Preprints of the 21st IFAC World Congress (Virtual)
Berlin, Germany, July 12-17, 2020

Automatic Code Generation Tool for Nonlinear Model Predictive Control with Jupyter [⋆]

S. Katayama [*] T. Ohtsuka [*]

[*] *Department of Systems Science, Graduate School of Informatics, Kyoto University, Sakyo-ku, Kyoto, 606-8501, Japan*

Abstract: We present an automatic code generation tool, `AutoGenU for Jupyter`, for nonlinear model predictive control (NMPC) with a user-friendly and interactive interface utilizing JupyterLab and Jupyter Notebook. We utilize a symbolic computation package SymPy for automatic C++ code generation. We also developed numerical solvers of NMPC using the continuation/GMRES (C/GMRES) method and multiple-shooting-based C/GMRES method in C++. `AutoGenU for Jupyter` provides the simulation environment of NMPC with these solvers and visualization of the simulation results. We give an example of code generation and numerical simulation of a swing-up control of a cart pole using `AutoGenU for Jupyter`.

Keywords: Predictive Control, Optimal Control, Nonlinear Control, Software Tools

1. INTRODUCTION

Nonlinear model predictive control (NMPC) (Magni et al. (2008)) has attracted much attention in both academic and industrial fields along due to dramatic improvements in numerical algorithms and CPUs. In NMPC, the finite horizon optimal control problem (FHOCP), on the basis of the dynamics of the system and the current state, is solved at each sampling time, and the initial value of the optimal control input is applied to the actual system. NMPC achieves state feedback control law by treating nonlinear dynamics and constraints explicitly as long as the FHOCP is solved within the given sampling period. However, the bottleneck of NMPC is the computational time to solve the FHOCP. Typically, we cannot solve the FHOCP analytically but only numerically because the system has nonlinear dynamics and nonlinear constraints. It may take more than the given sampling period to compute the numerical solution of the FHOCP when the sampling period is short, dimension of the variables is large, and dynamics of the system is complicated.

To solve the FHOCP numerically in a short computational time, various methods have been proposed. There are two types of such methods: one for solving the Hamilton-Jacobi-Bellman equation (HJBE) and the other for solving the two-point boundary-value problem (TPBVP). With an HJBE-based method, differential dynamics programming (DDP) approximates the value functions of the HJBE and updates the value function and the state trajectory iteratively (Tassa et al. (2008)). Several variants of the DDP are proposed (Sideris and Bobrow (2005); Tassa et al. (2012)). For TPBVP-based methods, many numerical algorithms based on Newton's method have been proposed. Diehl et al. (2005) proposed the real-time iteration (RTI) scheme that solves the quadratic programming (QP) at each sampling time. The continuation/GMRES (C/GMRES) method proposed by Ohtsuka (2004) tracks the optimal solution using the continuation method (Richter and DeCarlo (1983)).

Software tools as well as numerical algorithms have been proposed. `ACADO toolkit` (Houska et al. (2011)) generates efficient C code for the generalized-Gauss-Newton-based RTI scheme and has an interface with C++ and Matlab. Herceg et al. (2013) developed the `Multi-Parametric Toolbox (MPT)`, a MatLab toolbox that supports NMPC for the discrete-time piecewise affine system. `AutoGen` (Ohtsuka and Kodama (2002)) is an automatic C code generator for the real-time algorithm of NMPC proposed by Ohtsuka and Fujii (1997) and developed using the symbolic computation language Mathematica. `AutoGenU`, which was introduced by Ohtsuka (2004), was developed using Mathematica and supports automatic C code generation for the C/GMRES method. `AutoGenU for Maple` (Ohtsuka (2015)) generates C codes for the C/GMRES method using another symbolic computation language, i.e., Maple. Giftthaler M., et al. (2018) developed the `Control Toolbox` (CT), which provides NMPC solvers based on iLQR in C++. Deng and Ohtsuka (2018) developed a MatLab toolbox `ParMNMPC`, which generates efficient C/C++ codes of a Newton-type parallel computing method for NMPC (Deng and Ohtsuka (2019)). These software tools are written in Matlab, Mathematica, Maple, or C++. However, Python is the most popular programming language today because of its ease of development, numerous state-of-the-art open-source libraries, and interactive interfaces JupyterLab and Jupyter Notebook (Pérez and Granger (2007); Kluyver T., et al. (2016); Granger and Grout (2016)). Python also has a symbolic computation package SymPy (Meurer A., et al. (2017)), which makes automatic code generation possible. Therefore, it is worth developing a tool of NMPC written in Python and has an interface using JupyterLab and Jupyter Notebook. `CasADi` (Andersson

[⋆] This work was partly supported by JSPS KAKENHI Grant Number 15H02257.

et al. (2019)) is an open-source software for nonlinear optimization supporting C code generation and having Python API. It also has a Python interface specialized for NMPC, `MPCTools` (Risbeck and Rawlings (2015)). However, for more intuitive use, Jupyter interfaces are preferable.

In this paper, we present `AutoGenU for Jupyter`, an automatic code generation tool for NMPC with a user-friendly and interactive interface utilizing JupyterLab and Jupyter Notebook. We use SymPy for the symbolic computation and automatic C++ code generation, including simplification and common expression elimination. We also developed C/GMRES-based numerical solvers for NMPC in C++ for flexible use (e.g., for robotics). We implement not only the original single-shooting C/GMRES method, but also the multiple-shooting-based C/GMRES method with condensing (Shimizu et al. (2009)), which is not provided in the existing code generation tool (Ohtsuka (2015)). We also add the semi-smooth Fischer-Burmeister (FB) function for inequality constraints (Liao-McPherson et al. (2019)) in the code generation. This tool provides the simulation environment for NMPC using CMake and visualizes the simulation results using Matplotlib (Hunter (2007)) and seaborn (Waskom M., et al. (2012-2019)).

This paper is composed as follows. In Secion 2, we discuss the problem settings of NMPC. In Section 3, we introduce the C/GMRES and multiple-shooting-based C/GMRES methods. In Section 4, we present `AutoGenU for Jupyter`, and in Section 5, we give an example of code generation and numerical simulation using `AutoGenU for Jupyter`. In Section 6, we conclude our paper.

2. PROBLEM SETTINGS OF NONLINEAR MODEL PREDICTIVE CONTROL

We consider the nonlinear system

$$\dot{x}(t) = f(x(t), u(t), p(t)), \tag{1}$$

where $x(t) \in \mathbb{R}^n$ denotes the state vector, $u(t) \in \mathbb{R}^{m_u}$ the control input vector, and $p(t) \in \mathbb{R}^{m_p}$ the time-varying parameter. We also consider m_c dimensional equality constraints of the form

$$C(x(t), u(t), p(t)) = 0, \tag{2}$$

and m_h dimensional inequality constraints of the form

$$h(x(t), u(t), p(t)) \leq 0, \tag{3}$$

imposed on the system (1). Suppose that (3) is treated by the semi-smooth FB function. Note that inequality constraints can also be transformed into equality constraints (2) by introducing a dummy input (Ohtsuka (2004)) or considered in the cost function by introducing barrier functions (Nocedal and Wright (2006)). In NMPC, an FHOCP from the current time t to the finite future $t+T$ is solved each sampling time. That is, we want to find the optimal control input $u^*(t'; t; x(t))$ $(t \leq t' \leq t+T)$ that minimizes the cost function

$$J = \varphi(x(t+T), p(t+T)) + \int_t^{t+T} L(x(t', u(t'), p(t'))dt' \tag{4}$$

under (1)–(3), where $\varphi(\cdot,\cdot) : \mathbb{R}^n \times \mathbb{R}^{m_p} \to \mathbb{R}$ denotes the terminal cost and $L(\cdot,\cdot,\cdot) : \mathbb{R}^n \times \mathbb{R}^{m_u} \times \mathbb{R}^{m_p} \to \mathbb{R}$ denotes the stage cost. The time interval $[t, t+T]$ is called the horizon of NMPC. After obtaining the optimal control input $u^*(t'; t; x(t))$ $(t \leq t' \leq t+T)$ by solving the FHOCP, the initial value of the optimal control input, i.e., $u(t) = u^*(t; t; x(t))$, is applied to the actual system. To numerically obtain the optimal control input, we derive the Karush–Kuhn–Tucker (KKT) conditions, which are the necessary conditions of the optimal control. We introduce a new time axis on horizon τ $(0 \leq \tau \leq T)$ and a new state on the horizon $x^*(\tau; t)$, which denotes the state trajectory along the τ axis from $x(t)$ at $\tau = 0$. The FHOCP is then defined by finding the optimal control input along the τ axis $u^*(\tau; t)$ $(0 \leq \tau \leq T)$ that minimizes the cost function

$$J = \varphi(x^*(T; t), p(T; t)) + \int_0^T L(x^*(\tau; t), u^*(\tau; t), p(\tau; t))d\tau, \tag{5}$$

subject to

$$x^*(0; t) = x(t), \tag{6}$$

$$\frac{d}{d\tau} x^*(\tau; t) = f(x^*(\tau; t), u^*(\tau; t), p(\tau; t)), \tag{7}$$

$$C(x^*(\tau; t), u^*(\tau; t), p(\tau; t)) = 0, \tag{8}$$

and

$$h(x^*(\tau; t), u^*(\tau; t), p(\tau; t)) \leq 0. \tag{9}$$

Note that $p(\tau; t)$ corresponds to $p(\tau + t)$. To derive the KKT conditions, we introduce $\lambda \in \mathbb{R}^n$, $\mu \in \mathbb{R}^{m_c}$, and $\nu \in \mathbb{R}^{m_h}$, which denote the Lagrange multipliers for the state equation (7), equality constraints (8), and inequality constraints (9), respectively. We then introduce the Hamiltonian defined by

$$H(x, u, \lambda, \mu, \nu, p) := L(x, u, p) + \lambda^{\mathrm{T}} f(x, u, p) + \mu^{\mathrm{T}} C(x, u, p) + \nu^{\mathrm{T}} h(x, u, p). \tag{10}$$

The KKT conditions (Bryson and Ho (1975); Nocedal and Wright (2006)) are then given as

$$\lambda^*(T; t) = \varphi_x^{\mathrm{T}}(x^*(T; t), p(T; t)), \tag{11}$$

$$\dot{\lambda}^*(\tau; t) = - H_x^{\mathrm{T}}(x^*(\tau; t), u^*(\tau; t), \lambda^*(\tau; t), \mu^*(\tau; t), \nu^*(\tau; t), p(\tau; t)), \tag{12}$$

$$H_u^{\mathrm{T}}(x^*(\tau; t), u^*(\tau; t), \lambda^*(\tau; t), \mu^*(\tau; t), \nu^*(\tau; t), p(\tau; t)) = 0, \tag{13}$$

$$\nu^*(\tau; t) \geq 0,\ {\nu^*}^{\mathrm{T}}(\tau; t)\, h(x^*(\tau; t), u^*(\tau; t), p(\tau; t)) = 0. \tag{14}$$

Note that a scalar function with a subscript denotes the partial derivative of the scalar function with respect to the subscript. The unknown variables of the FHOCP are given by $x^*(\tau; t)$, $u^*(\tau; t)$, $\lambda^*(\tau; t)$, $\mu^*(\tau; t)$, and $\nu^*(\tau; t)$ for $0 \leq \tau \leq T$, which have to satisfy (6)–(14). Note that (9) and (14) denote the complementarity condition between $\nu^*(\tau; t)$ and $h(x^*(\tau; t), u^*(\tau; t), p(\tau; t))$. The semi-smooth FB function (Liao-McPherson et al. (2019)) is defined for a complementarity condition $a \geq 0$, $b \geq 0$, $ab = 0$ by

$$\psi(a, b) := \sqrt{a^2 + b^2 + \epsilon^2} - (a + b), \tag{15}$$

so that it has a property that

$$\psi(a, b) = 0 \Leftrightarrow a \geq 0, b \geq 0, \epsilon = \sqrt{2}ab, \tag{16}$$

where $\epsilon \geq 0$ relaxes the orthogonality $ab = 0$ to avoid $\psi(a, b)$ being undifferentiable when $(a, b) = (0, 0)$. Note that the original complementarity condition $a \geq 0$, $b \geq 0$, $ab = 0$ holds when $\epsilon \to 0$. We consider

$$\Psi(x, u, \nu, p) := \begin{bmatrix} \psi(\nu_{(1)}, -h_{(1)}(x, u, p)) \\ \vdots \\ \psi(\nu_{(m_h)}, -h_{(m_h)}(x, u, p)) \end{bmatrix} = 0 \tag{17}$$

instead of (9) and (14), where $\nu_{(i)}$ denotes the ith element of ν, $h_{(i)}(x,u,p)$ denotes the ith element of $h(x,u,p)$. Note that (17) is equivalent to (9) and (14) when $\epsilon \to 0$ and we can consider the original inequality constraints (3) with sufficiently small $\epsilon > 0$.

To numerically obtain the solution, we discretize the FHOCP. We divide the horizon into N steps and define $\Delta\tau := T/N$. We discretize the state on the horizon into $x_0^*(t), ..., x_N^*(t)$, control input into $u_0^*(t), ..., u_{N-1}^*(t)$, the Lagrange multipliers into $\lambda_1^*(t), ..., \lambda_N^*(t)$, $\mu_0^*(t), ..., \mu_{N-1}^*(t)$, $\nu_0^*(t), ..., \nu_{N-1}^*(t)$, and time-varying parameter into $p_0(t), ..., p_N(t)$. Note that $p_i(t)$ corresponds to $p(t + i\Delta\tau)$. We also discretize the KKT conditions (6)–(8), (11)–(13), and (17) into

$$x_0^*(t) = x(t), \tag{18}$$

$$x_{i+1}^*(t) = x_i^*(t) + f(x_i^*(t), u_i^*(t), p_i(t))\Delta\tau, \ i = 0, ..., N-1, \tag{19}$$

$$C(x_i^*(t), u_i^*(t), p_i(t)) = 0, \ i = 0, ..., N-1, \tag{20}$$

$$\lambda_N^*(t) = \varphi_x^{\mathrm{T}}(x_N^*(t), p_N(t)), \tag{21}$$

$$\begin{aligned}\lambda_i^*(t) = {} & \lambda_{i+1}^*(t) \\ & + H_x^{\mathrm{T}}(x_i^*(t), u_i^*(t), \lambda_{i+1}^*(t), \mu_i^*(t), \nu_i^*(t), p_i(t))\Delta\tau, \\ & \qquad i = 1, ..., N-1,\end{aligned} \tag{22}$$

$$\begin{aligned}& H_u^{\mathrm{T}}(x_i^*(t), u_i^*(t), \lambda_{i+1}^*(t), \mu_i^*(t), \nu_i^*(t), p_i(t)) = 0, \\ & \qquad i = 0, ..., N-1,\end{aligned} \tag{23}$$

and

$$\Psi(x_i^*(t), u_i^*(t), \nu_i^*(t), p_i(t)) = 0, \ i = 0, ..., N-1. \tag{24}$$

The optimal variables $x_0^*(t), ..., x_N^*(t)$, $u_0^*(t), ..., u_{N-1}^*(t)$, $\lambda_N^*(t)$, ..., $\lambda_1^*(t)$, $\mu_0^*(t), ..., \mu_{N-1}^*(t)$, and $\nu_0^*(t), ..., \nu_{N-1}^*(t)$ must satisfy (18)–(24), which define the TPBVP.

3. NUMERICAL ALGORITHMS OF NMPC BASED ON C/GMRES METHOD

3.1 C/GMRES method

The C/GMRES method regards $u_0^*(t)$, ..., $u_{N-1}^*(t)$, $\mu_0^*(t)$, ..., $\mu_{N-1}^*(t)$ and $\nu_0^*(t), ..., \nu_{N-1}^*(t)$ as variables to be determined. Under given $u_0^*(t), ..., u_{N-1}^*(t)$ $\mu_0^*(t), ..., \mu_{N-1}^*(t)$, and $\nu_0^*(t), ..., \nu_{N-1}^*(t)$, we can determine $x_0^*(t), ..., x_N^*(t)$ from (18) and (19) and $\lambda_N^*, ..., \lambda_1^*$ from (21) and (22). The errors in the optimality of given $u_0^*(t), ..., u_{N-1}^*(t)$, $\mu_0^*(t), ..., \mu_{N-1}^*(t)$, and $\nu_0^*(t), ..., \nu_{N-1}^*(t)$ are then computed by (23), (20), and (24). Let $U(t)$ be a vector composed of the variables to be determined and $F(U(t), x(t), t)$ be a vector-valued function composed of (23), (20), and (24), i.e., we define $U(t)$ and $F(U(t), x(t), t)$ as (25). Then the FHOCP is reduced to a nonlinear problem to find $U(t)$ satisfying $F(U(t), x(t), t) = 0$. This is also called the single-shooting method in contrast to the multiple-shooting method, which is introduced in the next subsection. With the C/GMRES method, we do not solve the nonlinear problem $F(U(t), x(t), t) = 0$ directly because it requires inefficient iterative search of the solution, such as Newton's method. Instead, the C/GMRES method tracks $U(t)$ without such iterations by computing the time derivative $\dot{U}(t)$ so that $U(t)$ satisfies $F(U(t), x(t), t) = 0$. If $U(0)$ satisfies $F(U(0), x(0), 0) = 0$, then we can track $U(t)$ satisfying $F(U(t), x(t), t) = 0$ by integrating $\dot{U}(t)$ satisfying the time derivatives of $F(U(t), x(t), t) = 0$. By applying the continuation method (Richter and DeCarlo (1983)) to $F(U(t), x(t), t) = 0$, we obtain

$$\dot{F}(U(t), x(t), t) = -\zeta F(U(t), x(t), t), \tag{26}$$

where ζ is a positive real value. The right side of (26) plays a role of stabilizing the numerical computation. Equation (26) is further transformed into

$$F_U \dot{U} + F_x \dot{x} + F_t = -\zeta F, \tag{27}$$

which can be seen as a linear problem of $\dot{U}$, e.g., as

$$F_U \dot{U} = -\zeta F - F_x \dot{x} - F_t. \tag{28}$$

Note that we omit the arguments in (27) and (28). With the C/GMRES method, the products of the Jacobians and vectors in (28) are computed efficiently by the finite difference approximation, and (28) is solved using the GMRES method (Kelly (1995)), a fast computational method for linear problems. After obtaining $\dot{U}(t)$ by solving (28), the solution is updated by

$$U(t + \Delta t) = U(t) + \dot{U}(t)\Delta t, \tag{29}$$

where $\Delta t > 0$ is a sampling period. The C/GMRES method solves (28) only once per update of $U(t)$, and the computational cost of the update of $U(t)$ corresponds to only one iteration with Newton's method.

Note that we may have to obtain the initial solution $U(0)$ satisfying $F(U(0), x(0), 0) = 0$ within a given sampling period. A strategy to compute $U(0)$ with a short computational time is to set the length of the horizon as a time-dependent smooth function such that $T(0) = 0$ and $T(t) \to T_f \ (t \to \infty)$, e.g.,

$$T(t) = T_f(1 - e^{-\alpha t}), \tag{30}$$

where T_f and α are positive values. Then $u_i^*(0) = u(0)$, $\mu_i^*(0) = \mu(0)$, $\nu_i^*(0) = \nu(0)$, $x_i^*(0) = x(0)$, and $\lambda_i^*(0) = \varphi_x^{\mathrm{T}}(x(0), p(0))$ hold for arbitrary i because $T(0) = 0$. We may obtain $u(0)$, $\mu(0)$, and $\nu(0)$ by solving

$$\begin{bmatrix} H_u^{\mathrm{T}}(x(0), u(0), \varphi_x^{\mathrm{T}}(x(0), p(0)), \mu(0), p(0)) \\ C(x(0), u(0), p(0)) \\ \Psi(x(0), u(0), \nu(0), p(0)) \end{bmatrix} = 0 \tag{31}$$

with a short time even with Newton's method because nonlinear problem (31) is sufficiently small.

3.2 Multiple-Shooting-Based C/GMRES Method

Problem Formulation In contrast to the single-shooting method, the multiple-shooting method regards the all variables $u_0^*(t), ..., u_{N-1}^*(t)$, $x_1^*(t), ..., x_N^*(t)$, $\lambda_1^*(t), ..., \lambda_N^*(t)$, $\mu_0^*(t), ..., \mu_{N-1}^*(t)$, and $\nu_0^*(t), ..., \nu_{N-1}^*(t)$ as variables to be determined of the TPBVP. Because the errors in optimality are expanded in all the variables, this formulation improves numerical stability from the single-shooting method in which the errors appear just in $U(t)$. Introducing

$$X(t) := \begin{bmatrix} x_1^{*\mathrm{T}}(t) & \lambda_1^{*\mathrm{T}}(t) & \cdots & x_N^{*\mathrm{T}}(t) & \lambda_N^{*\mathrm{T}}(t) \end{bmatrix}^{\mathrm{T}}, \tag{32}$$

then the FHOCP of the multiple-shooting method is reduced to the following nonlinear problem: find $U(t)$ defined by (25) and $X(t)$ defined by (32) satisfying $F(U(t), X(t), x(t), t)$ defined as the same form of (25) and $G(U(t), X(t), x(t), t)$ defined as (33).

$$U(t) := \begin{bmatrix} u_0^*(t) \\ \mu_0^*(t) \\ \nu_0^*(t) \\ \vdots \\ u_{N-1}^*(t) \\ \mu_{N-1}^*(t) \\ \nu_{N-1}^*(t) \end{bmatrix}, \quad F(U(t), x(t), t) := \begin{bmatrix} H_u^{\mathrm{T}}(x_0^*(t), u_0^*(t), \lambda_1^*(t), \mu_0^*(t), \nu_0^*(t), p_0(t)) \\ C(x_0^*(t), u_0^*(t), p_0(t)) \\ \Psi(x_0^*(t), u_0^*(t), \nu_0^*(t), p_0(t)) \\ \vdots \\ H_u^{\mathrm{T}}(x_{N-1}^*(t), u_{N-1}^*(t), \lambda_N^*(t), \mu_{N-1}^*(t), \nu_{N-1}^*(t), p_{N-1}(t)) \\ C(x_{N-1}^*(t), u_{N-1}^*(t), p_{N-1}(t)) \\ \Psi(x_{N-1}^*(t), u_{N-1}^*(t), \nu_{N-1}^*(t), p_{N-1}(t)) \end{bmatrix}. \tag{25}$$

$$G(U(t), X(t), x(t), t) := \begin{bmatrix} x_1^*(t) - x(t) - f(x(t), u_0^*(t), p_0(t))\Delta\tau \\ \lambda_1^*(t) - \lambda_2^*(t) - H_x^{\mathrm{T}}(x_1^*(t), u_1^*(t), \lambda_2^*(t), \mu_1^*(t), \nu_1^*(t), p_1(t))\Delta\tau \\ \vdots \\ x_{N-1}^*(t) - x_{N-2}^*(t) - f(x_{N-2}^*(t), u_{N-2}^*(t), p_{N-2}(t))\Delta\tau \\ \lambda_{N-1}^*(t) - \lambda_N^*(t) - H_x^{\mathrm{T}}(x_{N-1}^*(t), u_{N-1}^*(t), \lambda_N^*(t), \mu_{N-1}^*(t), \nu_{N-1}^*(t), p_{N-1}(t))\Delta\tau \\ x_N^*(t) - x_{N-1}^*(t) - f(x_{N-1}^*(t), u_{N-1}^*(t), p_{N-1}(t))\Delta\tau \\ \lambda_N^*(t) - \varphi_x^{\mathrm{T}}(x_N^*(t), p_N(t)) \end{bmatrix} = 0. \tag{33}$$

Condensing By using the continuation method, (25) and (33) are transformed into

$$F_U \dot{U} + F_X \dot{X} + F_x \dot{x} + F_t = -\zeta F, \tag{34}$$

and

$$G_U \dot{U} + G_X \dot{X} + G_x \dot{x} + G_t = -\zeta G, \tag{35}$$

and the solution U and X can be updated by integrating $\dot{U}$ and $\dot{X}$ as well as (29). For example, we can obtain $\dot{U}$ and $\dot{X}$ by solving

$$\begin{bmatrix} F_U & F_X \\ G_U & G_X \end{bmatrix} \begin{bmatrix} \dot{U} \\ \dot{X} \end{bmatrix} = -\zeta \begin{bmatrix} F \\ G \end{bmatrix} - \begin{bmatrix} F_x \\ G_x \end{bmatrix} \dot{x} - \begin{bmatrix} F_t \\ G_t \end{bmatrix}. \tag{36}$$

However, this problem is more expensive than that of the original C/GMRES method because the size of linear problem (36) is larger than that of (28). To reduce the computational cost in solving the linear problem of the multiple-shooting formulation, Shimizu et al. (2009) applied condensing (Bock and Plitt (1984)) and removed $\dot{X}$ and a part of $\dot{U}$ from (36). Note that for the condensing of a part of $\dot{U}$, F and U have to be partitioned as

$$F = [\bar{F}^{\mathrm{T}} \ \ \hat{F}^{\mathrm{T}}]^{\mathrm{T}}, \ \ U = [\bar{U}^{\mathrm{T}} \ \ \hat{U}^{\mathrm{T}}]^{\mathrm{T}} \tag{37}$$

so that the following three conditions are satisfied: 1) $\bar{F}$ and $\bar{U}$ have the same size. 2) $G_{\hat{U}} = 0$ and $\hat{F}_X = 0$ hold. 3) The product of $\hat{F}_{\hat{U}}^{-1}$ and an arbitrary vector can be obtained with small computational burden. Under 1)–3), we can remove $\dot{\hat{U}}$ from (36). For example, consider a saturation on the control input, i.e.,

$$u_{\min} \leq u_j(t), \ \ u_j(t) \leq u_{\max}, \tag{38}$$

where $j \in \{1, 2, ..., m_u\}$ specifies the constrained element of u. By introducing a dummy input $\hat{u} \in \mathbb{R}$, inequality constraints (38) are transformed into the equality constraint

$$u_j^2 + \hat{u}^2 - \left(\frac{u_{\max} - u_{\min}}{2} \right)^2 = 0. \tag{39}$$

The dummy input is treated as a variable to be determined in the TPBVP. To consider (39), a Lagrange multiplier for (39), $\mu \in \mathbb{R}$, is also appended to the variables to be determined. Then (39) satisfies the above three conditions for condensing, and we can remove the time derivatives of $\hat{u}$ and μ from the solution of the linear problem.

4. AUTOGENU FOR JUPYTER

4.1 Overview of AutoGenU for Jupyter

`AutoGenU for Jupyter` (Katayama (2018-2020)) is an open-source software tool for NMPC. It is mainly composed of three parts:

- AutoGenU.ipynb: A main interface of `AutoGenU for Jupyter` (we can rename the .ipynb file as we like).
- autogenu: A directory containing Python modules used in AutoGenU.ipynb for the automatic C++ code generation, building of C++ source files, execution and visualization of numerical simulations.
- include/cgmres, src: A directory containing C++ header files and source files of the C/GMRES and multiple-shooting-based C/GMRES methods. A subdirectory containing C++ header files and source files for numerical simulations are also included.

Through AutoGenU.ipynb, we can generate C++ source files of NMPC problem settings, build C++ source files, run numerical simulations, and draw graphs of the simulation results. Python modules in autogenu are developed just for AutoGenU.ipynb, i.e., developed under an assumption that users call them through AutoGenU.ipynb. On the other hand, C++ libraries can be used without AutoGenU.ipynb as long as users prepare C++ source files of NMPC problem settings. Hence, we explain all workflows of AutoGenU.ipynb and the C++ libraries. Note that we assume that AutoGenU.ipynb, autogenu, include, and src exist in the same working directory. We also assume that Python 3.6 or later, Jupyter notebook or JupyterLab, SymPy, Numpy, Matplotlib, seaborn, C++11 compiler, and CMake are installed in the environment.

Workflows of AutoGenU Figure 1 illustrates an overview of the workflows of `AutoGenU for Jupyter`. These workflows can be roughly divided into five parts. The first part is "problem description" in which users are required to input dimensions of x, u, $C(x, u, p)$ and $h(x, u, p)$ and define the symbolic functions $f(x, u, p)$, $C(x, u, p)$, $h(x, u, p)$, $L(x, u, p)$, and $\varphi(x, p)$. The second part, "symbolic computation", derives the TPBVP based on the symbolic functions defined in the previous part. The third part is "code generation", which generates C++ source

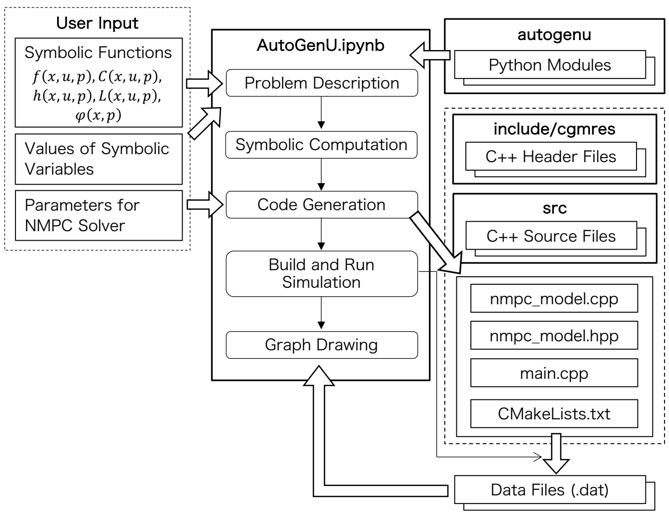


Fig. 1. Overview of AutoGenU for Jupyter

files of the NMPC problem settings. In this part, users are required to input the values of the symbolic variables used in the symbolic functions. AutoGenU.ipynb then generates nmpc_model.cpp and nmpc_model.hpp that describe a C++ class `NMPCModel` representing NMPC problem settings including $f(x,u,p)$, $C(x,u,p)$, $\varphi_x(x,p)$, $H_x(x,u,\lambda,\mu,\nu,p)$, $H_u(x,u,\lambda,\mu,\nu,p)$, $\Psi(x,u,\nu,p)$, and the parameters for them. The fourth part is "build and run simulation" in which users are required to select which NMPC solver to use, input parameters for the solver, and input parameters for numerical simulation. AutoGenU.ipynb then generates main.cpp, which calls the C/GMRES-based numerical solver, sets parameters of the solver, and calls a function to run the numerical simulations. A script for the building of C++ source files, CMakeLists.txt, is also generated in the same part. After that, C++ source files are built through CMakeLists.txt and numerical simulations are executed. After the numerical simulations, .dat files saving the time histories of the simulation results are generated. The last part is "graph drawing" in which the graphs of the simulation results are drawn based on the .dat files.

C/GMRES-Based Numerical Solvers of NMPC The following three solvers of NMPC are provided in src/solver directory:

- The original C/GMRES method (single shooting).
- The multiple-shooting-based C/GMRES method with condensing of $\dot{X}$.
- The multiple-shooting-based C/GMRES method with condensing of $\dot{X}$ and $\dot{\hat{U}}$ concerning (39).

To use these solvers, the solver class has to include C++ source files that define a class `NMPCModel`, which has the member functions shown in Fig. 2. The public member functions `stateFunc`, `phixFunc`, and `hxFunc` in Fig. 2 correspond to $f(x,u,p)$, $\varphi_x(x,p)$, and $H_x(x,u,\lambda,\mu,p)$, respectively. `huFunc` in Fig. 2 denotes a vector composed by $H_u(x,u,\lambda,\mu,p)$, $C(x,u,p)$, and $\Psi(x,u,p)$. Other public member functions `dimState` and `dimControlInput` return the dimensions of x and u and `dimConstraints` returns the total dimension of $C(x,u,p)$ and $h(x,u,p)$, respectively. Note that AutoGenU.ipynb automatically generates nmpc_model.hpp and nmpc_model.cpp, which describe `NMPCModel`.

```
class NMPCModel {
private:
  ...

public:
  void stateFunc(const double t, const double* x,
                 const double* u, double* f);
  void phixFunc(const double t, const double* x,
                double* phix);
  void hxFunc(const double t, const double* x,
              const double* u, const double* lmd,
              double* hx);
  void huFunc(const double t, const double* x,
              const double* u, const double* lmd,
              double* hu);
  int dimState() const;
  int dimControlInput() const;
  int dimConstraints() const;
  ...
};
```

Fig. 2. Necessary member functions of `NMPCModel`

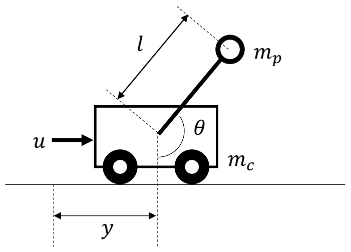


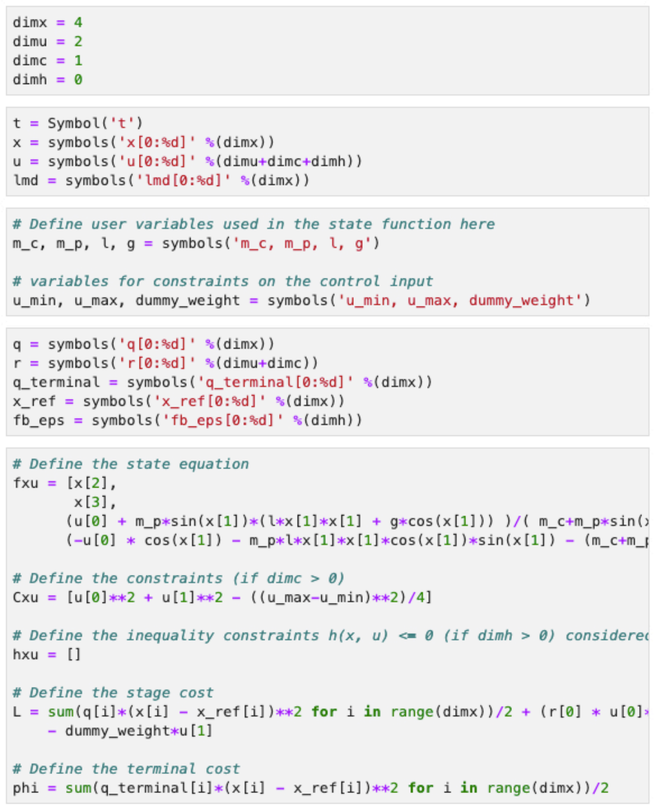
Fig. 3. Cart pole

5. APPLICATION EXAMPLE OF AUTOGENU FOR JUPYTER

In this section, we present an example of code generation and numerical simulation using `AutoGenU for Jupyter`. We consider a swing-up control of a cart pole, which is illustrated in Fig. 3, using the single-shooting C/GMRES method. The swing-up control of the pole in the cart pole is a difficult problem because it is under-actuated and its dynamics contain high nonlinearity. The state vector of the cart pole is given by $x := \begin{bmatrix} y & \theta & \dot{y} & \dot{\theta} \end{bmatrix}^{\mathrm{T}}$, and the state equation is given by

$$\dot{x} = f(x,u)$$
$$:= \begin{bmatrix} \dot{y} \\ \dot{\theta} \\ \dfrac{u + m_p \sin\theta (l\dot{\theta}^2 + g\cos\theta)}{m_c + m_p \sin^2\theta} \\ \dfrac{-u\cos\theta - m_p l \dot{\theta}^2 \cos\theta \sin\theta - (m_c + m_p) g \sin\theta}{l(m_c + m_p \sin^2\theta)} \end{bmatrix}. \tag{40}$$

We assume the physical parameters of the cart pole as $m_c = 2$ [kg], $m_p = 0.2$ [kg], and $l = 0.5$ [m]. We consider imposing limits on the control input, i.e., $u_{\min} \le u$, $u \le u_{\max}$ with $u_{\min} = -15$, $u_{\max} = 15$. We introduce a dummy input $\hat{u} \in \mathbb{R}$ and transform these inequality constraints into the equality constraint of the form (39). The objective is to invert the pole and stabilize the cart, i.e., to make the



```
dimx = 4
dimu = 2
dimc = 1
dimh = 0
```

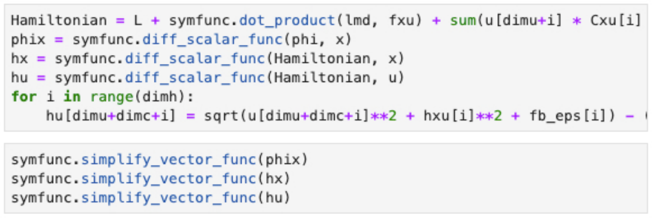
```
t = Symbol('t')
x = symbols('x[0:%d]' %(dimx))
u = symbols('u[0:%d]' %(dimu+dimc+dimh))
lmd = symbols('lmd[0:%d]' %(dimx))
```

```
# Define user variables used in the state function here
m_c, m_p, l, g = symbols('m_c, m_p, l, g')

# variables for constraints on the control input
u_min, u_max, dummy_weight = symbols('u_min, u_max, dummy_weight')
```

```
q = symbols('q[0:%d]' %(dimx))
r = symbols('r[0:%d]' %(dimu+dimc))
q_terminal = symbols('q_terminal[0:%d]' %(dimx))
x_ref = symbols('x_ref[0:%d]' %(dimx))
fb_eps = symbols('fb_eps[0:%d]' %(dimh))
```

```
# Define the state equation
fxu = [x[2],
       x[3],
      (u[0] + m_p*sin(x[1])*(l*x[1]*x[1] + g*cos(x[1])) )/( m_c+m_p*sin(
      (-u[0] * cos(x[1]) - m_p*l*x[1]*x[1]*cos(x[1])*sin(x[1]) - (m_c+m_

# Define the constraints (if dimc > 0)
Cxu = [u[0]**2 + u[1]**2 - ((u_max-u_min)**2)/4]

# Define the inequality constraints h(x, u) <= 0 (if dimh > 0) considere
hxu = []

# Define the stage cost
L = sum(q[i]*(x[i] - x_ref[i])**2 for i in range(dimx))/2 + (r[0] * u[0]
    - dummy_weight*u[1]

# Define the terminal cost
phi = sum(q_terminal[i]*(x[i] - x_ref[i])**2 for i in range(dimx))/2
```

Fig. 4. “Problem description” part of AutoGenU.ipynb



```
Hamiltonian = L + symfunc.dot_product(lmd, fxu) + sum(u[dimu+i] * Cxu[i]
phix = symfunc.diff_scalar_func(phi, x)
hx = symfunc.diff_scalar_func(Hamiltonian, x)
hu = symfunc.diff_scalar_func(Hamiltonian, u)
for i in range(dimh):
    hu[dimu+dimc+i] = sqrt(u[dimu+dimc+i]**2 + hxu[i]**2 + fb_eps[i]) -
```

```
symfunc.simplify_vector_func(phix)
symfunc.simplify_vector_func(hx)
symfunc.simplify_vector_func(hu)
```

Fig. 5. “Symbolic computation” part of AutoGenU.ipynb

state converge to $x_{\rm ref} = [0 \ \pi \ 0 \ 0]^{\rm T}$. For this purpose, we design the terminal cost as

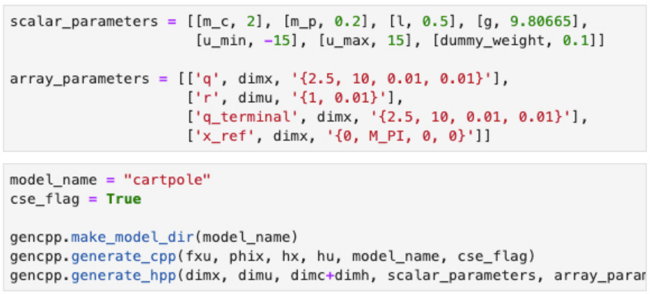
$$\varphi(x) = \frac{1}{2}(x - x_{\rm ref})^{\rm T} Q (x - x_{\rm ref}), \tag{41}$$

where $Q = \text{diag}\{2.5, 10, 0.01, 0.01\}$, and the stage cost as

$$L(x, u) = \frac{1}{2}(x - x_{\rm ref})^{\rm T} Q (x - x_{\rm ref}) + \frac{1}{2} r u^2 - \hat{r}\hat{u}, \tag{42}$$

where $r = 1$, $\hat{r} = 0.1$. Note that the last term in (42) is to decide the sign of $\hat{u}$ uniquely and avoid failures in the numerical calculation (see Ohtsuka (2004)).

The user inputs of these problem definitions on AutoGenU.ipynb and code generation of C++ source files of the NMPC problem settings are illustrated in Figs. 4, 5, and 6 corresponding to the “problem description”, “symbolic computation”, and “code generation” parts, respectively. In the “problem description” part, we first input the dimensions of the state, control input, equality constraints, and inequality constraints treated by the semi-smooth FB function, which are defined as `dimx`, `dimu`, `dimc`, and `dimh`, respectively, in the first cell of Fig. 4. Note that the control input is now given by $[u \ \hat{u}]^{\rm T} \in \mathbb{R}^2$



```
scalar_parameters = [[m_c, 2], [m_p, 0.2], [l, 0.5], [g, 9.80665],
                     [u_min, -15], [u_max, 15], [dummy_weight, 0.1]]

array_parameters = [['q', dimx, '{2.5, 10, 0.01, 0.01}'],
                    ['r', dimu, '{1, 0.01}'],
                    ['q_terminal', dimx, '{2.5, 10, 0.01, 0.01}'],
                    ['x_ref', dimx, '{0, M_PI, 0, 0}']]
```

```
model_name = "cartpole"
cse_flag = True

gencpp.make_model_dir(model_name)
gencpp.generate_cpp(fxu, phix, hx, hu, model_name, cse_flag)
gencpp.generate_hpp(dimx, dimu, dimc+dimh, scalar_parameters, array_para
```

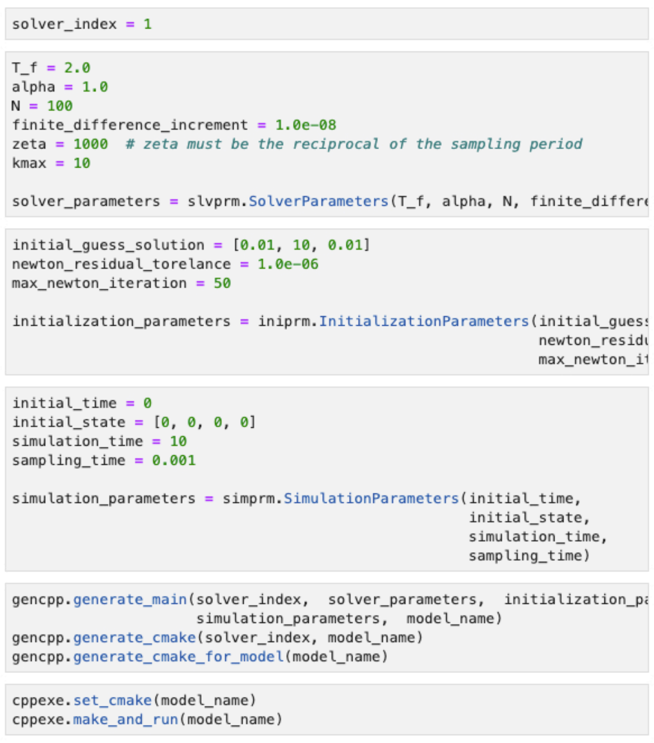
Fig. 6. “Code generation” part of AutoGenU.ipynb

i.e., `dimu` is 2. `dimc` is given by 1 and `dimh` by 0. The symbolic variables t, x, u, and λ are then defined on AutoGenU.ipynb, as shown in the second cell of Fig. 4. We then define symbolic variables used in $f(x, u, p)$, $C(x, u, p)$, $h(x, u, p)$, $L(x, u, p)$, $\varphi(x, p)$, as shown in the third and fourth cells of Fig. 4, and define these symbolic functions, as shown in the fifth cell of Fig. 4. In the next part, i.e., the “symbolic computation” part, $\phi_x(x, p)$, $H_x(x, u, \lambda, \mu, p)$, $H_u(x, u, \lambda, \mu, p)$, and $\Psi(x, u, \nu, p)$ are computed symbolically, as shown in the first cell of Fig. 5. The second cell simplifies these symbolic functions. In the “code generation” part, we input the values of the symbolic variables including physical parameters of the cart pole and weight parameters in the cost function, as shown in the first cell of Fig. 6. We also define the name of the directory where the C++ source files are generated and decide whether to use common subexpression elimination (CSE) in the code generation by `cse_flag`. If `cse_flag` is `True`, new variables are introduced to replace the common terms in the generated codes so that the computational time is reduced. After these settings, `gencpp.generate_cpp` and `gencpp.generate_hpp` generate nmpc_model.cpp and nmpc_model.hpp that contain class `NMPCModel`. Figure 7 shows an example of the generated nmpc_model.cpp with setting `cse_flag` by `True`. In Fig. 7, `x0`, ..., `x4` are introduced for CSE. The generated nmpc_model.hpp defines `NMPCModel`, as shown in Fig. 2. Note that the generated nmpc_model.hpp also contains the physical parameters of the cart pole and weight parameters in the cost function as its private member variables.

Next, we introduce the “build and run simulation” part, which is illustrated in Fig. 8. We first choose which solver to use to implement NMPC from the original C/GMRES method, multiple-shooting-based C/GMRES method with condensing of $\dot{X}$, and that with condensing of $\dot{X}$ and $\dot{\hat{U}}$. In this example, we choose the original C/GMRES method, as shown in the first cell of Fig. 8. Next, we have to set parameters for the solver, those for the initialization of the solution of NMPC, and those for numerical simulation. We set the parameters for the horizon in (30) as $T_f = 2$ and $\alpha = 1$, number of discretizations of the horizon as $N = 100$, stabilization parameter as $\zeta = 1000$, finite difference increment for the difference approximation as $h = 1.0 \times 10^{-8}$, and maximum number of the GMRES iterations as $k_{\rm max} = 10$. These parameter settings are shown in the second cell of Fig. 8. The parameters for the initialization of the solution are required for Newton's method to solve (31). We set the initial guess of the solution of Newton's method as $[0.1, \ 0.1, \ 0.1]^{\rm T}$, resid-

```
#include "nmpc_model.hpp"

void NMPCModel::stateFunc(const double t, const double*
   x, const double* u, double* f) {
     double x0 = sin(x[1]);
     double x1 = 1.0/(m_c + m_p*pow(x0, 2));
     double x2 = cos(x[1]);
     double x3 = l*pow(x[1], 2);
     double x4 = m_p*x0;
     f[0] = x[2];
     f[1] = x[3];
     f[2] = x1*(u[0] + x4*(g*x2 + x3));
     f[3] = x1*(-g*x0*(m_c + m_p) - u[0]*x2 - x2*x3*x4)/l
        ;
}

void NMPCModel::phixFunc(const double t, const double* x
   , double* phix) {
     ...
}

void NMPCModel::hxFunc(const double t, const double* x,
   const double* u, const double* lmd, double* hx) {
     ...
}

void NMPCModel::huFunc(const double t, const double* x,
   const double* u, const double* lmd, double* hu) {
     ...
}
```

Fig. 7. Example of generated nmpc_model.cpp with setting `cse_flag` by `True`. Details are omitted.



```
solver_index = 1
```

```
T_f = 2.0
alpha = 1.0
N = 100
finite_difference_increment = 1.0e-08
zeta = 1000  # zeta must be the reciprocal of the sampling period
kmax = 10

solver_parameters = slvprm.SolverParameters(T_f, alpha, N, finite_differe
```

```
initial_guess_solution = [0.01, 10, 0.01]
newton_residual_torelance = 1.0e-06
max_newton_iteration = 50

initialization_parameters = iniprm.InitializationParameters(initial_gues
                                                            newton_residu
                                                            max_newton_it
```

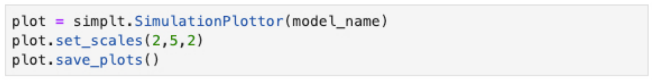
```
initial_time = 0
initial_state = [0, 0, 0, 0]
simulation_time = 10
sampling_time = 0.001

simulation_parameters = simprm.SimulationParameters(initial_time,
                                                    initial_state,
                                                    simulation_time,
                                                    sampling_time)
```

```
gencpp.generate_main(solver_index,  solver_parameters,  initialization_pa
                     simulation_parameters,  model_name)
gencpp.generate_cmake(solver_index, model_name)
gencpp.generate_cmake_for_model(model_name)
```

```
cppexe.set_cmake(model_name)
cppexe.make_and_run(model_name)
```

Fig. 8. “Build and run simulation” part of AutoGenU.ipynb

ual of torelance of Newton’s method as 1.0×10^{-6}, and the maximum number of the Newton’s iterations as 50, as depicted in the third cell of Fig. 8. Note that the solution of Newton’s method is composed as u, $\hat{u}$, and the Lagrange multiplier with respect to (39), i.e., its dimension is given by 3. In this example, we perform 10-s numerical simulation by setting the sampling period as 1 ms and initial state as $[0 \ \ 0 \ \ 0 \ \ 0]^{\mathrm{T}}$. The parameters for numerical simulation are input on AutoGenU.ipynb, as shown in the fourth cell of Fig. 8. After setting these parameters, main.cpp, which calls a function to execute the numerical simulations, and CMakeLists.txt are generated by `gencpp.generate_main`, `gencpp.generate_cmake`, and `gencpp.generate_cmake_for_model` in the fifth cell of Fig. 8. The last cell of Fig. 8 then builds C++ codes based on the CMakeLists.txt and executes numerical simulations.



```
plot = simplt.SimulationPlottor(model_name)
plot.set_scales(2,5,2)
plot.save_plots()
```

Fig. 9. “Graph drawing” part of AutoGenU.ipynb

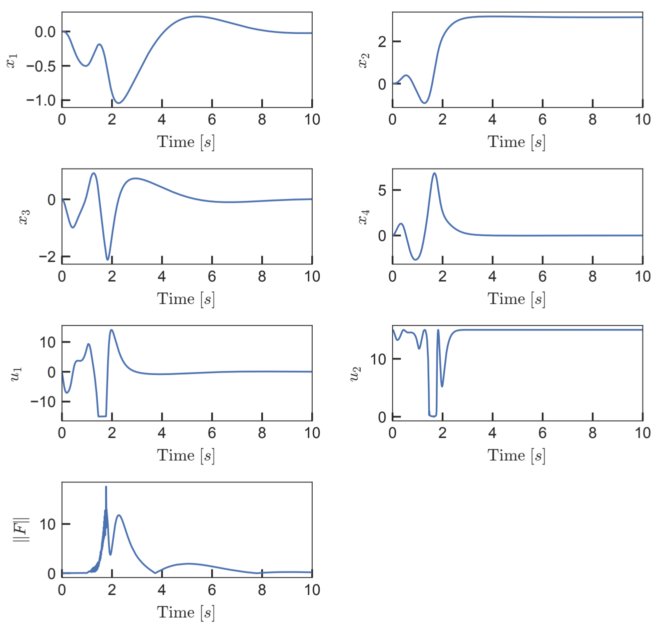


Fig. 10. Simulation results of controlling the cart pole

Figure 9 shows the “graph drawing” part. In this part, we set sizes of the graph, font, and the spaces between graphs by using `plot.set_scales`. Figure 10 shows the simulation results of swing-up control of the cart pole drawn by AutoGenU.ipynb. In this figure, u_1 is the actual control input applied to the cart pole, u_2 is the dummy input, and $\|F\|$ is the norm of (25). We can successfully swing-up the pole by satisfying the inequality constraints on the control input. The average computational time of the control update is 0.18 ms on a 1.6-GHz Intel Core i5 CPU, and we achieved real-time NMPC.

In this paper, we introduced just an example of controlling the cart pole using the single-shooting C/GMRES method. Katayama (2018-2020) gives other examples using the multiple-shooting-based methods and the semi-smooth FB function.

6. CONCLUSIONS

We presented a tool of NMPC having a user-friendly interface utilizing JupyterLab and Jupyter Notebook, `AutoGenU for Jupyter`. This tool provides automatic code generation of C++ source files representing NMPC problem settings using the symbolic computational package SymPy. It also supports the building of C++ source files with CMakeLists.txt, runs numerical simulations, and creates graphs of the simulation results. Future work includes applying the C++ C/GMRES-based NMPC solvers to ROS/ROS2 for easy application of the C/GMRES-based NMPC solvers to deploy on actual systems. Parallelization of the multiple-shooting-based C/GMRES method is also for future work to increase its speed.

REFERENCES

Andersson, J.A.E., Gillis, J., Horn, G., Rawlings, J.B., and Diehl, M. (2019). CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*, 11(1), 1–36.

Bock, H.G. and Plitt, K.J. (1984). A multiple shooting algorithm for direct solution of optimal control problems. *IFAC Proceedings Volumes*, 17(2), 1603 – 1608.

Bryson, A.E. and Ho, Y.C. (1975). *Applied Optimal Control: Optimization, Estimation, and Control.* CRC Press.

Deng, H. and Ohtsuka, T. (2018). A parallel code generation toolkit for nonlinear model predictive control. In *2018 IEEE Conference on Decision and Control (CDC)*, 4920–4926.

Deng, H. and Ohtsuka, T. (2019). A parallel Newton-type method for nonlinear model predictive control. *Automatica*, 109, 108560.

Diehl, M., Bock, H., and Schlöder, J.P. (2005). A real-time iteration scheme for nonlinear optimization in optimal feedback control. *SIAM J. Control and Optimization*, 43, 1714–1736.

Giftthaler M., et al. (2018). The Control Toolbox - an open-source C++ library for robotics, optimal and model predictive control. In *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, 123–129.

Granger, B. and Grout, J. (2016). JupyterLab: Building blocks for interactive computing. URL `https://jupyterlab.readthedocs.io/en/latest/`.

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

Houska, B., Ferreau, H.J., and Diehl, M. (2011). ACADO Toolkit – An Open Source Framework for Automatic Control and Dynamic Optimization. *Optimal Control Applications and Methods*, 32(3), 298–312.

Hunter, J.D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science and Engineering*, 9(3), 90–95.

Katayama, S. (2018-2020). AutoGenU for Jupyter. URL `https://github.com/mayataka/autogenu-jupyter`.

Kelly, C.T. (1995). *Iterative Methods for Linear and Nonlinear Equations.* Frontiers in Applied Mathematics. SIAM.

Kluyver T., et al. (2016). Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, 87 – 90.

Liao-McPherson, D., Huang, M., and Kolmanovsky, I. (2019). A regularized and smoothed fischer–burmeister method for quadratic programming with applications to model predictive control. *IEEE Transactions on Automatic Control*, 64(7), 2937–2944.

Magni, L., Raimondo, D.M., and Allgöwer, F. (2008). *Nonlinear Model Predictive Control: Towards New Challenging Applications*, volume 384. Springer.

Meurer A., et al. (2017). SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3, e103.

Nocedal, J. and Wright, S.J. (2006). *Numerical Optimization.* Springer, second edition.

Ohtsuka, T. (2004). A continuation/GMRES method for fast computation of nonlinear receding horizon control. *Automatica*, 40(4), 563 – 574.

Ohtsuka, T. (2015). A tutorial on C/GMRES and automatic code generation for nonlinear model predictive control. In *2015 European Control Conference (ECC)*, 73–86.

Ohtsuka, T. and Fujii, H.A. (1997). Real-time optimization algorithm for nonlinear receding-horizon control. *Automatica*, 33(6), 1147 – 1154.

Ohtsuka, T. and Kodama, A. (2002). Automatic code generation system for nonlinear receding horizon control. *Transactions of the Society of Instrument and Control Engineers*, 38(7), 617–623.

Pérez, F. and Granger, B.E. (2007). IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3), 21–29.

Richter, S.L. and DeCarlo, R.A. (1983). Continuation methods: Theory and applications. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-13(4), 459–464.

Risbeck, M.J. and Rawlings, J.B. (2015). MPCTools: Nonlinear model predictive control tools for CasADi (Python interface). URL `https://bitbucket.org/rawlings-group/mpc-tools-casadi`.

Shimizu, Y., Ohtsuka, T., and Diehl, M. (2009). A real-time algorithm for nonlinear receding horizon control using multiple shooting and continuation/Krylov method. *International Journal of Robust and Nonlinear Control*, 19(8), 919–936.

Sideris, A. and Bobrow, J.E. (2005). An efficient sequential linear quadratic algorithm for solving nonlinear optimal control problems. In *Proceedings of the 2005, American Control Conference, 2005.*, 2275–2280 vol. 4.

Tassa, Y., Erez, T., and Smart, B. (2008). Receding horizon differential dynamic programming. In *Advances in Neural Information Processing Systems 20*, 1465–1472.

Tassa, Y., Erez, T., and Todorov, E. (2012). Synthesis and stabilization of complex behaviors through online trajectory optimization. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 4906–4913.

Waskom M., et al. (2012-2019). seaborn: statistical data visualization. URL `https://seaborn.pydata.org/`.