MATMPC User Manual

Yutao Chen
Email: chenyutao890704@gmail.com

February 13, 2023

Contents

1 Installation

MATMPC requires two fundamental software and toolboxes

1. MATLAB (R2014b or above)
2. CaADi (3.3.0)

CaADi is the state-of-art automatic differentiation (AD) tool [**?**] that is employed in MATMPC for computing derivatives of a number of functions. For how to download and install CaADi, please visit this site.

1.1 Compiler

MATMPC is a collection of routines, mainly written in MATLAB scripts. However, its core algorithmic routines are written in MATLAB C API. To compile these routines into MEX functions, it is required to use GCC or MinGW (CLANG is not tested, but would probably work). In LINUX systems, users just need to install the compatible version of GCC for MATLAB. In WINDOWS systems, users need to install MinGW for MATLAB, from here.

1.2 Advanced functionality installation

1.2.1 Linear Algebra Library

By default, MATMPC employs MATLAB built-in linear algebra (LA) library lapack and blas from INTEL MKL. For the full condensing routine, MATMPC also supports using BLASFEO [**?**]. Note that, in WINDOWS, installing BLASFEO requires to install MinGW for WINDOWS (not for MATLAB), which is described in detail in the "HPIPM-tutorial".

1.2.2 QP Solver

MATMPC comes with MATLAB built-in QP solver *quadprog* but other QP solvers can also be used. By default, MATMPC provides compiled binaries of qpOASES [**?**] and OSQP [**?**] both for WINDOWS and LINUX. If users want to compile qpOASES themselves, please visit here and install its MATLAB interface only. If users want to compile OSQP, please visit here and install its MATLAB interface only.

MATMPC provides more QP solvers, including HPIPM and IPOPT [**?**], both using interior point method (IPM) for solving sparse QP problems. In order to use HPIPM, either for sparse or partially condensed QP, users should run */mex_core/compile_hpipm.m* to obtain compatible MEX functions. Such compilation requires to install BLASFEO and HPIPM. Details are given in the "HPIPM-tutorial". The IPOPT binary is delivered by OPTI Toolbox at this site. Hence, it is required to install OPTI Toolbox first. Note that OPTI Toolbox is only compatible with WINDOWS system.

2 Data Structure

2.1 Problem Description

2.1.1 NLP

In MATMPC, we solve the following nonlinear programming (NLP) problem:

$$\begin{aligned}
\min_{x_k,u_k}\ & \frac{1}{2}\sum_{k=0}^{N-1} d(h(x_k,u_k)-h_{ref}^k)_{W_k} + \frac{1}{2}d_N(h_N(x_N)-h_{ref}^N)_{W_N} \\
s.t.\ & 0 = x_0 - \hat{x}_0, \\
& 0 = x_{k+1} - \phi_k(x_k,u_k),\ k=0,1,\dots,N-1, \\
& \underline{x}_k \le x_k \le \overline{x}_k,\ k=0,1,\dots,N, \\
& \underline{u}_k \le u_k \le \overline{u}_k,\ k=0,1,\dots,N-1, \\
& \underline{r}_k \le r_k(x_k,u_k) \le \overline{r}_k,\ k=0,1,\dots,N-1, \\
& \underline{r}_N \le r_N(x_N) \le \overline{r}_N,
\end{aligned} \tag{1}$$

where $h:\mathbb{R}^{n_x}\times\mathbb{R}^{n_u}\to\mathbb{R}^{n_y}, h_N:\mathbb{R}^{n_x}\to\mathbb{R}^{n_{y_N}}$ are vector functions of state and control (x,u), with corresponding references h_{ref}^k and h_{ref}^N. Note that h, h_N can be nonlinear and nonconvex. The outer objective functions $d:\mathbb{R}^{n_y}\to\mathbb{R}, d_N:\mathbb{R}^{n_{y_N}}\to\mathbb{R}$ are assumed convex, e.g. linear sum or quadratic. W_k, W_N are weights for each term in d for stage k. $\hat{x}_0$ is the measurement of the current state. The function $r(x_k,u_k):\mathbb{R}^{n_x}\times\mathbb{R}^{n_u}\to\mathbb{R}^{n_c}$ and $r(x_N):\mathbb{R}^{n_x}\to\mathbb{R}^{n_{c_N}}$ can be linear or nonlinear, with lower and upper bound $\underline{r}_k, \overline{r}_k$. $\phi_k(x_k,u_k)$ is a numerical integration operator that solves the following initial value problem (IVP) and returns the solution at t_{k+1}.

$$0 = f_{impl}(\dot{x}(t), x(t), u(t), t), \quad x(0) = x_k. \tag{2}$$

If possible, the dynamics can be written in an explicit form as

$$\dot{x}(t) = f_{exp}(x(t), u(t), t), \quad x(0) = x_k. \tag{3}$$

The decision variables can be written in the compact form:

$$\begin{aligned}
\mathbf{x} &= \left[x_0^\top, x_1^\top, \dots, x_N^\top\right]^\top, \\
\mathbf{u} &= \left[u_0^\top, u_1^\top, \dots, u_{N-1}^\top\right]^\top
\end{aligned} \tag{4}$$

2.1.2 QP

At iteration i, given the initial trajectory $(\mathbf{x}^i, \mathbf{u}^i)$, problem (1) is linearized and would result in the following quadratic programming (QP) problem:

$$\begin{aligned}
\min_{\Delta\mathbf{x},\Delta\mathbf{u}} \quad & \sum_{k=0}^{N-1}(\frac{1}{2}\begin{bmatrix}\Delta x_k\\ \Delta u_k\end{bmatrix}^\top \begin{bmatrix}Q_k^i & S_k^i\\ S_k^{i^\top} & R_k^i\end{bmatrix}\begin{bmatrix}\Delta x_k\\ \Delta u_k\end{bmatrix} + \begin{bmatrix}g_{x_k}^i\\ g_{u_k}^i\end{bmatrix}^\top \begin{bmatrix}\Delta x_k\\ \Delta u_k\end{bmatrix}) \\
& + \frac{1}{2}\Delta x_N^\top Q_N^i \Delta x_N + g_{x_N}^{i^\top}\Delta x_N \\
s.t. \quad & \Delta x_0 = \hat{x}_0 - x_0, \\
& \Delta x_{k+1} = A_k^i \Delta x_k + B_k^i \Delta u_k + a_k^i,\ k = 0,\dots,N-1 \\
& \underline{x}_k - x_k^i \le \Delta x_k \le \overline{x}_k - x_k^i,\ k = 1,\dots,N \\
& \underline{u}_k - u_k^i \le \Delta u_k \le \overline{u}_k - u_k^i,\ k = 0,\dots,N-1 \\
& \underline{c}_k^i \le C_k^i \Delta x_k + D_k^i \Delta u_k \le \overline{c}_k^i,\ k = 0,\dots,N-1 \\
& \underline{c}_n - c_N^i \le C_N^i \Delta x_N \le \overline{c}_n - c_N^i,
\end{aligned} \tag{5}$$

where

$$\begin{aligned}
\Delta\mathbf{x} &= \mathbf{x} - \mathbf{x}^i, \\
\Delta\mathbf{u} &= \mathbf{u} - \mathbf{u}^i
\end{aligned} \tag{6}$$

and

$$\begin{aligned}
& g_{x_k}^i = \frac{\partial d^i}{\partial x_k}, \quad g_{u_k}^i = \frac{\partial d^i}{\partial u_k}, \quad g_{x_N}^i = \frac{\partial d_N^i}{\partial x_N}, \\
& A_k^i = \frac{\partial \phi_k}{\partial x_k}, \quad B_k^i = \frac{\partial \phi_k}{\partial u_k}, \quad a_k^i = \phi(x_k^i, u_k^i) - x_{k+1}^i, \\
& C_k^i = \frac{\partial r_k}{\partial x_k}, \quad D_k^i = \frac{\partial r_k}{\partial u_k}, \quad C_N^i = \frac{\partial r_N}{\partial x_N}, \\
& \overline{c}_k^i = \overline{r}_k - r_k(x_k^i, u_k^i), \quad \underline{c}_k^i = \underline{r}_k - r_k(x_k^i, u_k^i), \\
& \overline{c}_N^i = \overline{r}_N - r_N(x_N^i), \quad \underline{c}_N^i = \underline{r}_N - r_N(x_N^i)
\end{aligned} \tag{7}$$

The Hessian matrices Q_k, S_k, R_k can be approximated by the Gauss-Newton (GN) or the Generalized-Gauss-Newton (GGN) method. If h is (non)linear and d is sum of squares, both methods lead to the same Hessian approximation.

2.1.3 Dual solution

The optimal primal solution of (5) is denoted as $(\Delta\mathbf{x}^{i^*}, \Delta\mathbf{u}^{i^*})$. The dual solution of (5) are Lagrangian multipliers associated with constraints in (5). They are

$$\lambda = \begin{bmatrix}\lambda_0\\ \vdots\\ \lambda_N\end{bmatrix}, \quad \mu_u = \begin{bmatrix}\mu_{u_0}\\ \vdots\\ \mu_{u_{N-1}}\end{bmatrix}, \quad \mu_x = \begin{bmatrix}\mu_{x_0}\\ \vdots\\ \mu_{x_{N-1}}\end{bmatrix}, \quad \mu_g = \begin{bmatrix}\mu_{g_0}\\ \vdots\\ \mu_{g_N}\end{bmatrix} \tag{8}$$

where λ is with equality constraints, μ_u with input bounds, μ_x with state bounds, μ_g with general constraints. In detail, $\lambda_0 \in \mathbb{R}^{n_x}$ is with initial value constraint in (5), and $\lambda_k \in \mathbb{R}^{n_x}, k = 1, \ldots, N$ are with state evolution constraints. $\mu_{x_k} \in \mathbb{R}^{n_x}, k = 0, \ldots, N-1$ are with state bounds of $x_k, k = 1, \ldots, N$, since x_0 is included in the initial value constraint. $\mu_{g_k} \in \mathbb{R}^{n_c}, k = 0, \ldots, N-1$ are with general constraints, and $\mu_{g_N} \in \mathbb{R}^{n_{cn}}$ is with the general constraint at the terminal shooting point. Together with the primal solution, the dual solution is updated by (9)

2.1.4 Globalization

The primal solution is used to update the solution of (1) by

$$\mathbf{x}^{i+1} = \mathbf{x}^i + \alpha^i \Delta \mathbf{x}^{i^*}, \; \mathbf{u}^{i+1} = \mathbf{u}^i + \alpha^i \Delta \mathbf{u}^{i^*}, \tag{9}$$

where α^i is the step length determined by globalization strategies. A practical line search SQP algorithm employing ℓ_1 merit function [**?**] is employed in `MATMPC`. The merit function is defined as

$$m(\mathbf{w}; \mu) = l(\mathbf{w}) + \mu \|e(\mathbf{w})\|_1 \tag{10}$$

where $\mathbf{w} = [\mathbf{x}^\top, \mathbf{u}^\top]^\top$, $l(\mathbf{w})$ is the objective function of (1), $e(\mathbf{w})$ contains all constraints in (1) with slack variables for inequality constraints and μ the penalty parameter. The step $\alpha^i \Delta \mathbf{w}^i$ is accepted if

$$m(\mathbf{w}^i + \alpha^i \Delta \mathbf{w}^i; \mu^i) \leq m(\mathbf{w}^i; \mu^i) + \eta \alpha^i \mathcal{D}(m(\mathbf{w}^i; \mu^i); \Delta \mathbf{w}^i) \tag{11}$$

where $\mathcal{D}(m(\mathbf{w}^i; \mu^i); \Delta \mathbf{w}^i)$ is the directional derivative of m in the direction of $\Delta \mathbf{w}^i$. We adopt Algorithm 18.3 ([**?**], p. 545) to choose μ^i and compute α^i at each iteration.

2.2 Data in `MATMPC`

There are four main data structs in `MATMPC`, namely *settings*, *opts*, *input* and *mem*. There are two features regarding these structs:

1. The data contained in these structs are pre-defined. However, the dimension and size of the data is not checked while running simulation. Users should check it carefully otherwise MATLAB would probably crash due to memory leak.

2. The memory needed for data in these structs are allocated a priori. Users can pause the simulation and investigate values in the data for debugging.

2.2.1 *settings*

T_s	Sampling time
Ts_st	Shooting interval
s	number of integration steps per interval
nx	No. of states
nu	No. of controls
ny	No. of outputs (references)
nyN	No. of outputs at terminal stage
np	No. of parameters (on-line data)
nc	No. of general constraints
ncN	No. of general constraints at terminal stage
nbx	No. of bounds on states
nbu	No. of bounds on controls
nbx_idx	indexes of states which are bounded
nbu_idx	indexes of controls which are bounded
N	No. of shooting points
$N2$	No. of partial condensing blocks
r	No. of input move blocks

Table 1: variables in *settings* struct

2.2.2 *opts*

hessian	Gauss_Newton, Generalized_Gauss_Newton
integrator	ERK4, IRK3, IRK3-DAE EKR4-CASADI
hessian	gauss_newton
condensing	default_full, no, blasfeo_full, partial_condensing
qpsolver	qpoases, qpoases_mb, quadprog_dense, hpipm_sparse hpipm_pcond, ipopt_dense, ipopt_sparse ipopt_partial_sparse, osqp_sparse, osqp_partial_sparse
hotstart	yes, no (only for qpoases)
shifting	yes, no (naive one interval backward shifting)
ref_type	0-time invariant, 1-time varying(no preview), 2-time varying(preview)
nonuniform_grid	0-deactivate, 1-activate
RTI	yes, no

Table 2: variables in *opts* struct

2.2.3 *input*

x	$[x_0, x_1, \ldots, x_N] \in \mathbb{R}^{n_x \times (N+1)}$
u	$[u_0, u_1, \ldots, u_{N-1}] \in \mathbb{R}^{n_u \times N}$
x_0	initial state measurement
lb	$[\underline{r}_0; \underline{r}_1; \ldots; \underline{r}_N] \in \mathbb{R}^{Nn_c + n_{c_N}}$

ub	$[\overline{r}_0; \overline{r}_1; \ldots; \overline{r}_N] \in \mathbb{R}^{Nn_c+n_{c_N}}$
lbu	$[\underline{u}_0, \underline{u}_1, \ldots, \underline{u}_{N-1}] \in \mathbb{R}^{n_u \times N}$
ubu	$[\overline{u}_0, \overline{u}_1, \ldots, \overline{u}_{N-1}] \in \mathbb{R}^{n_u \times N}$
lbx	$[\underline{x}_1, \underline{x}_2, \ldots, \underline{x}_N] \in \mathbb{R}^{nbx \times N}$
ubx	$[\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_N] \in \mathbb{R}^{nbx \times N}$
od	$[p_0, p_1, \ldots, p_N] \in \mathbb{R}^{n_p \times (N+1)}$
W	$[w_0, w_1, \ldots, w_{N-1}] \in \mathbb{R}^{n_y \times N}$
WN	$w_N \in \mathbb{R}^{n_{y_N}}$
λ	$[\lambda_0, \lambda_1, \ldots, \lambda_N] \in \mathbb{R}^{n_x \times (N+1)}$
μ_u	$[\mu_{u_0}; \mu_{u_1}; \ldots; \mu_{u_{N-1}}] \in \mathbb{R}^{Nn_u}$
μ_x	$[\mu_{x_0}; \mu_{x_1}; \ldots; \mu_{x_{N-1}}] \in \mathbb{R}^{Nnbx}$
μ	$[\mu_{g_0}; \mu_{g_1}; \ldots; \mu_{g_N}] \in \mathbb{R}^{Nn_c+n_{c_N}}$

Table 3: variables in *input* struct

2.2.4 *mem*

The *mem* struct contains a lot of fields, which are partially selected here for elaboration.

sqp_maxit	maximum number of SQP iterations
kkt_lim	tolerance on optimality
Q	Hessian w.r.t. state $\mathbb{R}^{n_x \times (N+1)n_x}$
S	Hessian w.r.t. state and control $\mathbb{R}^{n_x \times Nn_u}$
R	Hessian w.r.t. control $\mathbb{R}^{n_u \times Nn_u}$
A	state transition matrix $\mathbb{R}^{n_x \times Nn_x}$
B	control transition matrix $\mathbb{R}^{n_x \times Nn_u}$
Cgx	general constraint Jacobian w.r.t. state $\mathbb{R}^{n_c \times Nn_x}$
Cgu	general constraint Jacobian w.r.t. control $\mathbb{R}^{n_c \times Nn_u}$
CgN	general constraint Jacobian w.r.t. terminal state $\mathbb{R}^{n_{c_N} \times n_x}$
gx	cost function gradient w.r.t. state $\mathbb{R}^{n_x \times (N+1)}$
gu	cost function gradient w.r.t. control $\mathbb{R}^{n_u \times N}$
a	linearized dynamics residual $\mathbb{R}^{n_x \times N}$
$ds0$	initial value deviation $\mathbb{R}^{n_x}$
lc	general constraint QP lower bound $\mathbb{R}^{Nn_c+n_{c_N}}$
uc	general constraint QP upper bound $\mathbb{R}^{Nn_c+n_{c_N}}$
lb_du	control bound QP lower bound $\mathbb{R}^{Nn_u}$
ub_du	control bound QP upper bound $\mathbb{R}^{Nn_u}$
lb_dx	state bound QP lower bound $\mathbb{R}^{Nnbx}$
ub_dx	state bound QP upper bound $\mathbb{R}^{Nnbx}$
Hc	condensed Hessian $\mathbb{R}^{Nn_u \times Nn_u}$
Ccx	condensed state bound Jacobian $\mathbb{R}^{Nnbx \times Nn_u}$
Ccg	condensed general constraint Jacobian $\mathbb{R}^{(Nn_c+n_{c_N}) \times Nn_u}$
gc	condensed cost function gradient $\mathbb{R}^{Nn_u}$
lcc	condensed general constraint lower bound $\mathbb{R}^{(Nn_c+n_{c_N})}$
ucc	condensed general constraint upper bound $\mathbb{R}^{(Nn_c+n_{c_N})}$

lxc	condensed state bound lower bound $\mathbb{R}^{Nnbx}$
uxc	condensed state bound upper bound $\mathbb{R}^{Nnbx}$
dx	optimal state update $\mathbb{R}^{n_x \times (N+1)}$
du	optimal control update $\mathbb{R}^{n_u \times N}$
lambda_new	new multiplier w.r.t. eq constraint $\mathbb{R}^{n_x \times (N+1)}$
mu_new	new multiplier w.r.t. general constraint $\mathbb{R}^{Nn_c+n_{c_N}}$
mu_x_new	new multiplier w.r.t. state bound $\mathbb{R}^{Nnbx}$
mu_u_new	new multiplier w.r.t. control bound $\mathbb{R}^{Nn_u}$

Table 4: Important fields in *mem* struct

3 How to use

In this section, a brief introduction of how to run a simulation is given.

3.1 MATLAB Simulation

1. For the first step, users should refer to `MATMPC` examples for how to write their models. Double check the dimensions of all data are correct.
2. In "Model_Generation.m", choose your model by setting "settings.model=...". Run this script.
3. Once model codes are generated, open "Simulation.m". Change a variety of options that may fit your application. Run this script.
4. Open "draw.m" and write your own plotting codes. Run this script to see your results.

3.2 Simulink Simulation

1. For the first step, users should refer to `MATMPC` examples for how to write their models. Double check the dimensions of all data are correct.
2. In "Model_Generation.m", choose your model by setting "settings.model=...". Run this script.
3. Once model codes are generated, open "Initialization_Simulink.m". Change a variety of options that may fit your application. Also configure the initial conditions. Run this script.
4. Open "MATMPC_SIMULINK". Run the simulation for the "Inverted Pendulum" example. Get an idea of how the simulation works.
5. Re-write or create a new Simulink simulation for your application.

4 Advanced Functions

Currently, `MATMPC` supports two advanced and unique functions:

1. The curvature-like measure of nonlinearity (CMoN) NMPC schemes [**?**, **?**].
2. Sparse discretization.

4.1 Input move blocking

Sparse discretization can reduce the computational burden of solving an NLP online by reducing the number of decision variables. In MATMPC, there are two ways of doing this: 1) input move blocking and 2) non-uniform grid. The difference of these to ways are shown in [**?**], Fig. 1. As compared in [**?**], we strongly recommend to use input move blocking, rather than the non-uniform grid strategy.

In MATMPC, users can activate the input move blocking by choosing the QP solver as "qpoases_mb". As a result, MATMPC will automatically use Algorithm 1 and 2 in [**?**] to condense the NLP. By using this QP solver, users must provide their own move blocking structure in "Init_Memory.m", at line 443. In particular, "mem.r" is the number of blocks. "mem.index_T" is the indexing vector, where each element denotes the starting index of each input block. Note that this vector always starts from 0 hence is of length $r + 1$. Also note that, as remarked in [**?**], an inpropoer move blocking structure may lead to infeasibility and instability. Hence, the tuning of "mem.index_T" is often a trial-and-error procedure and has no theoretical guarantees in most cases.