POP User Manual
Version 2.0

Richard Oberdieck, Nikolaos A. Diangelakis, Baris Burnak, Justin Katz, Styliani Avraamidou, Efstratios N. Pistikopoulos

Artie McFerrin Department of Chemical Engineering
Texas A&M University
College Station, TX, United States.

Contents

1 Introduction

In recent years, multi-parametric programming has received a growing interest, in particular due to its applicability to control and scheduling problems, as well as its capability of transferring the computational burden offline [1]. This in turn has led to the development of software solutions, most notably the MPT toolbox [2] and POP, the novel, state-of-the-art multiparametric programming software package presented in this user manual.

The software is MATLAB® based, but features links to NAG and CPLEX. It provides solution tools for multi-parametric linear, quadratic and mixed-integer programming problems, as well as a versatile problem generator and a comprehensive problem library. It is part of the PAROC framework [1], an integrated framework and software platform for the design, operational optimization and model-based control of process systems.

In addition, in version 2.0 POP has been connected to the modelling tool YALMIP [3], which enables the seamless and elegant modelling of dynamic systems.

2 General information

Disclaimer: POP is a prototype software tool, i.e. no guarantee about the correctness of the code is given. Any commercial users should kindly contact paroc@tamu.edu. Complementary to this user manual, you can find tutorial videos about the use of POP on our YouTube Channel[1] and up-to-date information on our website paroc.tamu.edu.

In general, POP considers the following optimization problem

$$\begin{aligned} z^*(\theta) = \min_{x,y} \quad & (Q\omega + H_t\theta + c)^T \omega + (Q_t\theta + c_t)^T \theta + c_c \\ \text{s.t.} \quad & Ax + Ey \leq b + F\theta \\ & A_{eq}x + E_{eq}y = b_{eq} + F_{eq}\theta \\ & x \in \mathbb{R}^n,\ y \in \{0,1\}^p,\ \omega = \left[x^T y^T\right]^T, \\ & \theta \in \Theta := \{\theta \in \mathbb{R}^q \,|CR_A\theta \leq CR_b\} \end{aligned} \tag{1}$$

where $Q \succ 0$, the matrices have appropriate dimensions and which is referred to as multi-parametric mixed-integer quadratic programming (mp-MIQP) problem, as well as its simpler counterparts, namely

- Multi-parametric linear programming (mp-LP) problems
- Multi-parametric quadratic programming (mp-QP) problems with $Q \succ 0$
- Multi-parametric mixed-integer linear programming (mp-MILP) problems

In the following, the following nomenclature is used:

Objective function: $(Q\omega + H\theta + c)^T \omega + (Q_t\theta + c_t)^T \theta$

Constraints: $Ax + Ey \leq b + F\theta$

Parameter space: $\Theta := \{\theta \in \mathbb{R}^q \,|CR_A\theta \leq CR_b\}$

[1] https://www.youtube.com/channel/UCCIHyN82C9MMwK55NS-lleg

Remark 1. Note that in the objective function, the continuous and discrete variables x and y are concatenated into the variable ω in order to simplify the formulation.

The POP toolbox features three main functionalities:

- Problem solver: problem (1) or its simpler counterparts are solved using state-of-the-art software
- Problem generator: feasible problems of arbitrary size are generated.
- Problem library: a library of mp-LP, mp-QP, mp-MILP and mp-MIQP problems which allow for benchmark analysis and algorithmic development

These features are available either directly in the command window or *via* an interface. In this manual, we will first describe the use of the interface in section 4 before we will show the use of the command window in section 5.

2.1 Installation and Citation

From the website paroc.tamu.edu, the file `installPOP.m` is available for download. Running this installer in the MATLAB® CommandWindow will initialize the installation in a folder of choice by the user. Please note the following:

- While POP can be used by only using MATLAB in-built functions (and its toolboxes), it is highly recommended that commercial solvers are used for stability reasons. In particular, POP features links to NAG or CPLEX as LP/QP solver. Hence it is suggested that the user first installs these solvers before installing POP. Note that for academic users CPLEX is available for free via the IBM Academic Initiative. In order to solve multi-parametric mixed-integer programming problems, POP offers links to MATLAB's intlinprog and CPLEX.
- If there are any software tools you would like to use in addition to NAG and CPLEX, please write us an email at paroc@tamu.edu.

In order to change the default options used by POP, please modify `OptionSet`. In order to change the options for a specific operation, the structure `options` can be handed to the appropriate functions.

If POP is used for a publication, the user is kindly asked to cite the following paper [4]:

> Oberdieck, R.; Diangelakis, N. A.; Papathanasiou, M. M.; Nascu, I.; Pistikopoulos, E. N. (2016) POP - Parametric Optimization Toolbox, Industrial and Engineering Chemistry Research, 55(33), 8979-8991.

3 Limitations

Before going into the details of POP, we would like to highlight some aspects of POP which have shown to be fragile or which have not been tested as extensively as others:

Equality constraints: The handling of equality constraints is not straightforward. While POP has ways to deal with them (we treat them as active inequality constraints), we have experienced that especially when implicit equality constraints are present, or equality constraints which do not satisfy the linear independence constraint qualification (LICQ), then there can be some problems in the code. This is subject of ongoing work and we will be working to improve this functionality as much as possible.

Comparison procedure for very small critical regions When the critical regions are very small (radius of the Chebyshev ball of less than 10^{-2}), then the comparison procedures might result in numerical errors, since the checks whether a region is 'empty' is based on the solution of certain optimization problems with a given tolerance. Unfortunately, we are not aware of an approach which would resolve this issue, as numerical tolerance is the ultimate limitation of any non-symbolic algorithm. If you have any suggestion on how to circumvent this problem, please let us know at paroc@tamu.edu.

4 Interface

The GUI is launched via the command `POP` in the command window. This will open the user interface (see Figure 2), where the user has the following options: Solver, Library or Generator.

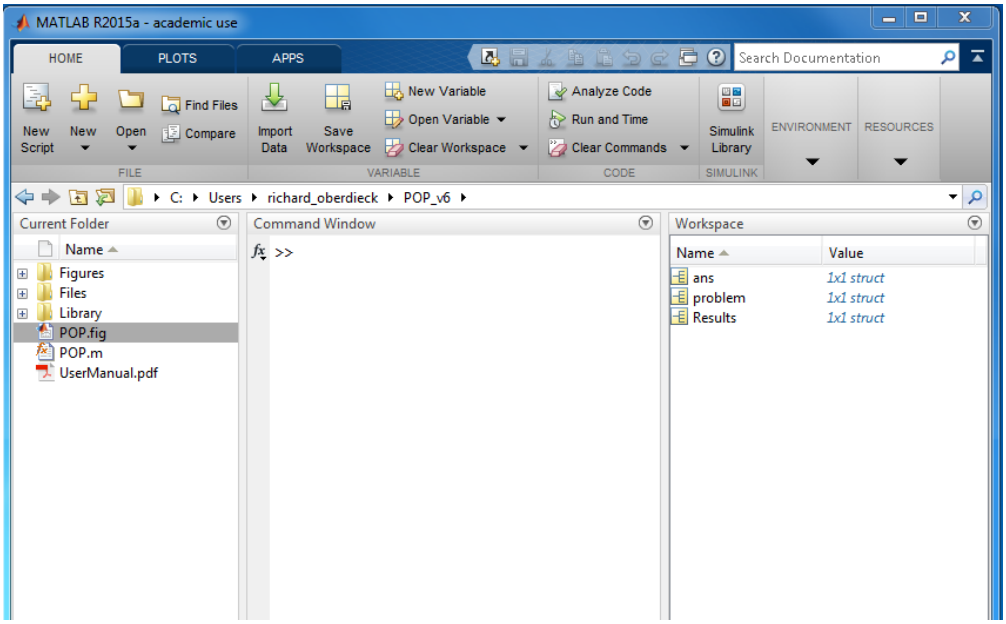

Figure 1: The starting point for the POP toolbox.

4.1 POP Solver

4.1.1 General handling

The POP solver can be accessed by either clicking on the "Solver" button on the welcome screen, or by selecting "Solver" on the top menu (see Figure 3). In order to solve a problem with POP, the problem needs to be defined in a structured array in the workspace in the following structure:

```
problem.Q    = [(n+p)x(n+p) double]        Quadratic term in OBJ
problem.Ht   = [(n+p)xq double]            Cross-term in OBJ
problem.c    = [(n+p)x1 double]            Linear term in OBJ
problem.Qt   = [qxq double]                Quadratic parametric term in OBJ
problem.ct   = [qx1 double]                Linear parametric term in OBJ
problem.cc   = [1x1 double]                Constant term in OBJ
problem.A    = [mxn double]                Constraint matrix of x
```

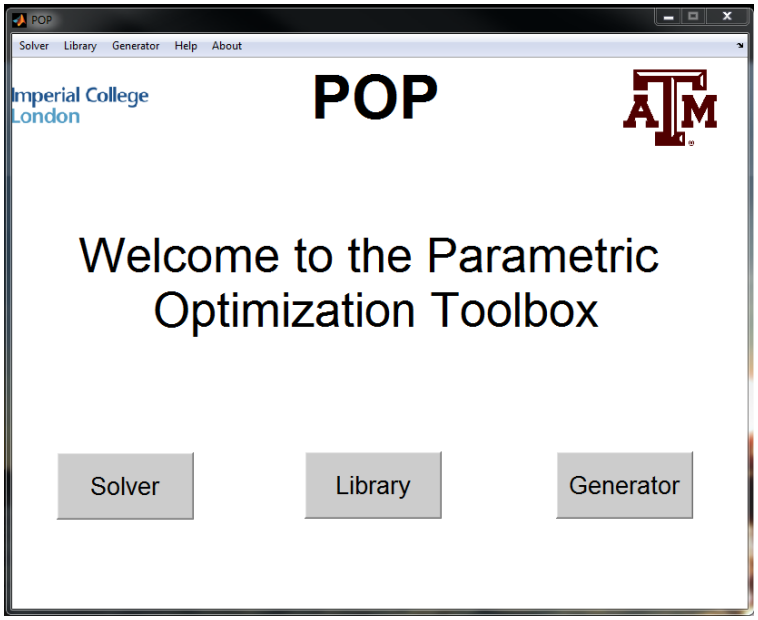

Figure 2: The welcome screen of the POP toolbox.

```
problem.E    = [mxp double]          Constraint matrix of y
problem.b    = [mx1 double]          Constant constraint term
problem.F    = [mxq double]          Constraint matrix of theta
problem.Aeq  = [wxn double]          Equality constraint matrix of x
problem.Eeq  = [wxp double]          Equality constraint matrix of y
problem.beq  = [wx1 double]          Constant equality constraint term
problem.Feq  = [wxq double]          Equality constraint matrix of theta
problem.Xmin = [nx1 double]          Lower bound for x
problem.Xmax = [nx1 double]          Upper bound for x
problem.Tmin = [qx1 double]          Lower bound for theta
problem.Tmax = [qx1 double]          Upper bound for theta
problem.CRA  = [rxq double]          Constraint matrix of parameter space
problem.CRb  = [rx1 double]          Constant term of parameter space
```

Note that the fields 'A', 'E', 'b', 'F', 'CRA' and 'CRb' need to be defined, while the other ones can be left empty and will be set to zero automatically. After the problem has been created, the problem struct[2] can then be selected from the corresponding drop-down menu. Additionally, the user can change the default options of the solver in the respective box (a list of the available options is shown in Table 1).

4.1.2 Post-processing

After the solution has been obtained, a pop-up menu appears which enables the user to perform post-processing steps (see Figure 4). Note that if you would like to access this functionality separately, you can also select the solution from the drop-down menu in the solver environment, which will enable the use of the post-processing steps.

[2]This is common MATLAB® nomenclature and refers to a structured array.

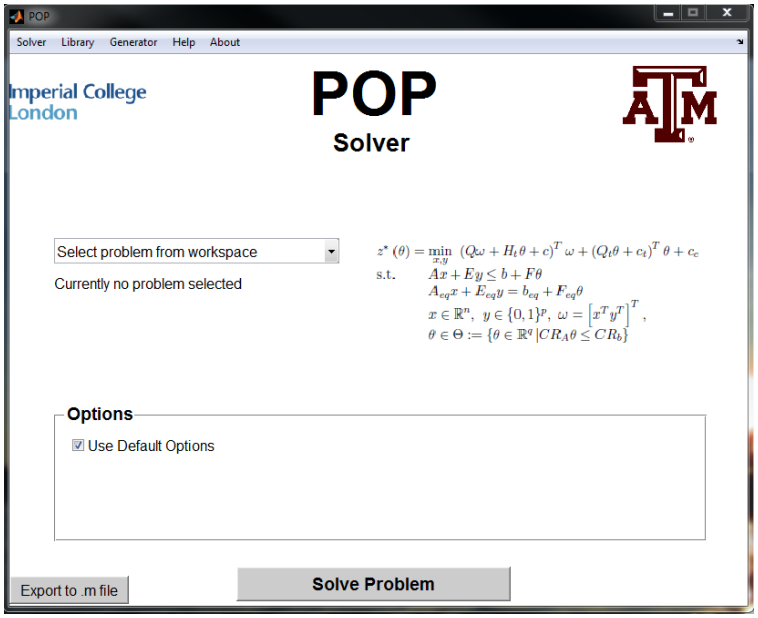

Figure 3: The POP Solver.

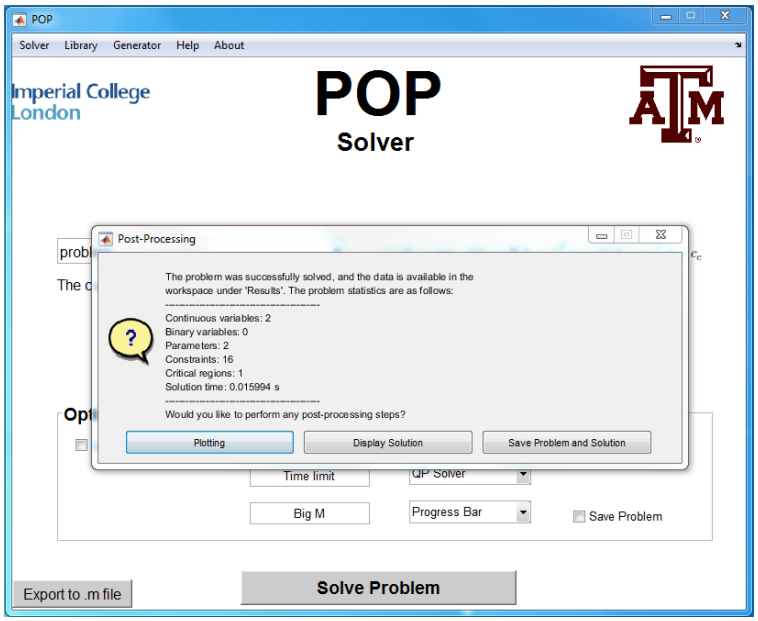

Figure 4: Post-Processing in POP.

Table 1: A list of the different options that can be specified in the GUI in POP.

Name	Default Value	Description
Tolerance	$1.4 * 10^{-8}$	Tolerance for the operations performed such as removal of redundant constraints and constraint violation
Time limit	600	Maximum time for the algorithm, in seconds
Big M	10^7	Standard bounds on variables if detected to be un-bounded. While there exist problems where this is not necessary, it increases the numerical stability of the problem
LP/QP Solver	Set at installation	Choice of LP and QP solver. The options are: 'MATLAB' [in-built functions `linprog` and `quadprog`], '`NAG`' and '`CPLEX`'.
mp-LP/QP Solver	'Graph'	Choice of mp-LP and mp-QP solver. The options are: 'Geometrical' [the geometrical algorithm presented in [5]], 'Combinatorial' [the combinatorial approach presented in [6]], 'Graph' [the connected-graph approach presented in [7]] and 'MPT' [the MPT solver, as of MPT v3.1 a combinatorial parametric linear complementarity problem solver [8]].
Progress Bar	'Off'	Enables the display of a progress bar during the computation.
Global Solver	Set at installation	Global solver used in multi-parametric mixed-integer programming. The options are: 'MATLAB' [in-built function `intlinprog`], 'CPLEX' [function `cplexmilp`], and 'Enumeration' [performs exhaustive enumeration of all possible combinations of binary variables].
Comparison	'None'	Comparison procedure used to compare parametric profiles. The options are: None [as presented in [9]] 'MinMax' [as presented in [10], 'Affine' [as presented in [11]] and 'Exact' [as presented in [12]], each of which performs the comparison of two objective functions differently.

4.1.3 Export to .m-file

This feature on the bottom left allows the user to export the current state of the GUI into an .m-file with a user-defined name. If the options have been modified from the default options, the corresponding structure for `options` is also created.

4.2 POP Library

POP features a problem library consisting of several 'test sets', each of which is contained in a separate folder in the folder 'Library'. The user can browse these sets via the provided interface (see Figure 5), and export the desired problems in order to create your own test sets or perform benchmark experiments (see Figure 6). Each entry of the library thereby features the problem formulation as well as statistical information regarding its solution. Separately, it is also possible to obtain statistical information about the different libraries and export this material into a .mat file.

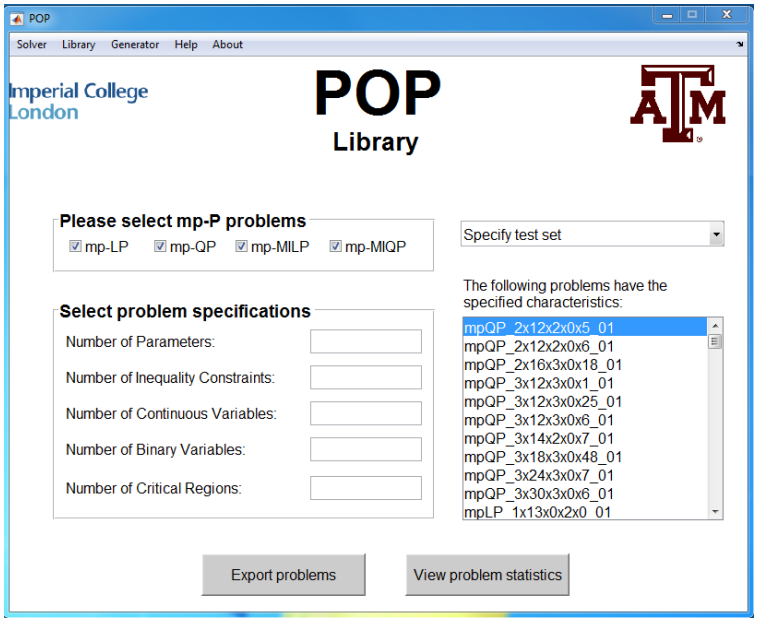

Figure 5: The POP Problem Library.

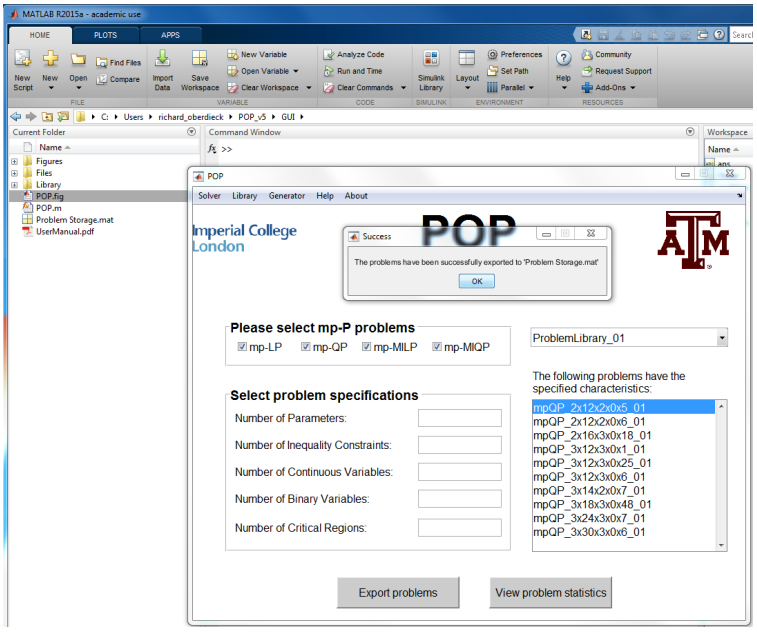

Figure 6: Exporting elements of the problem library.

4.3 POP Problem Generator

With POP, it is possible to generate random multi-parametric programming (mp-P) problems. Within the GUI, the user simply needs to specify the type as well as the size of the desired problem (see Figure 7). Additionally, the user can modify the options of the problem generator, in order to change the style of the problem generated. A list of the options is shown in Table 2.

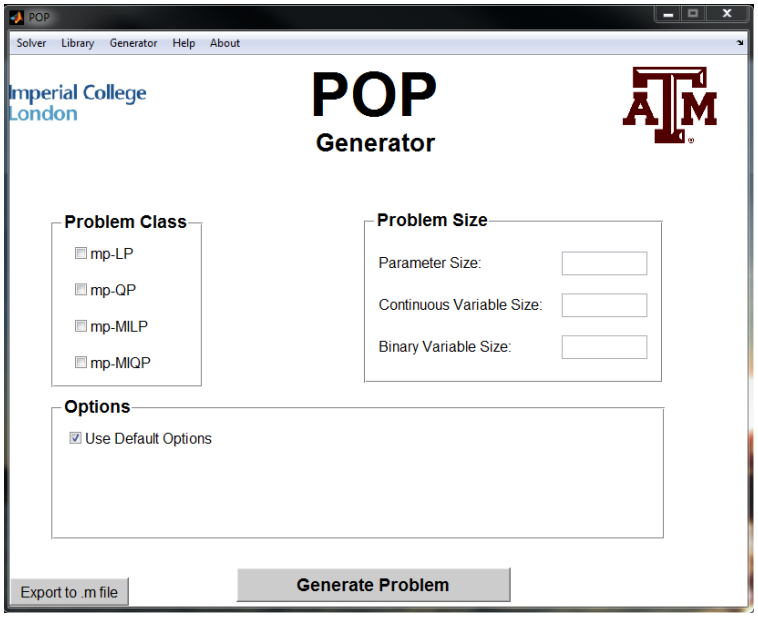

Figure 7: The POP Problem Generator.

4.3.1 Problem library generation

In order to generate a new problem library, you first need to generate the respective problems. Once these have been obtained, the variable containing these problems is available in the workspace. It can then be selected in the 'Solver' environment, allowing for the solution of the generated problems.

5 Direct use of POP

In this section, we describe the direct use of POP via the command window. In contrast to other toolboxes such as the MPT toolbox, POP is made up out of a set of function and does not define any additional classes[3]. This route was taken, as this makes the integration with other software tools much more straightforward and allows developers to use the content straight up. This section gives an overview over the different functions available. These functions are classified into two categories: **main** and **auxiliary**. While main functions are the ones mostly used during the execution of POP, auxiliary function are required by POP, but are rarely of interest of direct use.

[3]In MATLAB® terminology, classes are specific objects that are associated with user-defined functions and routines.

Table 2: The different options for the problem generator. Note that for most of them a range is given, i.e. the value itself is randomly generated at each function call.

Name of Option	Default	Meaning
Number of Constraints	1-4$(n+p+q)$	The number of constraints (excluding bounds) generated. Note that n denotes the number of continuous variables, p the number of binary variables and q the number of parameters.
Number of problems	1	The number of problems generated.
Lower Bound	-10	Lower bound of the parameter space. If the input is a scalar, the same lower bound applies to all parameters. If it is a vector (of the size of the parameters), then each entry is assigned to the lower bound of the corresponding parameter.
Upper Bound	10	Equivalent to the lower bound.
Range Value	5-25	Is a measure for the slack of the constraints created, i.e. the larger RangeValue the 'looser' the constraints. Has to be greater than 0.
Border on X	0.1-0.9	Denotes the probability of the continuous variables to be part of a certain constraint. Has to be between 0 and 1, where 1 means that the continuous variables will not have any constraints, and 0 that every constraint will contain elements of all continuous variables.
Shift on X	0.1-0.9	Denotes the range by which the number $\alpha \in [0,1]$ is shifted negatively, which in return will define the numerical value of the continuous variable coefficient in the constraint. The higher the shift is, the more probable it is that the entries of the continuous variables in the constraint matrix are negative.
Border on Y	0.1-0.9	Equivalent to 'Border on X' for binary variables.
Shift on Y	0.1-0.9	Equivalent to 'Shift on X' for binary variables.
Border on T	0.1-0.9	Equivalent to 'Border on X' for parameters.
Shift on T	0.1-0.9	Equivalent to 'Shift on X' for parameters.
Save Problem	'Off'	Enables or disables saving of the problem formulation in the current folder.

5.1 Main functions

5.1.1 mpQP

Given the problem definition of a multi-parametric quadratic programming (mp-QP) problem

```
problem.Q    = [nxn double]        Quadratic term in OBJ
problem.Ht   = [nxq double]        Cross-term in OBJ
problem.c    = [nx1 double]        Linear term in OBJ
problem.Qt   = [qxq double]        Quadratic parametric term in OBJ
problem.ct   = [qx1 double]        Linear parametric term in OBJ
problem.cc   = [1x1 double]        Constant term in OBJ
```

```
problem.A    = [mxn double]        Constraint matrix of x
problem.b    = [mx1 double]        Constant constraint term
problem.F    = [mxq double]        Constraint matrix of theta
problem.Aeq  = [wxn double]        Equality constraint matrix of x
problem.beq  = [wx1 double]        Constant equality constraint term
problem.Feq  = [wxq double]        Equality constraint matrix of theta
problem.Xmin = [nx1 double]        Lower bound for x
problem.Xmax = [nx1 double]        Upper bound for x
problem.Tmin = [qx1 double]        Lower bound for theta
problem.Tmax = [qx1 double]        Upper bound for theta
problem.CRA  = [rxq double]        Constraint matrix of parameter space
problem.CRb  = [rx1 double]        Constant term of parameter space
```

this problem can be solved using

```
[Solution, Time, Outer] = mpQP(problem,options)
```

where the second input is an optional structured array containing solver options (see 5.1.6). The output is thereby as follows:

Solution: The solution of the mp-QP problem, given in the form

```
Solution = 1xt struct array with fields:
                ActiveSet
CR
                Solution
```

Each element features the solution itself [in the field 'Solution'] for the continuous variable X the objective function Z and the Lagrangian multipliers λ and μ [in case of equality constraints], the critical region [in the field 'CR'] defined by $CR.A \cdot \theta \leq CR.b$, and the active set for each solution. Note that if 'MPT' is used for the solution, the active set is not available as a field.

Time: Timing information for the solution algorithm. These depend on the algorithm used for the solution:

Geometrical: 'Total' [total time required for solution], 'QP' [solution of the QP problem], 'Red' [removal of redundant constraints], 'Point' [identification of new θ_0]

Combinatorial: 'Total' [total time required for solution], 'Validation' [check whether combination is infeasible/already explored/LICQ fulfilled], 'Feasibility' [feasibility check of the parametric solution $x(\theta), \lambda(\theta)$], 'Optimality' [optimality check of parametric solution], 'Count' [number of active sets considered, i.e. how many active sets passed the validation stage], 'Iterations' [number of iterations of the algorithm, including validation]

Connected-Graph: 'Total' [total time required for solution], 'Validation' [check whether combination is infeasible/already explored/LICQ fulfilled], 'Feasibility' [feasibility check of the parametric solution $x(\theta), \lambda(\theta)$], 'Optimality' [optimality check of parametric solution and removal of redundant constraints], 'Count' [number of active sets considered, i.e. how many active sets passed the validation stage], 'Iterations' [number of iterations of the algorithm, including validation]

MPT: 'Total' [total time required for solution]

Outer: The feasible set of the solution, i.e. the set $Outer.A \cdot \theta \leq Outer.b$ contains all the different critical regions of the solution. Note that only the geometrical algorithm gives this information as part of the algorithm. However, the feasible set can be obtained by running the function `GetOuter`.

Remark 2. Note that in order to use the MPT solver in POP, MPT needs to be downloaded separately from people.ee.ethz.ch/ mpt/3/.

As an example, consider the following mp-QP:

$$\begin{aligned} \underset{x}{\text{minimize}}\ & 0.4x^Tx + \theta^T \begin{bmatrix} 1 & -1 \end{bmatrix}^T x \\ \text{subject to}\ & \begin{bmatrix} -1 \\ 1 \\ -0.85 \\ -0.84 \\ 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} x \leq \begin{bmatrix} 0.2 \\ 11 \\ 5.15 \\ 0.99 \\ 3 \\ 3.8 \\ 100 \\ 100 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ -0.51 & 0 \\ 0.54 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \\ & \theta \in \Theta = \{\theta \in \mathbb{R}^2 \mid -10 \leq \theta_i \leq 10, i = 1, 2\} \end{aligned} \tag{2}$$

Within POP, problem (2) is defined as:

```
problem.Q = 0.4;
problem.Ht = [1,-1];
problem.A = [-1; 1; -0.85; -0.84; 1; -1; 1; -1];
problem.b = [0.2; 11; 5.15; 0.99; 3; 3.8; 100; 100];
problem.F = [0, 0; 0, 0; -0.51, 0; 0.54, 0; 0, 0; 0, 0; 0, 0; 0, 0];
problem.CRA = [1, 0; 0, 1; -1, 0; 0, -1];
problem.CRb = [10; 10; 10; 10];
```

5.1.2 mpMIQP

Given the problem definition of a mp-MIQP problem (see problem 1)

```
problem.Q    = [(n+p)x(n+p) double]        Quadratic term in OBJ
problem.Ht   = [(n+p)xq double]            Cross-term in OBJ
problem.c    = [(n+p)x1 double]            Linear term in OBJ
problem.Qt   = [qxq double]                Quadratic parametric term in OBJ
problem.ct   = [qx1 double]                Linear parametric term in OBJ
problem.cc   = [1x1 double]                Constant term in OBJ
problem.A    = [mxn double]                Constraint matrix of x
problem.E    = [mxp double]                Constraint matrix of y
problem.b    = [mx1 double]                Constant constraint term
problem.F    = [mxq double]                Constraint matrix of theta
problem.Aeq  = [wxn double]                Equality constraint matrix of x
problem.Eeq  = [wxp double]                Equality constraint matrix of y
problem.beq  = [wx1 double]                Constant equality constraint term
problem.Feq  = [wxq double]                Equality constraint matrix of theta
problem.Xmin = [nx1 double]                Lower bound for x
```

```
problem.Xmax = [nx1 double]                Upper bound for x
problem.Tmin = [qx1 double]                Lower bound for theta
problem.Tmax = [qx1 double]                Upper bound for theta
problem.CRA  = [rxq double]                Constraint matrix of parameter space
problem.CRb  = [rx1 double]                Constant term of parameter space
```

This problem can be solved using

```
[Solution, Time] = mpMIQP(problem,options)
```

where the second input is an optional structured array containing solver options (see 5.1.6). The output is thereby as follows:

Solution: The solution of the mp-MIQP problem, given in the form

```
Solution = 1xt struct array with fields:
                CR
                Integer
                Solution
```

Each element features the solution itself [in the field 'Solution'] for the continuous variable X, the binary variable Y and the objective function Z, the critical region [in the field 'CR'] defined by $CR.A \cdot \theta \leq CR.b$, and the active set for each solution. Note that the field 'Solution' might feature an array of structs, as it has an envelope of solutions.

Time: Timing information for the solution algorithm. The four fields are: (i) Global [time to solve the global optimization problem], (ii) mpQP [time to solve the resulting mp-QP problem], (iii) Comparison [time for the comparison of the solution with the current best upper bound] and (iv) Total [the total algorithm time].

Remark 3. In order to solve the MINLP problem to identify the next candidate integer solution, a global optimization solver is required. In the past, Baron and Antigone have been used with a link to GAMS. However, apart from the inherent fragility and limitation of using two software tools (MATLAB and GAMS) instead of one, this also requires GAMS to start up and shut down at every iteration. Thus, we reformulate the MINLP into the following MILP:

$$
\begin{aligned}
z = \min_{x,y,\theta,t} \; & t \\
\text{s.t.} \; & Ax + Ey \leq b + F\theta \\
& A_{eq}x + E_{eq}y = b_{eq} + F_{eq}\theta \\
& \lfloor \text{OBJ} - \hat{z}\,(\theta) \rfloor \leq t \\
& \text{Integer Cut} \\
& t \leq -10^{-7} \\
& x \in \mathbb{R}^n, \; y \in \{0,1\}^p, \; \omega = \left[x^T y^T\right]^T, \\
& \theta \in \Theta := \{\theta \in \mathbb{R}^q \,|CR_A\theta \leq CR_b\},
\end{aligned}
\tag{3}
$$

where 'Integer Cut' exclude previously visited combinations [9, 12], and $\text{OBJ} - \hat{z}\,(\theta) \leq t$ denotes the 'Parametric Cut', i.e. that there has to be one point where the new solution is at least -10^{-7} better than the upper bound. However, if OBJ is quadratic, then this is a quadratic constraint and thus we compute two McCormick underestimators, denoted by $\lfloor \cdot \rfloor$, which linearize the problem and convert it into a MILP.

5.1.3 ProblemGenerator

This function generates the random problems. It is used in the following way:

```
problem = ProblemGenerator(Class,Size,options)
```

The inputs are thereby handled the following way:

Class: This input specifies the class of the problem to be generated. It can have the following structure:

- A char, i.e. 'mpLP', 'mpQP', 'mpMILP' or 'mpMIQP'. This way you specify one problem class
- A cell array, i.e. {'mpLP','mpQP'}. This way you can specify multiple problem classes to be generated
- Empty, i.e. '[]'. This will choose a random problem class.

Size: This structured array specifies the size requirements for the problem to be generated. It can be empty (i.e. 'Size = []'), then a problem of arbitrary size (between 1 and 20) is generated. Otherwise it should be a structured array featuring:

- Size.x: the number of the continuous variables. Accepts scalar as well as vectors as inputs.
- Size.y: the number of the binary variables. Accepts scalar as well as vectors as inputs.
- Size.t: the number of the parameters. Accepts scalar as well as vectors as inputs.
- Size.m: the number of the constraints. Accepts scalar as well as vectors as inputs.

Note that if one of the fields '.x', '.t' or '.y' [only for 'mpMILP' and 'mpMIQP' problems] is missing from the struct, then it is assigned an random number between 1 and 20.

options: This structured array allows the user to specify certain options (see Table 3). Any non-specified options will be set to their default value.

The output is thereby a feasible (i.e. non-empty feasible region) randomly generated problem, ready to be inserted into one of the solvers.

Remark 4. With certain combinations of Sizes, the algorithm is not able to find a feasible solution. If after 1000 variations of the non-specified parameters the algorithm is still not able to generate a feasible problem, it will stop. As it is a random generator, simply running the code again might do the trick. Otherwise you need to change something in the size definitions.

Remark 5. When the saving option is switched on, the problem is saved in the following format:

```
name = Class_qxmxnxpx0_X.mat
```

where Class is the problem class (i.e. 'mpLP','mpQP','mpMILP' or 'mpMIQP'), q is the number of parameters, m is the number of constraints, n is the number of continuous variables, p is the number of binary variables (if any) and X is a running index depending on whether a problem with the same specifications already exists in the same directory. Note that the last number is 0, as it is reserved for the number of critical regions obtained when you solve the problem.

Table 3: The options for the problem generator with and without GUI. Note that the number of constraints can be specified by (a) using the .m field in Size, or by changing the 'Constraint Factor', which defines the number of constraints based on the number of variables and paramters.

Name of Option with GUI	Name of Option without GUI (as fields of the 'option' struct.
Number of Constraints	.ConstraintFactor
Number of problems	.ProblemNumber
Lower Bound	.LowerBound
Upper Bound	.UpperBound
Range Value	.RangeValue
Border on X	.XBorder
Shift on X	.XShift
Border on Y	.YBorder
Shift on Y	.YShift
Border on T	.TBorder
Shift on T	.TShift
Save Problem	.Save

5.1.4 LibraryGeneration

As shown in Table 3, it is possible to use the problem generator to generate more than one problem. Thus, it is possible to generate your own library straight up. In order to solve the generated problems, POP provides the function 'LibraryGeneration' with the following syntax:

```
Stats = LibraryGeneration(Folder,options)
```

where the second input is an optional structured array containing solver options (see 5.1.6). The input 'Folder' can thereby be a folder where the problems are stored (according to the nomenclature stated above) or an array of problems obtained from the 'ProblemGenerator' function. The output 'Stats' reports several statistics related to the solution of the problems. As an example, below you can see how a small library of 10 test problems can be formulated as solved with the two functions described:

```
>> Size.x = 2:4; Size.t = 2:4;
>> options.ProblemNumber = 10;
>> problem = ProblemGenerator({'mpLP','mpQP'},Size,options);
>> Stats = LibraryGeneration(problem);
```

5.1.5 PlotSolution

In order to visualize the solution from a mp-P problem, POP features the function 'PlotSolution', which enables the plotting of both the partitioning of the parameter space as well as the optimal objective function. The syntax is thereby

```
PlotSolution(Solution, tfixed, option,options)
```

where the last input is an optional structured array containing solver options (see 5.1.6). The input 'Solution' is thereby the solution from a mp-P problem. The input 'tfixed' is optional and fixes

certain parameters to a given value, and the resulting slice is plotted. Note that 1D, 2D and 3D plots are supported. The 'option' input specifies the type of input:

- 'all' (default): The partitioning and the optimal objective function is shown
- 'CR': Only the partitioning is shown
- 'OBJ': Only the optimal objective function is shown

Note that `PlotSolution` supports 1,2 and 3D plotting, although the 3D plotting is still in its experimental phase.

5.1.6 OptionSet

This function defines the standard options within POP. It is possible to give a specific option setting to POP with the optional input `options`. Each option is discussed below[4]

`options.LPSolver` Type of solver for LPs. Default: Established at installation. Choices:

- `'MATLAB'`: Uses `linprog`.
- `'NAG'`: Uses `e04mf`.
- `'CPLEX'`: Uses `cplexlp`.
- `{'NAG','CPLEX'}`: Uses `e04mf`. If it fails, use `cplexlp`.

`options.QPSolver` Type of solver for QPs. Default: Established at installation. Choices:

- `'MATLAB'`: Uses `quadprog`.
- `'NAG'`: Uses `e04nf`.
- `'CPLEX'`: Uses `cplexqp`.
- `{'NAG','CPLEX'}`: Uses `e04nf`. If it fails, use `cplexqp`.

`options.GlobalSolver` The global solver used for mixed-integer problems. Established at installation. Choices:

- `'Enumeration'`: Exhaustively enumerates all possible integer combinations.
- `'MATLAB'`: Uses `intlinprog` to solve the MILP (3).
- `'CPLEX'`: Uses `cplexmilp` to solve the MILP (3).

`options.mpSolver` Type of solver for mp-LP/mp-QP. Default: `'Graph'`. Choices:

- `'Geometrical'`: Uses the geometrical algorithm [5].
- `'Combinatorial'`: Uses a variation of the combinatorial algorithm [6].
- `'Graph'`: Uses the connected-graph algorithm [7].
- `'MPT'`: Uses the MPT solver `mpt_plcp` [8].

`options.SolutionStyle` Solution representation (for storage). Default: `'full'`. Choices:

[4]Note that there are more options available in `OptionSet` than in the GUI.

- `'full'`: Yields the full solution of a mp-QP problem, i.e. critical region description, solution and active set. For larger problems (more than 1000 CRs), this requires significant amount of storage.
- `'reduced'`: Yields a matrix of dimensions $g \times n$, where g is the number of critical regions. Each row of the matrix is thereby an optimal active set, with minimal storage requirements. Together with the problem formulation, this matrix can be given to `SolutionGeneration` to retrieve the full solution.

`options.tolerance` Numerical tolerance. Default: `1.4901e-8`.

`options.MinimalStep` Minimal step size for geometrical algorithm. Default: `1000`.

`options.Step` Step size increase for geometrical algorithm. Default: `10`.

`options.TimeMax` Time out in seconds. Default: `600`.

`options.BigM` Big-M limits for variables. Default: `1e7`.

`options.Comparison` The comparison procedure used. Default: `'None'`. Choices:

- `'None'`: No comparison procedure is performed, and the solutions are just concatenated [9].
- `'MinMax'`: Using the function `DeltaSolver` It is checked whether there exists a point in the CR where the new solution is optimal [10].
- `'Affine'`: If both solutions are optimal in a CR (as established by `DeltaSolver`), then the CR is partitioned into polytopes using McCormick under- and overestimators [11].
- `'Exact'`: The exact solution of a mp-MIQP is obtained by using the approach in [12].

`options.DeltaSolver` The solution procedure for the 'MinMax' problem. Default: `'Approximate'`. Choices:

- `'Approximate'`: In case a non-convex problem is encountered, only a single McCormick is used to solve the problem.
- `'fmincon'`: Uses the function `fmincon` to solve the problem up to a certain tolerance. This might increase the computational time significantly.

`options.Progress` Show the progress of the algorithm. Default: `'None'`. Choices:

- `'None'`: No sign of the progress is shown.
- `'mpQP'`: A progress bar appears for the solution of the mp-QP problem.
- `'mpMIQP'`: A progress bar appears for the solution of the mp-MIQP problem.
- `'Tree'`: The connected graph of a mp-QP solution is shown. This is only available when the connected graph solver is used. In addition, the plot only becomes possible by pausing after each iteration for 0.2 seconds. Thus, `'Tree'` should only be used to visualize the solution, but not as a live animation for a problem of unknown complexity.

5.1.7 ss2qp

```
[mpv, online] = ss2qp(mpc, ss)
```

This function allows the formulation of the quadratic programming problem for an optimal control problem. The input is thereby the state-space model `ss`, given as the usual linear discrete time state space model (LTI) equations:

```
x(t+1)  = Ax(t) + Bu(t) + Cd(t)
y(t)  = Dx(t) + Eu(t) + e,
```

thus, `ss` is a structured array featuring the fields '.A', '.B' etc. The only notable exception to the `ss` definition from MATLAB® is `d`, the vector of measured disturbances (inputs that we have no direct control upon). The term `e` is the process model mismatch, an optional feature that will be discussed below. $e = \hat{y}(0) - y(0)$, the difference between real (measured) minus the predicted output at time t=0.

As for the `mpc` argument, this defines the MPC controller. It is a structured array, with most fields optional. Depending on what you define, you turn on/off various options for the MPC transformation. The generic optimization problem solved is:

$$
\begin{aligned}
\min_{u} \; & x_N^T P x_N \\
& + \sum_{k=1}^{OH-1} \left(x_k^T Q_k x_k + \left(y_k - y_k^R\right)^T Q R_k \left(y_k - y_k^R\right) \right) \\
& + \sum_{k=0}^{NC-1} \left(\left(u_k - u_k^R\right)^T R_k \left(u_k - u_k^R\right) + \Delta u_k^T R1_k \Delta u_k \right) \\
\text{s.t. } & x_{k+1} = A\,x_k + B\,u_k + C\,d_k \\
& y_k = D\,x_k + E\,u_k + e \\
& u_{min} \le u_k \le u_{max} \\
& \Delta u_{min} \le \Delta u_k \le \Delta u_{max} \\
& x_{min} \le x_k \le x_{max} \\
& y_{min} \le y_k \le y_{max},
\end{aligned}
\tag{4}
$$

The usual terminology is used, i.e. x are states, y outputs and u controls. All are (discrete) time dependent vectors. The subset of output variables that get tracked have time-dependent setpoints y^R. Finally Δu are changes in control variables, $\Delta u(k) = u(k) - u(k-1)$. Note $\Delta u(0)$ requires the last control action $u(-1)$ which becomes an extra parameter whenever you use the Δu formulation. Be careful with the timing of various discrete intervals k for the summations! Time starts at $k = 0$ where we know the plant state x_0 and can measure the plant output y_0. The control action in the first interval u_0 is an optimization variable. The prediction horizon is OH and control horizon NC (usually smaller, see Figure 8.

Remark 6. Whenever the control horizon is shorter than the prediction horizon, all later controls remain fixed, i.e. u(M-1) = u(M) = ... = u(N-1).

You can selectively leave terms out of the objective function just by not defining the respective quadratic term. So if you don't define `mpc.Q` field, there won't be state terms in the objective.

There are a few ways you can define these quadratic coefficients. Say you have 2 controls u and an output horizon of N=2 then the following are all valid definitions of the control table `R`:

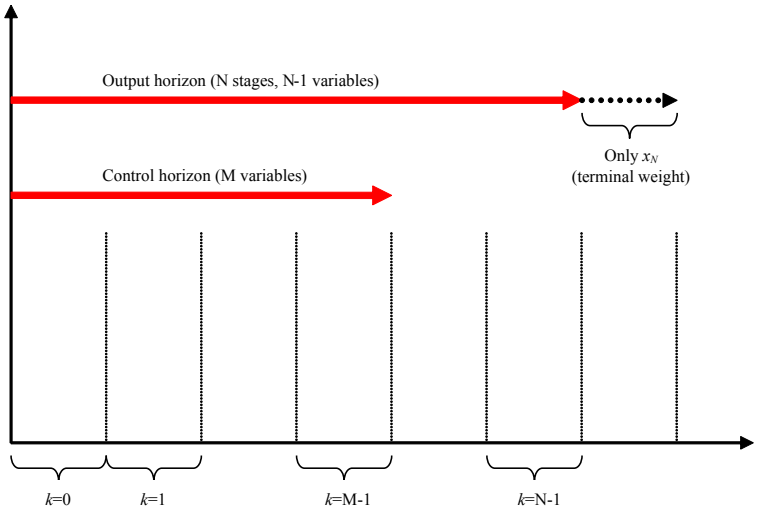

Figure 8: A schematic representation of the output and control horizon, OH and NC, respectively.

```
mpc.R = [2 3];       scaling factors, same across horizon
mpc.R = [2 -1; -1 3];       single period matrix, enables off-diagonal terms
mpc.R = [2 3  4 5];       scaling factors, one set per time interval
mpc.R = [2 0 0 0;  0 3 0 0; 0 0 4 0; 0 0 0 5];  matrix version for all horizon
```

Note that this format enables different weight matrices across the horizon. In the following, we define specific aspects of MPC formulation which can be handled with `ss2qp`:

Control move blocking: A method to decrease the number of optimization variables and the numerical complexity is move blocking, where a control level is applied for successive time intervals, effectively keeping it constant. This setting affects all the controls; it's not possible to have different blocking per individual control variable. The easiest way to do move blocking is with `mpc.uzip`, e.g.:

```
mpc.uzip = 2;  2 successive controls are blocked, ie u(0)=u(1), u(2)=u(3) etc
```

If you want to set irregular intervals, use the more robust `u_bundle` field, grouping the time intervals in a cell array, e.g. for a control horizon `NC=3`

```
mpc.u_bundle = {[1,3],[2]};  u(0)=u(2), u(1) free (index 1 corresponds
                                                    to time 0...)
```

Variable input reference: You can define if an input has reference value (appears as a parameter). The field `mpc.uRef` is the main setting for this option:

```
mpc.uRef=[NaN 1 0];
```

In this case only use the first input as parameter and tries to fix the values of the second and third to 1 and 2 respectively. Note that if we use the option `mpc.uRef`, we need to add references values of all the inputs, and this references value will be for the entire control horizon. In case we prefer to leave some inputs without reference we have to transform the matrix `R` accordingly (zero the values of input we want to leave free).

Variable state reference: `ss2qp` does not have the option for reference values in state variables, but we can overcome that by changing matrix `mpc.D` in order to make all the state variables, output variables.

Variable output tracking: You can control which outputs appear in the objective, and the type of tracking. The field `Ysp_Idx` is the main setting, holding the indices of the outputs to be tracked:

```
mpc.Ysp_Idx = [2];  only track the 2nd output (as appears in SS model)
```

All the remaining tracking parameters, including the quadratic coefficient `QR` (objective function) are anchored to `Ysp_Idx`. E.g. you shouldn't add weights for non-tracked output variables in `QR`. By default a single setpoint is added as a parameter, which is assumed constant over the output horizon $[1 \rightarrow OH - 1]$. It is possible to set bundling properties per output (unlike control move blocking) using the optional field `Ysp_Mode`, e.g. for N=4:

```
mpc.Ysp_Mode{1} = {[1] [2] [3]};  a different setpoint for 1st output
                                                    at each point
```

By default each setpoint introduces a new parameter, but it is possible to use another optional field to fix setpoints for particular outputs, called `Ysp_Val`. This should have the same number of entries as `Ysp_Mode` with values specifying the requested fixed setpoint value e.g for the vector above:

```
mpc.Ysp_Val{1} = {6,8, NaN};  fixed setpoints for intervals 1-2 and
                                                  parameter for 3
```

Note the use of MATLAB's `NaN` number to denote the free parameter, and specific fixed setpoint values otherwise.

Constraint control: Optionally each input, output and other variables can be constrained, either from below or above. The bound can be a specific hard coded number or a free parameter (same across all prediction horizon). It's best to illustrate the capability with specific examples:

```
mpc.Xmax = [2 2];
mpc.Xmin = [-1 -1]; Two states, upper bound 2 and lower bound -1
-----------------------------------
mpc.Xmax = [2 NaN];
mpc.Xmin = [-1 inf]; First state same as above, second state parametrically
                          bound from above and unbounded from below i.e.
-inf <= x_2 <= \theta
```

Again note the use of `NaN` symbol for free bounds and MATLAB's `inf` for infinity (unbounded). The bounds normally apply to all horizons but it is possible to turn off specific time intervals for all variables using `Xnocon` fields, e.g for N=5:

```
mpc.Xnocon = [2 3];   2nd and third period states unconstrained,
                                                 1,4 & 5 constrained
```

Process model mismatch: A simple way to eliminate steady state offsets for output variables. At the beginning of the prediction horizon, we know/estimate the state $x(0)$ and can measure the real output so we can estimate the process model mismatch as:

$$e = \hat{y}(0) - y(0) = \hat{y}(0) - (Cx(0) + Du(0)) . \tag{5}$$

The optional field `Ymismatch` declares the indices of outputs that feature mismatch. Typically it is aligned to `Ysp_Idx` field for tracking but it should be mainly declare outputs that are known to be mispredicted.

```
mpc.Ymismatch = [1 2];  output indices where we apply process-model mismatch
```

One parameter per output is introduced (constant for all prediction horizon). The measurement itself is the parameter - you don't have to calculate the initial error.

Table 4 lists all the fields supported in the MPC argument, and they are mostly optional as explained above:

The output from the `ss2qp` function is thereby the `mpv` and the `online` argument:

`mpv`: This contains the equivalent mp-QP formulation that corresponds to the MPC problem. All the equations (state space look-ahead) have been eliminated and the constraints reproduced for all future time intervals. Redundant inequalities have been removed and the only extra information required for the multiparametric solver is the range of parameters to search. Note that the lower and upper bounds on the parameters can best be added using the function `Bounds2Const`.

`online`: This contains the online controller version and information. In the interest of clarity and cross validation, the second returned struct contains the "online" version of the explicit controller that can be used with a standard QP solver for a second opinion or for a quick way to screen alternative MPC controller tuning parameters without deriving the explicit form (which can be time consuming). The most important fields can be used to construct the following QP optimization:

$$\begin{aligned} J(\theta) = & \underset{U}{\text{minimize}} \quad \tfrac{1}{2} U^T H U + \left(\theta^T F + U_{lin}\right) U + \theta^T T T \theta + T_{lin}\theta + c \\ & \text{subject to } AU \leq b + S\theta. \end{aligned} \tag{6}$$

If you need to reconstruct the dependent variables of the MPC (states, outputs etc) you can do so via the system of equations in `online.L` field, which is a composite matrix with the coefficients to the equation:

$$I x_D + L_1 x_F + L_2 \theta_1 + c = 0 \;\Rightarrow\; x_D = -L_1 x_F - L_2 \theta_1 - c \tag{7}$$

where x_D and x_F are the dependent and free variables respectively, θ_1 are the parameters associated to the state-space model and c is a constant column. The names of the variables and parameters are also returned for your convenience in fields `namesVars`, `namesFree` and `namesTheta`. As an extra complication, the order of columns (variables) in L matrix may be changed as a result of the Gaussian elimination (compared to `namesVars` vector) and this order can be found in `online.xid` field.

Table 4: The options for `mpc` field in the `ss2qp` function.

Option field	Description
`OH`	Output horizon
`NC`	Control horizon
`Q`	Objective coefficient for states (x)
`P`	Terminal weight matrix for states ($t = N$)
`QR`	Quadratic matrix for tracked outputs (y)
`pYmul`	Scalar adding more terminal weight for the last interval (`pYmul*QR`) to force output setpoints to be met
`R`	Quadratic matrix for manipulated variables (u)
`R1`	Weight matrix for output moves (Δu)
`Umax`	Upper bound vector for inputs; `NaN/Inf` supported
`Umin`	Lower bound vector for inputs
`Unocon`	Subset of control horizon where inputs U are unconstrained
`Xmax`	Upper bound vector for states; `NaN/Inf` supported
`Xmin`	Lower bound vector for states
`Xnocon`	Subset of output horizon where states X are unconstrained
`Ymax`	Upper bound vector for outputs; `NaN/Inf` supported
`Ymin`	Lower bound vector for outputs
`Ynocon`	Subset of output horizon where outputs Y are unconstrained
`YsoftP`	Penalty coefficient (positive), turns on output constraint softening, adds extra slack variable in objective function
`DUmax`	Upper bound vector for input changes; `NaN/Inf` supported
`DUmin`	Lower bound vector for input changes
`DUnocon`	Subset of control horizon where input changes ΔU are unconstrained
`bFixedDisturbance`	When measured disturbances d are present in the SS model, by default the disturbance introduces one extra parameter per time period (time varying). Setting this field to 1 freezes the disturbances at their initial levels, reducing the number of parameters
`uzip`	Control move blocking width, in number of consecutive time intervals
`u_bundle`	Cell array with customized blocking intervals for all outputs
`Ymismatch`	Which output indices are subject to process model mismatch
`Ysp_Idx`	Indices of tracked outputs
`Ysp_Mode`	Setpoint blocking cell array per output (similar to `u_bundle`)
`Ysp_Val`	Same in structure as `Ysp_Mode`; `NaN` for free setpoint or specific number for fixed setpoint
`uRef`	Reference point for the input variable

5.2 Auxiliary functions

Remark 7. Most of these functions feature an optional input `options`, i.e. a structured array containing solver options (see 5.1.6).

5.2.1 addbounds

```
[constr, idx] = addbounds(bound, bMax, horz, nocon)
```

This local function adds constraints for bounds on variables. It accepts bound vector (time independent, one per variable), whether we're setting upper bounds, horizon length and unconstrained intervals. It returns the constraints in local vars mode (nx*horz columns) with parameters and constant RHS also returns indices of parameters associated with bounds (if any). This function is only used in `ss2qp`.

5.2.2 BinContConversion

```
problem = BinContConversion(problem, Cont2Bin, Bin2Cont)
```

This function converts a binary/continuous variable in the mp-MILP/mp-MIQP problem into a continuous/binary variable. The inputs are:

problem: The problem struct used in POP.

Cont2Bin: The indices as array of the continuous variables that should become binary variables. Default is [].

Bin2Cont: The indices as array of the binary variables that should become continuous variables. Default is [].

For example the definition 'Cont2Bin = [1 5]' would change the 1st and 5th continuous variable into a binary.

5.2.3 Bounds2Const

```
[A,b] = Bounds2Const(Xmin, Xmax, dim)
```

Given the lower and upper bounds 'Xmin' and 'Xmax' as a vector with n entries, the function outputs the corresponding constraint set. Note that if the lower and upper bounds are the same for all dimensions, you can also specify them as scalar values and the dimensionality in the input 'dim' (which is optional otherwise). Note that there is a complementary function called 'Const2Bounds'

5.2.4 CandidateGeneration

```
Lists = CandidateGeneration(Sets, iac, problem)
```

This function generates all possible candidates from the current active set and the minimal representation of the critical region. The function is thereby part of the 'ConnectedGraph' approach, i.e. the fact that the solution to a mp-LP/mp-QP problem is given by a connected graph. For more information, see:

- Gal, T.; Nedoma, J. (1972) Multiparametric Linear Programming. Management Science, 18(7), 406-422.
- Oberdieck, R.; Diangelakis, N.A.; Pistikopoulos, E.N. (2016) Explicit Model Predictive Control: A connected-graph approach, submitted.

5.2.5 Chebyshev

```
[x,R] = Chebyshev(A,b,EqIdx,BinIdx)
```

Given the set of linear constraints defined by 'A' and 'b', this function calculates the Chebyshev center of the polytope, 'x' and the corresponding radius 'R'. If the polytope is empty, then 'NaN' is returned. The 3rd and 4th input are optional:

EqIdx: If any of the constraints are equality constraints, this can be specified in this array.

BinIdx: If any of the varaibles are binary variables, this can be specified in this array.

5.2.6 Combinatorial

```
[Solution,Timing] = Combinatorial(problem)
```

This function is a mp-LP/QP solver, which solves the 'problem' and provides the parametric solution. The approach is based on [6]. However, on the contrary to the paper, this approach first solves the feasibility LP before continuing to the optimality LP.

5.2.7 Comparison

```
Sol = Comparison(Solution, Upper, Outer)
```

Given the solution to a mp-LP/mp-QP, 'Solution', its feasible space 'Outer' as well as the upper bound in the space considered, 'Upper', compare the upper bound and solution to obtain the optimal profile 'Sol' based on certain comparison procedures.

5.2.8 ComputeSolution

```
Solution = ComputeSolution(problem, sol_theta)
```

Given the parametric solution 'sol_theta' to the mp-LP/mp-QP problem 'problem', defined by a [nx(q+1) double] (the constant term is the last column of 'sol_theta'), the function generates the appropriate structure 'Solution' containing the information on the critical region and objective function values.

5.2.9 con2vert

```
[V,nr] = con2vert(A,b)
```

Given the set of linear constraints defined by 'A' and 'b' this open source function (provided by Michael Kleder) calculates the corresponding vertices based on a primal-dual method [13]. Additionally, the index 'nr' identifies the list of the rows in 'A' which are not redundant constraints.

5.2.10 ConnectedGraph

```
[Solution,Time] = ConnectedGraph(problem)
```

This is a mp-LP/QP solver which emplyos the connected graph approach from [7] to solve `problem`. It is part of the wrapper `mpQP`.

5.2.11 Const2Bounds

```
[Xmin, Xmax] = Const2Bounds(A,b)
```

The complementary function to 'Bounds2Const', this function calculates the lower and upper bound of a polytope defined by a set of linear constraints via the solution of $2n$ LP problems. If an variable is unbounded, the corresponding 'Xmin' or 'Xmax' value is infinity (minus or plus, respectively).

5.2.12 ConstraintReverse

```
CRnew = ConstraintReverse(CR_star, CR)
```

Given the critical region 'CR_star' and the set 'CR' (where $CR_{star} \subseteq CR$), 'ConstraintReverse' explores the remaining part of 'CR', i.e. $\overline{CR} = CR \backslash CR_{star}$, by reversing the different constraints. The new set of critical regions is given in CRnew (and thus $CRnew \cup CR_{star} = CR$) [14].

5.2.13 CRSplitting

```
NewUpdate = CRSplitting(Upper, Solution, CR)
```

Given the upper bound 'Upper' and the solution 'Solution' as well as a corresponding parameter space 'CR', this auxiliary function provides the partitions based on an affine approximation of the difference of objective functions [11].

5.2.14 currentPOP

```
version = currentPOP
```

This function outputs the current version of POP.

5.2.15 DeltaSolver

```
[Tmin,Tmax] = DeltaSolver(Diff, CR)
```

Given a general quadratic function 'Diff' and the polytope 'CR' (with the fields 'CR.A' and CR.b'), this function solves for minimum and maximum value the function attains over the polytope, 'Tmin' and 'Tmax', respectively. Note that the function features two options: either it is solved with a very simple approximation to get under and overestimations of 'Tmin' and 'Tmax' respectively, or 'fmincon' is employed to solve the problem exactly.

5.2.16 DisplaySolution

```
DisplaySolution(Solution)
```

This function produces a print out of the solution of a mp-P problem.

5.2.17 EqualitySubstitution

```
problem = EqualitySubstitution(problem)
```

This function substitutes all equality constraints in a mp-P problem. However, this function is **work in progress**, and should not be used without caution.

5.2.18 exactLocate

```
f = exactLocate(x,P,Q,r)
```

This function is a subfunction used in the `PlotSolution` function for the plotting of the exact solution.

5.2.19 exactSol

```
[c,ceq] = exactSol(x,Solution,i,j)
```

This function is a subfunction used in the `ExactSolution` function for the calculation of the exact solution.

5.2.20 ExactSolution

```
New_Sol = ExactSolution(Solution)
```

This function takes the solution struct featuring envelopes of solutions and polytopic critical regions and provides the exact solution by solving the corresponding QCLP feasibility problems. Please note that the function uses that MATLAB in-built function 'fmincon'. Numerical tests have shown that the solver is incredibly sensitive to the tolerances used. Hopefully, a future version of POP will feature a work-around for this problem.

5.2.21 facetInnerCircle

```
[x,R] = facetInnerCircle(A,b,ind)
```

Given a polytope defined by the set of linear constraints 'A' and 'b', as well as an index of a constraint, 'ind', this function calculates the Chebyshev center on that specific constraint. In other words, we assume $A_i x = b_i$ and ask how large is the largest ball that can be fit in this $n-1$ dimensional space. The output is the point, 'x', and the radius, 'R'.

5.2.22 FindActiveSets

```
[iac,candidates] = FindActiveSets(sol, lam, A, b, EqS, isLP)
```

Given the output from a LP or QP solver, i.e. the primal solution, 'sol', the Lagrangian multipliers, 'lam', the constraints 'A' and 'b' and the information on whether it is a LP or not, 'isLP', as well as the indices of the constraint set which are equality constraints, this function identifies the indices of the active set 'iac'. In addition, it also returns a set of 'candidates', which might also be the correct active set. This is necessary for the geometrical algorithm, since the active set definition is not always unique.

5.2.23 FixParameter

```
Solution = FixParameter(Solution,tfixed)
```

This function fixes the parameter values to the values specified in 'tfixed'. If it should not be fixed, it should be set to 'NaN'.

5.2.24 FirstRegion

```
[Solution, Sets] = FirstRegion(problem)
```

This function performs the first iteration of the geometrical algorithm.

5.2.25 Geometrical

```
[Solution,Timing,Outer] = Geometrical(problem)
```

This function is a step-size based mpLP/QP solver, which solves the 'problem' and provides the parametric solution. The approach is based on [5] with modifications in several key elements.

5.2.26 GetOuter

```
Outer = GetOuter(Solution)
```

Given the solution to a mp-LP/mp-QP problem this function calculates the feasible space 'Outer' by using the step-size approach from the geometrical algorithm.

5.2.27 GlobalOptimization

```
Integer = GlobalOptimization(problem, Solution)
```

Given the problem formulation 'problem', as well as the solution from a previous iteration, 'Solution', this function calculates a candidates integer solution for the solution of the mp-QP problem.

5.2.28 InfeasibleRemoval

```
problem = InfeasibleRemoval(problem)
```

This helper function removes infeasible constraints from the problem formulation by solving 1 LP for each constraint.

5.2.29 installPOP

```
installPOP(dir)
```

This function installs the POP toolbox in the directory dir, and automatically adds it to the MATLAB path. If no directory is provided, the MATLAB toolbox directory is automatically used.

5.2.30 IntegerFix

```
problem = IntegerFix(problem, intVec)
```

Given the problem formulation 'problem', and a candidate integer solution 'intVec', this function fixes the appropriate elements in 'problem' and returns the corresponding mp-LP/mp-QP problem.

5.2.31 LibraryGeneration

```
Stats = LibraryGeneration(Folder)
```

This function takes a folder or array of problems containing a set of mp-LP/mp-QP/mp-MILP/mp-MIQP problems, solves all the problems (if possible), and returns (a) the updated folder containing the number of critical regions etc. such that it is straight up usable in POP, (b) the statistics for the solution [timing information, infeasible, fails] and (c) a .mat file containing all the solutions. These are stored separately, as the storage space becomes very quickly unmanagable.

5.2.32 lsolver

```
[sol,how,lambda] = lsolver(c,A,b,x0,EqIdx)
```

This function interfaces POP with the LP solver selected. Given the linear part 'c' constrained by 'A' and 'b', the starting point 'x0' and the possibility to transform some of the inequality constraints in 'A' and 'b' into equality constraints, it solves the resulting linear programming problem and returns the solution 'sol', the exitflag (as a char, i.e. 'ok' for normal completion and 'infeasible' for infeasbilitiy) and the Lagrangian multipliers 'lambda'.

5.2.33 McCormick

```
[Estimate, delta] = McCormick(Quad,CR,n)
```

This function generates 'n' McCormick relaxations of the quadratic function 'Quad' over the polytope 'CR'. The output is the approxiamtion 'Estimate' as well as the maximum deviation from the quadratic function 'delta'.

5.2.34 milsolver

```
[sol, how, fval] = milsolver(c, A, b, BinIdx, EqIdx)
```

This function provides a unified wrapper for the different MILP solvers available [intlinprog from MATLAB, cplexmilp from CPLEX, and exhaustive enumeration]. It provides the solution to the following optimization problem

$$\begin{aligned} fval = \underset{x}{\text{minimize}} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b \\ & x(BinIdx) \text{ are binary} \\ & A(BinIdx,:)x = b(BinIdx), \text{ i.e. they are equality constraints} \end{aligned} \tag{8}$$

The elements of the problem which are non-continuous are thereby specified by 'BinIdx'. For example, if the 2nd and 10th variable should be binary, set BinIdx = [2,10]. Note that the solver options set in 'OptionSet' are used. Additionally note that the polytope Ax <= b should be compact, otherwise unbounded solutions might result. You can check for unboundedness using the function 'Const2Bounds'.

5.2.35 MinkowskiSum

```
Set = MinkowskiSum(SetA,SetB)
```

This function calculates the Minkowski sum between `SetA` and `SetB`, defined in half-space representation.

5.2.36 miqsolver

```
[sol, how, fval] = miqsolver(Q, c, A, b, BinIdx, EqIdx)
```

This function provides a unified wrapper for the different MIQP solvers available [cplexmiqp from CPLEX, and exhaustive enumeration]. It provides the solution to the following optimization problem

$$\begin{aligned} fval = \underset{x}{\text{minimize}} \quad & \tfrac{1}{2}x^TQx + c^Tx \\ \text{subject to} \quad & Ax \leq b \\ & x(BinIdx) \text{ are binary} \\ & A(BinIdx,:)x = b(BinIdx), \text{ i.e. they are equality constraints} \end{aligned} \tag{9}$$

The elements of the problem which are non-continuous are thereby specified by 'BinIdx'. For example, if the 2nd and 10th variable should be binary, set BinIdx = [2,10]. Note that the solver options set in 'OptionSet' are used. Additionally note that the polytope Ax <= b should be compact, otherwise unbounded solutions might result. You can check for unboundedness using the function 'Const2Bounds'.

5.2.37 ModifyOptionSet

```
ModifyOptionSet(options)
```

Given the input struct 'options', this function generates a new file 'OptionSet' automatically. This can be helpful to run computational studies for the different algorithms in POP. Note that the input should have the fields which it wants to be modified, e.g. `options.mpSolver = 'Geometrical';`. Those options which are not specified in the input will be set to their default value.

5.2.38 MPTtoPOP

```
POP_Solution = MPTtoPOP(MPT_Solution,problem)
```

This function converts the solution obtained from the MPT toolbox to the equivalent output of the POP toolbox. Note that the second input is optional and denotes the original (POP) problem formulation. It is required to calculated the optimal objective function value. While functions such as 'PointLocation' will still work, if you would like to use the output as part of a larger program we suggest that you provide the problem formulation, if possible.

5.2.39 nchooseallk

```
V = nchooseallk(Set, k)
```

This function is a version of the MATLAB `nchoosek` function: instead of just giving out all the picks for k number of selections, this function gives all the options for all picks UP to k number of selections.

5.2.40 normalizeConstraints

```
[A,b,C,d] = normalizeConstraints(A,b,C,d)
```

This function takes the polytope defined by the inequality constraints 'A' and 'b', and the equality constraints 'C' and 'd', and returns the normalized version, where double entries are removed and hidden equality constraints are detected.

5.2.41 Old2New

```
problem = Old2New(problem,CR)
```

This function converts problem formulations from the old POP toobox into the new syntax. Note that we assume that there are NO equality constraints.

5.2.42 OptionSet_Default

```
options = OptionSet
```

The default values for the options.

5.2.43 Parallel_Combinatorial

```
[Solution,Time] = Parallel\textunderscore Combinatorial(problem)
```

This function implements the combinatorial approach in a parallel way. For more information on the algorithm see the function verb|Combinatorial|.

Note the following:

- Parallelism has inherent overheads. Thus, especially when only 2 or 3 workers are used in parallel, the parallel algorithm might in fact be [significantly] slower than the sequential version
- The parameter `rho\textunderscore limit` regulates how many problems are solved on a worker in parallel until the results are combined together. Tests have shown that a high value of `rho\textunderscore limit` yields good performance for the combinatorial algorithm. However, if a computational study is attempted, the user should check several values of `rho\textunderscore limit` to understand its impact.

5.2.44 Parallel_ConnectedGraph

```
[Solution,Time] = Parallel\textunderscore ConnectedGraph(problem)
```

This function implements the combinatorial approach in a parallel way. For more information on the algorithm see the function verb|ConnectedGraph|.

Note the following:

- Parallelism has inherent overheads. Thus, especially when only 2 or 3 workers are used in parallel, the parallel algorithm might in fact be [significantly] slower than the sequential version
- The parameter `rho\textunderscore limit` regulates how many problems are solved on a worker in parallel until the results are combined together. Tests have shown that a high value of `rho\textunderscore limit` yields good performance for the combinatorial algorithm. However, if a computational study is attempted, the user should check several values of `rho\textunderscore limit` to understand its impact.

5.2.45 ParOptConversion

```
problem = ParOptConversion(problem, Par2Opt, Opt2Par)
```

This function converts a parameter/optimization variable in the mp-LP/mp-QP problem into an optimization variable/parameter. The inputs are: 'problem' [the problem struct used in POP], 'Par2Opt' [the indices as array of the parameters that should become optimization variables. Default is []. 'Opt2Par' [The indices as array of the optimization variables that should become parameters. Default is [].

5.2.46 pGeneration

```
pGeneration(mcode)
```

This function takes as input a .m code or a directory featuring .m files, and (a) generates the corresponding .p code files and (b) creates .m file which contains only the top part (comments) in order to enable the command 'help'. Note that if there exists a file and a directory of the same name, the function will always prioritize the directory.

Note: In order to avoid unwanted deletion of files or directories, a new directory will always be created which as the suffix '_pVersion'.

5.2.47 PlotCR

```
PlotCR(CR,nRegions)
```

A subroutine that allows the plotting of arrays of polytopes 'CR' defined by the fields 'A' and 'b'. The optional input nRegions specifies the plot title.

5.2.48 PointLocation

```
[nCR,x,z,y] = PointLocation(Solution,theta)
```

Given the solution to a multi-parametric programming, 'Solution', and a point 'theta', this function finds the corresponding partition 'nCR' and evaluates the variables 'x' and 'y' and the objective function 'z'.

5.2.49 PontryaginDifference

```
Set = PontryaginDifference(SetA,SetB)
```

This function calculates the Pontryagin difference between `SetA` and `SetB`, defined in half-space representation. Currently the function only considers sets which contain the origin.

5.2.50 POPviaMPT

```
[MPT_Output, POP_Output, Timing] = POPviaMPT(POP_problem)
```

This function provides a way to solve a mp-P problem given in POP-syntax with the MPT toolbox from ETH Zurich. The output is thereby an object of the 'Opt' class as well as the solution to the mp-P problem.

5.2.51 ProblemSet

```
problem = ProblemSet(problem)
```

This sub-routine fills all the undefined fields in the problem structs with zeros and also checks whether the provided problem is feasible/lower-dimensional in the initial parameter space. It also checks for boundedness and adds the appropriate bounding constraints if necessary. Lastly, it also handles inequality and equality constraints, and it removes redundant inequality constraints. In order to switch that feature off, set 'problem.processing = 'Off';'.

5.2.52 Projection

```
CR = Projection(A,b,Idx,BinIdx,method)
```

This function projects the polytope $Ax \leq b$ along the indices 'Idx'. Thus, the output is a region, $CR.A * x(Idx) \leq CR.b$. In the case of binary variables, these are specified in the input 'BinIdx'. Note that in that case, the output will potentially be an array of critical region.

5.2.53 qphess

```
[hx,user,iwsav] = qphess(n,jthcol,hessn,ldh,x,user,iwsav)
```

A sub-routine related to NAG.

5.2.54 qsolver

```
[sol,how,lambda] = qsolver(Q,c,A,b,x0,EqIdx)
```

This function interfaces POP with the LP solver selected. Given the quadratic part 'Q' and the linear part 'c' constrained by 'A' and 'b', the starting point 'x0' and the possibility to transform some of the inequality constraints in 'A' and 'b' into equality constraints, it solves the resulting quadratic programming problem and returns the solution 'sol', the exitflag (as a char, i.e. 'ok' for normal completion and 'infeasible' for infeasbilitiy) and the Lagrangian multipliers 'lambda'.

5.2.55 Redundandize

```
[A,b,removedRows] = Redundandize(A,b,Xmin,Xmax)
```

This function removes the redundant constraints from a set of linear inequalities 'A' and 'b' given the lower and upper bounds 'Xmin' and 'Xmax' (optional inputs) [15, 16].

5.2.56 scaleRows

```
[A,b] = scaleRows(A,b)
```

This function normalizes the constraints 'A' and 'b'.

5.2.57 SolutionGeneration

```
Solution = SolutionGeneration(problem,ActiveSets)
```

Given the problem formulation and the optimal active sets, this function generates the corresponding parametric solution.

5.2.58 U_triangulate

```
[A, columns, rankA] = U_triangulate(A, maxC, bUnit)
```

This function returns an upper triangular matrix transformation.

5.2.59 UnionOfPolytopes

```
HybridRegion = UnionOfPolytopes(Region)
```

This function takes as input n Regions in halfspace representation, i.e. `Region(k).{A,b}` with $k = 1, ..., n$ and yields a single region `HybridRegion.{A,E,b}`, which is the equivalent representation of the n Regions using binary variables.

5.2.60 updatePOP

```
update = updatePOP
```

This function updates the POP toolbox. If 'update = 1', then an update occurred, and the list of updated files is reported in 'lsfile'. If 'update = 0', then no update occurred.

5.2.61 VerifySolution

```
ok = VerifySolution(Solution,problem,nPoints)
```

This function takes the solution to a multi-parametric programming problem, 'Solution', the original problem formulation 'problem' and the optional input of a number of points, 'nPoints', and randomly generates 'nPoints' and solves the resulting LP/QP/MILP/MIQP at the specific points. If there is a problem, then 'ok = false'.

References

[1] Efstratios N. Pistikopoulos, Nikolaos A. Diangelakis, Richard Oberdieck, Maria M. Papathanasiou, Ioana Nascu, and Muxin Sun. PAROC - an Integrated Framework and Software Platform for the Optimization and Advanced Model-Based Control of Process Systems. *Chemical Engineering Science*, 136:115–138, 2015.

[2] M. Herceg, M. Kvasnica, C. N. Jones, and M. Morari. Multi-Parametric Toolbox 3.0. In *Control Conference (ECC), 2013 European*, pages 502–510, 2013.

[3] J. Löfberg. YALMIP: A toolbox for modeling and optimization in MATLAB. In *CCA/ISIC/CACSD*, 2004.

[4] Richard Oberdieck, Nikolaos A. Diangelakis, Maria M. Papathanasiou, Ioana Nascu, and Efstratios N. Pistikopoulos. POP – Parametric Optimization Toolbox. *Industrial & Engineering Chemistry Research*, 55(33):8879–8991, 2016.

[5] M. Baotic. An Efficient Algorithm for Multiparametric Quadratic Programming, 2002.

[6] Arun Gupta, Sharad Bhartiya, and P.S.V. Nataraj. A novel approach to multiparametric quadratic programming. *Automatica*, 47(9):2112–2117, 2011.

[7] Richard Oberdieck, Nikolaos A. Diangelakis, and Efstratios N. Pistikopoulos. Explicit Model Predictive Control: A connected-graph approach. *Automatica*, accepted for publication, 2016.

[8] Martin Herceg, Colin N. Jones, Michal Kvasnica, and Manfred Morari. Enumeration-based approach to solving parametric linear complementarity problems. *Automatica*, 62:243–248, 2015.

[9] Vivek Dua, Nikolaos A. Bozinis, and Efstratios N. Pistikopoulos. A multiparametric programming approach for mixed-integer quadratic engineering problems. *Computers & Chemical Engineering*, 26(4–5):715–733, 2002.

[10] Daniel Axehill, Thomas Besselmann, Davide M. Raimondo, and Manfred Morari. A parametric branch and bound approach to suboptimal explicit hybrid MPC. *Automatica*, 50(1):240–246, 2014.

[11] Richard Oberdieck, Martina Wittmann-Hohlbein, and Efstratios N. Pistikopoulos. A branch and bound method for the solution of multiparametric mixed integer linear programming problems. *Journal of Global Optimization*, 59(2-3):527–543, 2014.

[12] Richard Oberdieck and Efstratios N. Pistikopoulos. Explicit hybrid model-predictive control: The exact solution. *Automatica*, 58(0):152–159, 2015.

[13] David Avis and Komei Fukuda. A pivoting algorithm for convex hulls and vertex enumeration of arrangements and polyhedra. *Discrete & Computational Geometry*, 8(1):295–313, 1992.

[14] Alberto Bemporad, Manfred Morari, Vivek Dua, and Efstratios N. Pistikopoulos. The explicit linear quadratic regulator for constrained systems. *Automatica*, 38(1):3–20, 2002.

[15] A. L. Brearley, G. Mitra, and H. P. Williams. Analysis of mathematical programming problems prior to applying the simplex algorithm. *Mathematical Programming*, 8(1):54–83, 1975.

[16] J. Telgen. *Redundancy and linear programs.* Mathematisch Centrum, Amsterdam, 1981.