and is quite quick ~25 ns per constraint in the solution

It first creates approximate(near exact) integer representations for each constraint for each region in the solution

This approximation step is justified in that it will find equality between 2 constraints if the L2 norm of the difference is below 10E-12

Then the positive and negative versions of these constraints [-x < -1, x < 1 are on different sides of the same hyperplane] are made into a format that can be hashed (tuples of ints)

With this is is relatively strait forward to check for uniqueness with the set

The first loop scans thru all of the constraints and if the constraint contains a unique hyperplane

1) if it is a unique hyper plane store the index, add the integer representation to the set, then index the integer representation to the index
2) if it is not a unique hyperplane do nothing

The second loop scans thru the constraints again and assigns them unique hyper plane indices and the parity(what side of the hyper plane that they are on)

Parameters **`overall`** – The solution of a multiparametric programming problem

Returns returns indices of fundamental hyperplanes, indices of constraints back to fundamental hyperplane, parity of constraint

`mpo.upop.upop_utils.`**`find_unique_region_functions`**`(`*solution:* mpo.solution.Solution`)` → Tuple[List[int], List[int], List[int]]

`mpo.upop.upop_utils.`**`find_unique_region_hyperplanes`**`(`*solution:* mpo.solution.Solution`)` → Tuple[List[int], List[int], List[int]]

This is an overload of the find_unique_hyperplane function

Parameters **`solution`** –

Returns

`mpo.upop.upop_utils.`**`get_chebychev_centers`**`(`*solution:* mpo.solution.Solution`)` → List[numpy.ndarray]

Calculates and returns a list of all of the theta chebychev centers for the critical regions in the solution

Parameters **`solution`** – An mp programming Solution

Returns A list of all of the chebychev centers of the regions in the solutions

`mpo.upop.upop_utils.`**`get_descriptions`**`(`*solution:* mpo.solution.Solution`)` → dict

`mpo.upop.upop_utils.`**`get_outer_boundaries`**`(`*indices: List[int]*, *parity: List[int]*`)`

Takes in the global constraint indices to the fundamental hyperplanes and their parity finds all planes with only one parity version aka only one verity of them appears in the original set.

This method is linear w.r.t. number of indices, by the use of sets and hash maps

Parameters

- **`indices`** – list of indices that maps the solution constraints into the fundamental hyperplanes

- **parity** – the side of the hyperplane that the constraint represents

Returns

mpo.upop.upop_utils.**verify_outer_boundary**(*solution: mpo.solution.Solution*, *hyper_indices: List[int]*, *outer_indices: List[int]*, *chebychev_centers: Optional[List[numpy.ndarray]] = None*) → List[int]

This checks all of the possible outer boundary indices for errors, failures to solve for the minimal set of fundamental hyperplanes in the solution

Parameters

- **solution** – An mp programming solution
- **hyper_indices** – The list of all fundamental hyperplane indices
- **outer_indices** – The list of identified exterior hyperplane indices
- **chebychev_centers** – the list of chebychev centers in the theta space for every critical region {Optional}

Returns List of verified outer boundary constraints

Module contents

mpo.utils package

Submodules

mpo.utils.chebyshev_ball module

mpo.utils.chebyshev_ball.**chebyshev_ball**(*A: numpy.ndarray*, *b: numpy.ndarray*, *equality_constraints: Optional[Iterable[int]] = None*, *bin_vars: Iterable[int] = ()*, *deterministic_solver='glpk'*)

Chebyshev ball finds the largest ball inside of a polytope defined by Ax <= b This is solved by the following LP

min{x,r} -r

st: Ax + ||A_i||r <= b

A_{eq}*x = b_{eq}

r >=0

Returns a List with [pos, r] where pos is a numpy array r is a real number

Parameters

- **A** – LHS Constraint Matrix
- **b** – RHS Constraint column vector
- **equality_constraints** – indices of

rows that have strict equality A[eq] @ x = b[eq] :param bin_vars: indices of binary variables :param deterministic_solver: The underlying solver to use, eg. gurobi, ect :return: the optimization output of the LP problem, the coordinates can be found in output['sol'], with output['sol'][-1] giving the chebyshev radius

mpo.utils.constraint_utilities module

`mpo.utils.constraint_utilities.`**`calculate_redundant_constraints`**(*A*, *b*)

Removes weakly redundant constraints, method is from the appendix of the Oberdieck paper

url: https://www.sciencedirect.com/science/article/pii/S0005109816303971

Parameters

- **A** – LHS constraint matrix
- **b** – RHS constraint column vector

Returns The processes constraint pair [A, b]

`mpo.utils.constraint_utilities.`**`cheap_remove_redundant_constraints`**(*A: numpy.ndarray*, *b: numpy.ndarray*) → List[numpy.ndarray]

Removes zero rows, normalizes the constraint rows to ||A_i||_{L_2} = 1, and removes duplicate rows

Parameters

- **A** – LHS constraint matrix
- **b** – RHS constraint column vector

Returns The processes constraint pair [A, b]

`mpo.utils.constraint_utilities.`**`constraint_norm`**(*A: numpy.ndarray*) → numpy.ndarray

Finds the L2 norm of each row of a matrix

Parameters **A** – numpy matrix

Returns A column vector of the row norms

`mpo.utils.constraint_utilities.`**`facet_ball_elimination`**(*A: numpy.ndarray*, *b: numpy.ndarray*) → List[numpy.ndarray]

Removes weakly redundant constraints, method is from the appendix of the Oberdieck paper

url: https://www.sciencedirect.com/science/article/pii/S0005109816303971

Parameters

- **A** – LHS constraint matrix
- **b** – RHS constraint column vector

Returns The processes constraint pair [A, b]

`mpo.utils.constraint_utilities.`**`is_full_rank`**(*A: numpy.ndarray*, *indices: Optional[List[int]] = None*) → bool

Tests if the matrix A[indices] is full rank Empty matrices e.g. A[[]] will default to be full rank

Parameters

- **A** – Matrix
- **indices** – indices to consider in rank

Returns if the matrix is full rank or not mpo.utils.constraint_utilities.**process_region_constraints**(*A: numpy.ndarray*, *b: numpy.ndarray*) → List[numpy.ndarray]

Removes all strongly and weakly redundant constraints

Parameters

- **A** – LHS constraint matrix
- **b** – RHS constraint column vector

Returns The processes constraint pair [A, b]

mpo.utils.constraint_utilities.**remove_duplicate_rows**(*A: numpy.ndarray*, *b: numpy.ndarray*) → List[numpy.ndarray]

Finds and removes duplicate rows in the constraints A @ x <= b

mpo.utils.constraint_utilities.**remove_strongly_redundant_constraints**(*A: numpy.ndarray*, *b: numpy.ndarray*) → List[numpy.ndarray]

Removes strongly redundant constraints by testing the feasibility of each constraint if activated

mpo.utils.constraint_utilities.**remove_zero_rows**(*A: numpy.ndarray*, *b: numpy.ndarray*) → List[numpy.ndarray]

Finds rows equal to zero in A and then removes them from A and b

Parameters

- **A** – LHS Matrix constraint
- **b** – RHS Column vector

Returns a list[A_cleaned, b_cleaned] of filtered constraints

mpo.utils.constraint_utilities.**row_equality**(*row_1: numpy.ndarray*, *row_2: numpy.ndarray*, *tol=1e-16*) → bool

Tests if 2 row vectors are approximately equal

Parameters

- **row_1** –
- **row_2** –
- **tol** – tolerable L2 norm of the difference

Returns True if rows are equal

mpo.utils.constraint_utilities.**scale_constraint**(*A: numpy.ndarray*, *b: numpy.ndarray*) → List[numpy.ndarray]

Normalizes constraints

Parameters

- **A** – LHS Matrix constraint
- **b** – RHS column vector constraint

Returns a list [A_scaled, b_scaled] of normalized constraints

mpo.utils.general_utils module

mpo.utils.general_utils.**latex_matrix**(*A: Union[List[str], numpy.ndarray]*) → str
Creates a latex string for a given numpy array

Parameters **A** – A numpy array

Returns A latex string for the matrix A

mpo.utils.general_utils.**make_column**(*x: Union[List, numpy.ndarray]*) → numpy.ndarray
Makes x into a column vector :param x: a list or a numpy array :return: a numpy array that is a column vector

mpo.utils.general_utils.**make_row**(*x: Union[List, numpy.ndarray]*) → numpy.ndarray
Makes x into a row vector :param x: a list or a numpy array :return: a numpy array that is a row column

mpo.utils.general_utils.**mpo_block**(*mat_list*)

mpo.utils.general_utils.**num_cpu_cores**()
Finds the number of allocated cores,with different behavior in windows and linux.

In Windows, returns number of physical cpu cores

In Linux, returns number of available cores for processing (this is for running on cluster or managed environment)

Returns number of cores

mpo.utils.general_utils.**remove_size_zero_matrices**(*list_matrices: List[numpy.ndarray]*) → List[numpy.ndarray]
Removes size zero matrices from a list

Parameters **list_matrices** – A list of numpy arrays

Returns returns all matrices from the list that do not have a dimension of 0 in any index

mpo.utils.general_utils.**select_not_in_list**(*A: numpy.ndarray, list_: Iterable[int]*) → numpy.ndarray
Filters a numpy array to select all rows that are not in a list

Parameters

- **A** – a numpy array
- **list** – a list of indices that you want to remove

Returns return a numpy array of A[not in list_]

mpo.utils.geometric module

mpo.utils.geometric.**gen_tess_points_simplex**(*simplex*)

Parameters **simplex** –

Returns

mpo.utils.geometric.**make_domain_subdivision**(*A_t, b_t*)

mpo.utils.geometric.**make_simplex**(*n: int*)

mpo.utils.geometric.**make_subdomains**(*points*)

mpo.utils.geometric.**revised_tess_simplex**(*simplex, half_split=False*)

mpo.utils.mpqp_utils module

`mpo.utils.mpqp_utils.`**`calculate_control_law`**(*program:* mpo.mp_program.MPQP_Program, *active_set: List[int]*) → Tuple

`mpo.utils.mpqp_utils.`**`check_feasibility`**(*program:* mpo.mp_program.MPQP_Program, *active_set*)

`mpo.utils.mpqp_utils.`**`check_optimality`**(*program:* mpo.mp_program.MPQP_Program, *active_set: list*)

Tests if the active set is optimal for the provided mpqp program

x | theta | lambda | slack | t max t

1) Qu + (A_Ai)^T lambda_Ai + c = 0
2) A_Ai*u - b_ai-F_ai*theta = 0
3) A_Aj*u - b_aj-F_aj*theta + sj_k= 0
4) t*e_1 <= lambda_Ai,
5) t*e_2 <= s_Ji
6) t >= 0,
7) lambda_Ai>= 0,
8) s_Ji>=0
9) A_t*theta<= b_t

Parameters

- **`program`** – an mpqp program
- **`active_set`** – active set being considered in the optimality test

Returns dictionary of parameters, or None if active set is not optimal

`mpo.utils.mpqp_utils.`**`gen_cr_from_active_set`**(*program:* mpo.mp_program.MPQP_Program, *active_set: List[int]*, *check_full_dim=True*) → Optional[*mpo.critical_region.CriticalRegion*]

Builds the critical region of the given mpqp from the active set.

Parameters

- **`program`** – the MQMP_Program to be solved
- **`active_set`** – the active set combination to build this critical region from
- **`check_full_dim`** – Keyword Arg, if true will return null if the region has lower dimensionality

Returns Returns the associated critical region if fully dimensional else returns None

`mpo.utils.mpqp_utils.`**`get_boundary_types`**(*region: numpy.ndarray*, *omega: numpy.ndarray*, *lagrange: numpy.ndarray*, *regular: numpy.ndarray*) → List

Classifies the boundaries of a polytope into Omega constraints, Lagrange multiplier = 0 constraints, and Activated program constraints :param region: :param omega: :param lagrange: :param regular: :return:

`mpo.utils.mpqp_utils.`**`get_feasible_theta`**(*program:* mpo.mp_program.MPQP_Program) → Union[None, numpy.ndarray] `mpo.utils.mpqp_utils.`**`get_feasible_theta_2`**(*program: mpo.mp_program.MPQP_Program*) → Union[None, numpy.ndarray]

Finds a feasible theta constraint of the multi-parametric problem

Pseudo-Code Steps:

1) Find and calculate the Theta Ball of the theta feasible space
2) See if the center of the ball is a valid theta point
3) if not retry up to {100} times to find a feasible point in the theta ball of the problem
4) If one can not be found in the theta ball returns None

Parameters **`program`** – MP Program

Returns feasible theta point or None

`mpo.utils.mpqp_utils.`**`get_region`**(*program: mpo.mp_program.MPQP_Program*) → *mpo.critical_region.CriticalRegion*

`mpo.utils.mpqp_utils.`**`is_full_dimensional`**(*A*, *b*)

`mpo.utils.mpqp_utils.`**`theta_ball`**(*program: mpo.mp_program.MPQP_Program*) → Optional[*mpo.solver_interface.solver_utils.SolverOutput*]

Finds the chebychev ball in the theta feasible space of a problem.

Parameters **`program`** – MPQP_Program

Returns basic result from the LP

`mpo.utils.mpqp_utils.`**`zeros`**(*x: int*, *y: int*) → numpy.ndarray

Auxiliary function returns a numpy array of zeros of dimensions x by y (rows, columns)

Parameters

- **`x`** – Number of rows
- **`y`** – Number of Columns

Returns Numpy array of zeros

mpo.utils.solver_utils module

`mpo.utils.solver_utils.`**`get_active_set`**(*soln: mpo.solver_interface.solver_utils.SolverOutput*) → numpy.array

finds the active set of a problem output

Parameters **`soln`** – dict results from the solver interface

Returns the active set of a solution

Module contents

3.1.2 Submodules

3.1.3 mpo.critical_region module

class mpo.critical_region.**CriticalRegion**(*A: numpy.ndarray, b: numpy.ndarray, C: numpy.ndarray, d: numpy.ndarray, E: numpy.ndarray, f: numpy.ndarray, active_set: Union[List[int], numpy.ndarray], omega_set: Union[List[int], numpy.ndarray] = <factory>, lambda_set: Union[List[int], numpy.ndarray] = <factory>, regular_set: Union[List[int], numpy.ndarray] = <factory>*)

Bases: `object`

Critical region is a polytope that defines a region in the uncertainty space with an associated optimal value, active set, lagrange multipliers and constraints

x() = A + b

() = C + d

CR := { : E <= f}

active_set: numpy array of indices

constraint_set: if this is a A@x = b + F@theta boundary

lambda_set: if this is a = 0 boundary

boundary_set: if this is a E <= f boundary

A: numpy.ndarray

C: numpy.ndarray

E: numpy.ndarray

active_set: Union[List[int], numpy.ndarray]

b: numpy.ndarray

d: numpy.ndarray

evaluate(*theta: numpy.ndarray*) → numpy.ndarray
Evaluates x() = A + b

f: numpy.ndarray

get_constraints()

is_full_dimension() → bool
Tests dimensionality of critical region

is_inside(*theta: numpy.ndarray*) → numpy.ndarray
Tests if point is inside of the critical region

lagrange_multipliers(*theta: numpy.ndarray*) → numpy.ndarray
Evaluates () = C + d

lambda_set: Union[List[int], numpy.ndarray]

omega_set: Union[List[int], numpy.ndarray]

regular_set: Union[List[int], numpy.ndarray]

3.1.4 mpo.mp_program module

class mpo.mp_program.**MPLP_Program**(*A: numpy.ndarray*, *b: numpy.ndarray*, *c: numpy.ndarray*, *A_t: numpy.ndarray*, *b_t: numpy.ndarray*, *F: numpy.ndarray*, *active_set: Union[List[int], numpy.ndarray]*)

Bases: `object`

The standard class for linear multiparametric programming

A: numpy.ndarray

A_t: numpy.ndarray

F: numpy.ndarray

active_set: Union[List[int], numpy.ndarray]

b: numpy.ndarray

b_t: numpy.ndarray

c: numpy.ndarray

display_latex() → None
Displaces Latex text of the multiparametric problem

display_warnings() → None
Displaces warnings

latex() → List[str]
Generates latex of the multiparametric problem

Returns returns latex of the

num_constraints() → int
Returns number of constraints

num_t() → int
Returns number of uncertain variables

num_x() → int
Returns number of parameters

process_constraints() → None
Removes redundant constraints from the multiparametric programming problem

scale_constraints() → None
Rescales the constraints of the multiparametric problem to ||[A|-F]||_i = 1, in the L2 sense

solve_theta(*theta_point: numpy.ndarray*, *deterministic_solver='glpk'*) → Optional[*mpo.solver_interface.solver_utils.SolverOutput*]
Substitutes theta into the multiparametric problem and solves

Parameters **theta_point** – an uncertainty realization

Returns the solver output of the substituted problem, returns None if not solvable

warnings() → List[str]
Checks the dimensions of the matrices to ensure consistency

class mpo.mp_program.**MPQP_Program**(*A: numpy.ndarray*, *b: numpy.ndarray*, *c: numpy.ndarray*, *Q: numpy.ndarray*, *A_t: numpy.ndarray*, *b_t: numpy.ndarray*, *F: numpy.ndarray*, *active_set=None*)

Bases: *mpo.mp_program.MPLP_Program*

The standard class for quadratic multiparametric programming, inherits from MPLP_Program

A: numpy.ndarray

A_t: numpy.ndarray

F: numpy.ndarray

active_set: Union[List[int], numpy.ndarray]

b: numpy.ndarray

b_t: numpy.ndarray

c: numpy.ndarray

latex() → List[str]

Creates a latex output for the multiparametric problem

process_constraints() → None

Removes redundant constraints from the multiparametric programming problem

solve_theta(*theta_point: numpy.ndarray*) → Optional[*mpo.solver_interface.solver_utils.SolverOutput*]

Substitutes theta into the multiparametric problem and solves

Parameters **theta_point** – an uncertainty realization

Returns the solver output of the substituted problem, returns None if not solvable

warnings() → List[str]

Checks the dimensions of the matrices to ensure consistency

3.1.5 mpo.plot module

mpo.plot.**gen_vertices**(*solution:* mpo.solution.Solution)

Generates the vertices associated with the mixed

Parameters **solution** – a multiparametric region

Returns a list of a collection of vertices sorted counterclockwise that correspond to the specific region

mpo.plot.**parametric_plot**(*solution:* mpo.solution.Solution, *save_path: Optional[str] = None*, *show=True*) → None

Makes a simple plot from a solution

Parameters

- **solution** – a multiparametric solution
- **save_path** – if specified saves the plot in the directory
- **show** – Keyword argument, if True displays the plot otherwise does not display

Returns no return, creates graph of solution

mpo.plot.**plotly_plot**(*solution:* mpo.solution.Solution, *save_path: Optional[str] = None*, *show=True*) → None

Makes a plot via the plotly library, this is good for interactive figures that you can embed into webpages and handle interactively

Parameters

- **solution** –
- **save_path** – Keyword argument, if a directory path is specified it will save a html copy and a png to that directory
- **show** – Keyword argument, if True displays the plot otherwise does not display

Returns

mpo.plot.**sort_clockwise**(*vertices: List[numpy.ndarray]*) → List[numpy.ndarray]

Sorts the vertices in clockwise order, fixes crossed polytopes in rendering

Parameters **vertices** –

Returns

3.1.6 mpo.problem_generator module

mpo.problem_generator.**generate_mplp**(*x: int = 2, t: int = 2, m: int = 10*) → *mpo.mp_program.MPLP_Program*

Parameters

- **x** – number of parameters
- **t** – number of uncertain variables
- **m** – number of constraints

Returns

mpo.problem_generator.**generate_mpqp**(*x: int = 2, t: int = 2, m: int = 10*) → *mpo.mp_program.MPQP_Program*

Generates a random mpqp problem with of the following characteristics

Parameters

- **x** – number of x dimensions
- **t** – number of theta dimensions
- **m** – number of constraints

Returns A random mpqp problem of the specified type

3.1.7 mpo.settings module

3.1.8 mpo.solution module

class mpo.solution.**Solution**(*program: Union[*mpo.mp_program.MPLP_Program, mpo.mp_program.MPQP_Program*], critical_regions: List[*mpo.critical_region.CriticalRegion*]*)

Bases: object

The Solution object is the output of multiparametric solvers, it contains all of the critical regions as well as holds a copy of the original problem that was solved

add_region(*region:* mpo.critical_region.CriticalRegion) → None

Adds a region to the solution

Parameters **region** – region to add to the solution

Returns None

critical_regions: List[*mpo.critical_region.CriticalRegion*]

evaluate(*theta_point: numpy.ndarray*) → Union[None, numpy.ndarray]
returns the optimal x* from the solution

Parameters **theta_point** – an uncertainty realization

Returns the calculated x* from theta

get_region(*theta_point: numpy.ndarray*) → Union[None, *mpo.critical_region.CriticalRegion*]
Find the critical region in the solution that corresponds to the theta provided

Parameters **theta_point** – an uncertainty realization

Returns the region that contains theta

program: Union[*mpo.mp_program.MPLP_Program*, *mpo.mp_program.MPQP_Program*]

verify_solution()

verify_theta(*theta_point: numpy.ndarray*) → bool
Checks that the result of the solution is consistent with theta substituted multiparametric problem

Parameters **theta_point** – an uncertainty realization

Returns True if they are the same, False if they are different

3.1.9 Module contents

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

L

M

N

O

P

R

S

T

V

W

Z