Model Predictive Control

9. Model Predictive Control with MATLAB

Jun.-Prof. Dr.-Ing. Daniel Görges
Juniorprofessur für Elektromobilität
Technische Universität Kaiserslautern

Toolboxes for Model Predictive Control with MATLAB

- **Model Predictive Control Toolbox**
  - commercial (offered by The MathWorks)
  - focused on applications (modeling, design, simulation, code generation)
- **Multi-Parametric Toolbox (MPT)**
  - open source (developed at ETH Zürich)
  - focused on research (modeling, design, code generation, multi-parametric progr., comput. geometry)
- **Yet Another LMI Parser (YALMIP)**
  - open source (developed at Linköping University)
  - focused on research (modeling, design, optimization, linear matrix inequalities)
- **Hybrid Toolbox**
  - open source (developed at IMT Lucca and ETH Zürich)
  - focused on research (modeling, design, simulation, code generation, specialization on hybrid systems)

Introduction

- **Installation**
  - Execute the installation script `install_mpt3.m` which can be downloaded from

    control.ee.ethz.ch/~mpt/3/Main/Installation

- **Updating**
  - Execute the commands

```
tbxmanager
clear classes
mpt_init
```

- **Removal**
  - Execute the command

```
tbxmanager uninstall mpt mptdoc cddmex fourier glpkmex hysdel ...
lcp yalmip sedumi espresso
```

Introduction

- **Object-Oriented Programming**
  - The Multi-Parametric Toolbox is based on object-oriented programming
  - Object-oriented programming relies on objects, classes, instances, methods, and properties
- **Objects**
  - Everything is an object! (Alan Curtis Kay, pioneer of object-oriented programming)
- **Classes**
  - A class is a template for objects with the same behavior and the same parameters
  - E.g. the class `Car` defines objects which can accelerate, brake, etc. and have a mass etc.
- **Instances**
  - An instance is a realization of an object from a class
  - E.g. `mycar = Car('Mass',800)` creates an instance of the class `car` with mass 800 kg
  - Note that instances are often denoted as objects

Introduction

- **Methods**
  - Methods are algorithms describing the behavior of objects from classes
  - Methods are accessed with the `.` - operator
  - E.g. `mycar.accelerateCar(2)` will accelerate the car with acceleration 2 m/s²
  - E.g. `mass = mycar.getMass()` will return the mass of the car
- **Properties**
  - Properties are parameters describing the state of objects from classes
  - Methods are accessed with the `.` - operator
  - E.g. `mycar.mass = 750` will set the mass of the car to 750 kg directly
  - E.g. `mycar.setMass(750)` will set the mass of the car to 750 kg indirectly using a method
  - Note that properties are also denoted as members

Modeling and Simulation

- **Modeling of Polyhedra**
  - A polyhedron with H-representation $P = \{\boldsymbol{x} | \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\}$ can be defined with

```
P = Polyhedron(A,b)
P = Polyhedron('A',A,'b',b)
```

  - A polyhedron with H-representation $P = \{\boldsymbol{x} | \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{A}_\mathrm{e}\boldsymbol{x} = \boldsymbol{b}_\mathrm{e}, \boldsymbol{lb} \leq \boldsymbol{x} \leq \boldsymbol{ub}\}$ can be defined with

```
P = Polyhedron('A',A,'b',b,'Ae',Ae,'be',be,'lb',lb,'ub',ub)
```

  - The parameters describing a polyhedron in H-representation can be obtained with

`P.A`, `P.b`, `P.Ae`, `P.be`

  - The parameters describing a polyhedron in H-representation can be obtained more compactly with

`P.H`, `P.He`

where $\boldsymbol{H} = (\boldsymbol{A} \quad \boldsymbol{b})$ and $\boldsymbol{H}_\mathrm{e} = (\boldsymbol{A}_\mathrm{e} \quad \boldsymbol{b}_\mathrm{e})$

Modeling and Simulation

- **Modeling of Polyhedra**
  - A polyhedron with V-representation $P = \{\boldsymbol{x} | \boldsymbol{x} = \boldsymbol{V}^T\boldsymbol{\alpha}, \boldsymbol{\alpha} \geq \boldsymbol{0}, \boldsymbol{1}^T\boldsymbol{\alpha} = 1\}$ can be defined with

    ```
    P = Polyhedron(V)
    P = Polyhedron('V',V)
    ```

    with the rows of the matrix $\boldsymbol{V}$ describing the vertices
  - The parameters describing a polyhedron in V-representation can be obtained with

    ```
    P.V
    ```
  - The representation of a polyhedron can be obtained with

    ```
    P.hasHRep, P.hasVRep
    ```
  - The representation of a polyhedron can be converted with

    ```
    P.computeHRep, P.computeVRep
    ```

Modeling and Simulation

- **Modeling of Polyhedra**
  - The minimal representation of a polyhedron can be computed with

    `P.minHRep`, `P.hasVRep`

    which removes all redundant (in)equalities or vertices
  - The minimal representation of a polyhedron can be checked with

    `P.irredundatHRep`, `P.irredundantVRep`
  - A polyhedron can be visualized with

    `P.plot`

    `plot(P)`
- **Exercise**
  - Define and visualize the polyhedron introduced on Slide 3-22 under MATLAB using MPT

Modeling and Simulation

- **Modeling of an LTI System**
  - An LTI system can be defined with

```
A = [1 1; 0 1];
B = [1; 0.5];
C = [1 0];
D = 0;
Ts = 1;
sys = LTISystem('A',A,'B',B,'C',C,'D',D,'Ts',Ts)
```

  - Note that an unforced LTI system is defined if `'B'` is omitted
  - Note that only the state equation is defined if `'C'` and `'D'` are omitted
  - Note that the sampling period 1 is assumed if `'Ts'` is omitted

Modeling and Simulation

- **Modeling of an LTI System**
  - The initial state can be defined with

    ```
    x_0 = [1; 1.5];
    sys.initialize(x_0)
    ```

  - The current state can be obtained with

    ```
    x = sys.getStates()
    ```

  - The current output can be obtained with

    ```
    y = sys.output()
    ```

Modeling and Simulation

- **Simulation of an LTI System**
  - The next state for a given input can be obtained with

    ```
    u = 0.5;
    x_next = sys.update(u)
    ```

  - The state and output sequence for a given input sequence can be obtained with

    ```
    U = [-2 -2 -1 0 1 2 2]
    data = sys.simulate(U)
    ```

  - Note that `data` is a structure with the fields `X`, `Y`, and `U`
  - Note that the fields can be accessed with `data.X`, `data.Y`, and `data.U`

Regulation

- **Definition of the Cost Function**
  - The state and input weighting matrix for a quadratic cost function can be defined with

```
Q = eye(sys.nx);
sys.x.penalty = QuadFunction(Q);
R = eye(sys.nu);
sys.u.penalty = QuadFunction(R);
```

  - The state and input weighting matrix for a linear cost function (1-norm) can be defined with

```
sys.x.penalty = OneNormFunction(Q);
sys.u.penalty = OneNormFunction(R);
```

  - The state and input weighting matrix for a linear cost function (∞-norm) can be defined with

```
sys.x.penalty = InfNormFunction(Q);
sys.u.penalty = InfNormFunction(R);
```

Regulation

- **Definition of the Terminal Cost**
  - The terminal cost according to Theorem 6.3 can be defined with

```
P = sys.LQRPenalty;
sys.x.with('terminalPenalty');
sys.x.terminalPenalty = P;
```

- **Definition of the Constraints**
  - The state and input (box) constraints can be defined with

```
sys.x.min = [-5; -5];
sys.x.max = [5; 5];
sys.u.min = -1;
sys.u.max = 1;
```

Regulation

- **Definition of the Terminal Constraint Set**
  - The terminal constraint set according to Theorem 6.3 can be defined with

```
X_N = sys.LQRSet;
sys.x.with('terminalSet');
sys.x.terminalSet = X_N;
```

- **Definition of a Model Predictive Controller**
  - A model predictive controller for an LTI system with given prediction horizon can be defined with

```
N = 5;
ctrl = MPCController(sys,N);
```

  - Note that the prediction horizon can alternatively be defined with

```
ctrl.N = 5;
```

Regulation

- **Computation of the Optimal Input**
  - The optimal input for a given state can be computed with

```
u = ctrl.evaluate(x)
```

- **Computation of the Optimal Input, State, and Output Sequence and Cost**
  - The optimal input, state, and output sequence and cost for a given state can be computed with

```
[u feasible openloop] = ctrl.evaluate(x)
```

  - Note that `feasible` is a binary variable indicating whether the problem was feasible
  - Note that `openloop` is a structure with the fields `U`, `X`, `Y`, and `cost`
  - Note that the fields can be acc. with `openloop.U`, `openloop.X`, `openloop.Y`, `openloop.cost`
- **Visualization of the Optimal Input, State, and Output Sequence**
  - The optimal input, state, and output sequence can be visualized with

```
ctrl.model.plot();
```

Regulation

- **Definition of the Closed-Loop System**
  - The closed-loop system can be defined with

    ```
    loop = ClosedLoop(ctrl,sys)
    ```

  - Note that the system used for the closed loop may be different from the system used for design
    - E.g. a simplified system has been used for design to reduce the complexity
    - The complete system should then be used for the simulation to assess the controller
- **Computation of the Closed-Loop Input, State, and Output Sequence and Cost**
  - The closed-loop input, state, and output sequence and cost for a given state can be computed with

    ```
    data = loop.simulate(x,N_sim)
    ```

  - Note that `data` is a structure with the fields `U`, `X`, `Y`, and `cost`
  - Note that the fields can be accessed with `data.U`, `data.X`, `data.Y`, and `data.cost`

Regulation

- **Visualization of the Closed-Loop Input, State, and Output Sequence**
  - The closed-loop input, state, and output sequence can be visualized with

```
plot(0:N_sim-1,data.U);
plot(0:N_sim,data.X);
plot(0:N_sim-1,data.Y);
```

- **Generation of an Explicit Model Predictive Controller**
  - An explicit model predictive controller (PWA state feedback controller) can be generated with

```
expctrl = ctrl.toExplicit()
```

- **Visualization of a PWA State Feedback Controller**
  - A PWA state feedback controller can be visualized with

```
expctrl.feedback.fplot();
```

Regulation

- **Visualization of the Optimal Cost Function**
  - The optimal cost function can be visualized with

```
expctrl.cost.fplot();
```

- **Visualization of the Regions**
  - The regions can be visualized with

```
expctrl.partition.plot();
```

- **Visualization of the Closed-Loop State Trajectory**
  - The closed-loop state trajectories can be visualized with

```
expctrl.clicksim();
```

  - Note that the initial state is defined by clicking
  - Note that the visualization is only available for second-order systems (phase plane)

Regulation

- **Exercise**
  - Consider the mass-spring-damper system introduced on Slide 8-5ff with mass $m = 4\ \mathrm{kg} = \mathrm{const.}$
  - Define the model under MATLAB using MPT with the sampling period $h = 0.5\ \mathrm{s}$
  - Design a model predictive controller under MATLAB using MPT with
    - quadratic cost function
    - state weighting matrix $\boldsymbol{Q} = 100\boldsymbol{I}_{2\times 2}$
    - input weighting factor $R = 1$
    - state constraints $\begin{pmatrix}-1\ \mathrm{m} & -0.5\,\frac{\mathrm{m}}{\mathrm{s}}\end{pmatrix}^T \leq \boldsymbol{x} \leq \begin{pmatrix}1\ \mathrm{m} & 0.5\,\frac{\mathrm{m}}{\mathrm{s}}\end{pmatrix}^T$
    - input constraints $-1.5\ \mathrm{N} \leq u \leq 1.5\ \mathrm{N}$
    - terminal weighting matrix $\boldsymbol{P}$ according to Theorem 6.3
    - terminal constraint set $\mathbb{X}_N$ according to Theorem 6.3
    - prediction horizon $N = 5$

Multi-Parametric Toolbox

Regulation

- **Exercise**
  - Simulate the closed-loop system under MATLAB using MPT for the initial state $\boldsymbol{x}_0 = \begin{pmatrix} 1\,\mathrm{m} & 0\,\frac{\mathrm{m}}{\mathrm{s}} \end{pmatrix}^T$
  - Visualize the closed-loop input and state sequence one below the other in two diagrams
  - Design an explicit model predictive controller under MATLAB using MPT
  - Visualize the PWA state feedback controller, optimal cost function, and regions
  - Visualize the closed-loop state trajectories
  - Simulate the closed-loop system under MATLAB for the initial state $\boldsymbol{x}_0 = \begin{pmatrix} 0\,\mathrm{m} & 0\,\frac{\mathrm{m}}{\mathrm{s}} \end{pmatrix}^T$ and the disturbance $w(k) = 0.1\big(\sigma(k+2) - \sigma(k+3)\big)$ which is added to the input $u(k)$ both with MATLAB and Simulink
  - Visualize the closed-loop state and input sequence one below the other in two diagrams
- **Hints**
  - Use `Example_9_Regulation.m` as a template
  - Simulations of discrete-time systems can be realized in MALTAB using a for-loop

Tracking

- **Definition of a Time-Varying Reference**
  - A time-varying output reference can be defined with

```
sys.y.penalty = QuadFunction(eye(sys.ny));
sys.y.with('reference');
sys.y.reference = 'free';
...
y_ref = [ones(1,10) 2*ones(1,10) 3*ones(1,10)];
data = loop.simulate(x,N_sim,'y.reference',y_ref)
```

  - Note that a time-varying state reference can be defined analogously
  - Note that an explicit model predictive controller can be generated also for tracking
  - The reference input is then an additional parameter

Tracking

- **Definition of a Time-Varying Reference**
  - A time-varying output reference can be defined using the delta input formulation with

```
Q = eye(sys.ny);
sys.y.penalty = QuadFunction(Q);
sys.u.with('deltaPenalty');
R = eye(sys.nu);
sys.u.deltaPenalty = QuadFunction(R);
sys.y.with('reference');
sys.y.reference = 'free';
...
u_0 = 0;
y_ref = [ones(1,10) 2*ones(1,10) 3*ones(1,10)];
data = loop.simulate(x,N_sim,'u.previous',u_0,'y.reference',y_ref)
```

Multi-Parametric Toolbox

Tracking

- **Definition of a Time-Varying Reference**
  - Note that an explicit model predictive controller can be generated also for tracking
  - The reference input and the previous output are then additional parameters
- **Exercise**
  - Consider the mass-spring-damper system and the model predictive controller studied on Slide 9-19
  - Extend the model predictive controller for tracking the time-varying state reference

$$x_{1,\text{ref}}(k) = 0.5\big(\sigma(k+10) - \sigma(k+30)\big)\ \text{m}$$

  - Simulate the closed-loop system under MATLAB using MPT for the initial state $\boldsymbol{x}_0 = \begin{pmatrix} 0\ \text{m} & 0\,\frac{\text{m}}{\text{s}} \end{pmatrix}^T$
  - Visualize the closed-loop input and state sequence and the reference sequence
- **Hints**
  - Use `Example_9_Tracking.m` as a template
  - Investigate the influence of the state weighting matrix $\boldsymbol{Q}$

Additional Constraints

- **Definition of Constraints with Filters**
  - Constraints are defined with filters on signals such as the input `u`, the state `x`, and the output `y`
  - Filters are activated using `with` and deactivated with `without`
- **Definition of Lower and Upper Bounds on Signals**
  - Lower and upper bounds on signals can be defined with (cf. Slide 9-13)

```
sys.x.min = [-5;  5];
sys.x.max = [5; 5];
```

- **Definition of Soft Lower and Upper Bounds on Signals**
  - Soft lower and upper bounds on signals can be defined with

```
sys.y.with('softMin');
sys.y.with('softMax');
```

Additional Constraints

- **Definition of Lower and Upper Bounds on the Rates of Signals**
  - Lower and upper bounds on the rates of signals can be defined with

```
sys.u.with('deltaMin');
sys.u.with('deltaMax');
sys.u.deltaMin = -10;
sys.u.deltaMax = 10;
```

- **Definition of a Move Blocking Constraint**
  - A move blocking constraint (cf. Slide 5-22) can be defined with

```
sys.u.with('block');
sys.u.block.from = 3;
sys.u.block.to = N;
```

Additional Constraints

- **Definition of a Set Constraint**
  - A set constraint can be defined with

```
P_set = Polyhedron('V',[0 0; 1 0; 0 1]);
sys.x.with('setConstraint');
sys.x.setConstraint = P_set;
```

- **Definition of a Terminal Set Constraint**
  - A terminal set constraint can be defined with

```
P_terminal_set = Polyhedron('Ae',eye(sys.nx),'be',zeros(sys.nu,1));
sys.x.with('terminalSet');
sys.x.terminalSet = P_terminal_set;
```

  - Note that the commands given above define the terminal constraint $\boldsymbol{x}(k+N) = \mathbf{0}$
  - This terminal constraint can be used to ensure stability as shown on Slide 6-27

Additional Constraints

- **Definition of an Initial Set Constraint**
  - An initial set constraint can be defined with

```
P_initial_set = Polyhedron('lb',-10,'ub',10);
sys.x.with('initialSet');
sys.x.initialSet = P_initial_set;
```

  - Note that the commands given above define the initial set constraint $-10 \leq x(0) \leq 10$
  - An initial set constraint can be used to reduce the exploration space in multi-parametric programming and therefore the computation time for the generation of an explicit model predictive controller

- **Definition of Binary Constraints**
  - Binary constraints can be defined with

```
sys.u.binary = true;      % enforce all elements binary
sys.u.binary = idx;       % enforce the elements indexed by idx binary
```

Additional Constraints

- **Definition of Constraints with YALMIP**
  - Constraints can also be defined by interfacing with YALMIP
  - YALMIP provides more flexibility to formulate constraints (e.g. time-varying or joint constraints)
  - YALMIP can be interfaced with

```
Y = ctrl.toYALMIP();
ctrl.fromYALMIP(Y);
```

- **Definition of Time-Varying Constraints with YALMIP**
  - Time-varying constraints can be defined with

```
Y = ctrl.toYALMIP();
Y.constraints = [Y.constraints, -0.5 <= y.variables.u(:,1) <= 0.5];
Y.constraints = [Y.constraints, -0.8 <= y.variables.u(:,2) <= 0.8];
ctrl.fromYALMIP(Y);
```

Additional Constraints

- **Definition of Time-Varying Constraints with YALMIP**
  - Note that the commands given above define the constraints $-0.5 \leq \boldsymbol{u}(0) \leq 0.5, -0.8 \leq \boldsymbol{u}(1) \leq 0.8$
- **Definition of Joint Input and State Constraints with YALMIP**
  - Joint state and input constraints, i.e. constraints in standard form (cf. Slide 5-10), can be defined with

```
Y = ctrl.toYALMIP();
for k = 1:size(Y.variables.u,2)
    Y.constraints = [Y.constraints ...
                     M*Y.variables.x(:,k)+E*Y.variables.u(:,k) <= b];
end;
ctrl.fromYALMIP(Y);
```

Code Generation

- **Export of a PWA State Feedback Controller to MATLAB Code**
  - A PWA state feedback controller can be exported to MATLAB code with

```
expctrl.optimizer.toMatlab('mycontroller.m','primal','obj');
```

  - The optimal input sequence and region for a given state can then be obtained from

```
[U, region] = mycontroller(x)
```

  - Note that the exported MATLAB code is independent from MPT
  - Note that the exported MATLAB code is executed much faster than the MPT evaluation function
  - Note that the PWA state feedback controller can be reduced to provide only the optimal input with

```
expctrl.optimizer.trimFunction('primal',ctrl.model.nu);
expctrl.optimizer.toMatlab('mycontroller.m','primal','obj');
```

  - This can further reduce the computation time

Code Generation

- **Export of a PWA State Feedback Controller to C Code**
  - A PWA state feedback controller can be exported to C code with

```
expctrl.exportToC('file','directory');
```

  - The command given above generates the files

`file.c`	pure C code for a target system (e.g. a microcontroller)
`file_mex.c`	mex interface for fast evaluation in MATLAB
`file_sfunc.c`	S-function interface for fast evaluation in Simulink

  - The interfaces can be compiled with

```
mex file_mex.c
mex file_sfunc.c
```

  - The compiled functions can be used in MATLAB and Simulink like any other function

Additional Tools

- **Computation of an Invariant Set**
  - An invariant set can be computed with

    ```
    P_invariant = sys.invariantSet();
    ```

  - Note that the command given above yield the positive invariant set for unforced systems and the maximal control invariant set for forced systems
- **Visualization of an Invariant Set**
  - An invariant set can be visualized with

    ```
    P_invariant.plot();
    ```

Additional Tools

- **Documentation**
  - The documentation of a class can be shown in the Help Browser with

    ```
    doc LTISystem
    ```

  - The documentation of a class can be shown in the Command Window with

    ```
    help LTISystem
    ```

  - The methods of a class can be shown in the Command Window with

    ```
    methods('LTISystem')
    ```

  - The properties of a class can be shown in the Command Window with

    ```
    properties('LTISystem')
    ```

Introduction

- **Model Predictive Control with YALMIP**
  - YALMIP allows the formulation and solution of model predictive control problems in a natural way
  - YALMIP is particularly useful for
    - complex constraints (cf. Slide 9-28f)
    - explicit model predictive control (yalmip.github.io/example/explicitmpc/)
    - robust model predictive control (yalmip.github.io/example/robustmpc/)
    - hybrid model predictive control (yalmip.github.io/example/hybridmpc/)
    - model predictive control of LPV systems (yalmip.github.io/example/explicitlpvmpc/)
  - YALMIP for general problems is described under yalmip.github.io/example/standardmpc/
  - YALMIP for regulation problems introduced on the following slides

Regulation

- **Modeling of an LTI System**
  - An LTI system can be defined with

```
A = [1 1; 0 1];
B = [1; 0.5];
```

- **Definition of the Cost Function**
  - The state and input weighting matrix for a quadratic cost function can be defined with

```
Q = eye(nx);
R = eye(nu);
```

- **Definition of the Constraints**
  - The state and input (box) constraints can be defined with

```
x_min = [-5; -5]; x_max = [5; 5];
u_min = -1; u_max = 1;
```

Regulation

- **Modeling of the Model Predictive Controller**
  - An model predictive controller for the LTI system with given prediction horizon can be defined with

```
N = 5;
u = sdpvar(repmat(nu,1,N),repmat(1,1,N));
x = sdpvar(repmat(nx,1,N),repmat(1,1,N));
x_0 = sdpvar(nx,1);
constr = [];
cost = 0;
x{1} = x_0;
for i = 1:N
    x{i+1} = A*x{i}+B*u{i};
    cost = cost+x{i}'*Q*x{i}+u{i}*R*u{i};
    constr = [constr, x_min <= x{i} <= x_max, u_min <= u{i} <= u_max];
end;
```

Regulation

- **Computation of the Optimal Input**
  - The optimal input for a given state can be computed with

```
optimize([constr, x_0 == [1; 1.5]],cost);
value(u{1})
```

- **Computation of the Closed-Loop Input and State Sequence**
  - The closed-loop input and state sequence for a given state can be computed with

```
N_sim = 15;
x_sim{1} = [1; 1.5];
for k = 1:N_sim
    optimize([constr, x_0 == x_sim{k}],cost);
    u_sim{k} = value(u{1});
    x_sim{k+1} = A*x_sim{k}+B*u_sim{k};
end;
```