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

Towards an Open Toolchain for Fast Nonlinear MPC for Serial Robots [⋆]

Alejandro Astudillo [∗] **Joris Gillis** [∗] **Wilm Decré** [∗]
Goele Pipeleers [∗] **Jan Swevers** [∗]

[∗] *MECO Research Team, Dept. of Mechanical Engineering, KU Leuven.*
Flanders Make - DMMS-M, Leuven, Belgium.
(e-mail: {alejandro.astudillovigoya, joris.gillis, wilm.decre, goele.pipeleers, jan.swevers}@kuleuven.be).

Abstract: This paper presents an open toolchain tailored for deployment of nonlinear model predictive control for serial robots. The toolchain provides a direct workflow from problem definition to solution deployment on a serial robot based on open-source software. Thus, we provide an insightful selection of modules for rigid body dynamics, numerical optimization, and robot control, and a strategy to make them cooperate in a way that is efficient in terms of computation and engineering time. A detailed numerical study is presented for path-following MPC on a 7-degrees-of-freedom robot, showing the efficiency and ease of use of the presented toolchain while comparing its modules with other tools.

Keywords: Nonlinear model predictive control; Serial robots; Real-time optimization.

1. INTRODUCTION

Automated solutions with robots are of vital importance in increasing quality, safety, competitiveness, and profit margin within industrial processes. Modern industry requires robots to execute autonomous tasks while being easy to setup, safe to work with, and robust. This calls for development of tools that simplify the workflow of robot-based solution deployment, while ensuring satisfaction of user-defined constraints.
Versatility and high workspace-to-robot-size ratio are very important features of serial robots within industry. Such robots consist of series of rigid links connected by joint actuators, where the last link in the chain is called end-effector. These robots usually perform repetitive tasks which are either hard-coded or controlled by simple, reactive controllers. Such controllers cannot respond well to disturbances nor changes in the environment of the robot.
Model predictive control (MPC) is an optimal control technique which optimizes the performance of a system, based on a defined criterion or objective. This optimization contemplates the satisfaction of input and state constraints. MPC for serial robots is an active research topic since it can be used to actively control a robot accounting for predicted nonlinear dynamics of robots and obstacles, actuator limits, safety constraints, and a desired performance objective. Although an extensive literature review, see Verschueren et al. (2019), Janeček et al. (2017), Englert et al. (2019), showed that there are multiple frameworks for optimization and MPC, none of them is both tailored for serial robots and simple to use. These frameworks need the user to explicitly define system dynamics or kinematics, system limits, the optimization problem to be solve, as well as requiring the user to employ additional software to deploy the solution.
In this work we propose a toolchain for nonlinear MPC of serial robots, entirely based on open-source software. The presented toolchain may be of great interest to researchers, roboticists, and control engineers, even though all theoretical concepts described in this paper are already known. Its main advantage is that it eases the workflow from problem definition to solution deployment in a serial robot, requiring few user-inputs and using open-source tools with computationally-efficient algorithms while allowing a great deal of flexibility.
The remainder of the paper is structured as follows. First, Section 2 introduces the type of optimization problems the toolchain aims to solve and the algorithmic background on which the solution is based. Next, the structure and workflow of the toolchain are presented in Section 3. Afterwards, Section 4 presents results and discussion of a numerical example. We close the paper with concluding remarks.

1.1 Notation

For a variable $x \in \mathbb{R}^{n_x}$, let us define $\dot{x} \in \mathbb{R}^{n_x}$ as its element-wise derivative with respect to time. For a nonempty closed convex set W, we define operator $\mathbf{\Pi}_W(v) = \mathbf{argmin}_{w \in W} ||w - v||$ as the projection onto W, and operator $\mathbf{dist}_W(v) = \mathbf{inf}_{w \in W} ||w - v||$ as distance from W. Overlined variables $\overline{w}$ define local variables in an algorithm. Subindex ee represents the end-effector of the robot.

[⋆] The authors would like to thank Flanders Make SBO MULTIROB: "Rigorous approach for programming and optimal control of multi-robot systems", FWO project G0A6917N of the Research Foundation - Flanders (FWO - Flanders), and KU Leuven-BOF PFV/10/002 Centre of Excellence: Optimization in Engineering (OPTEC) for supporting this research. Flanders Make is the Flemish strategic research centre for the manufacturing industry.

2. PRELIMINARIES

This section briefly discusses preliminary concepts and algorithms on which the toolchain relies. We first describe concepts on MPC and robot modeling, and then introduce the algorithms used to solve optimization problems within the toolchain.

2.1 Multiple shooting for optimal control

Continuous-time nonlinear dynamics can be described as an ordinary differential equation $\dot{x} = \xi(x, u, t)$, where $x \in \mathbb{R}^{n_x}$ is the system state vector, $u \in \mathbb{R}^{n_u}$ is the system input vector, and t is time. Continuous-time dynamics ξ can be discretized by means of a explicit Runge-Kutta integrator, which leads to the discrete-time system

$$x_{k+1} = \xi_d(x_k, u_k). \tag{1}$$

The multiple shooting method, introduced by Bock and Plitt (1984), divides the interval over which a function is discretized in $N \in \mathbb{N}$ subintervals, based on a time grid $T_{ms} := \{t_k \; : \; k \in (0, N)\}$. Therefore, by introducing grid state variables $(x_0, x_1, ..., x_N)$ and input variables $(u_0, u_1, ..., u_{N-1})$, one can formulate independent dynamic constraints on each grid point of T_{ms} as in (1), leading to a parallelizable set of dynamic constraints.

2.2 Model predictive control

Model predictive controllers allow controlling a nonlinear system while optimizing its performance and satisfying contraints by solving an optimal control problem (OCP) at every sample time δ_t. When using multiple shooting to define dynamic constraints, such OCP has the form of

$$\begin{aligned}
&\underset{w}{\text{minimize}} && V_N(x_N) + \sum_{k=0}^{N-1} V(x_k, u_k) && \text{(2a)}\\
&\text{subject to} && x_0 - p = \mathbf{0}_{n_x}, && \text{(2b)}\\
& && x_{k+1} - \xi_d(x_k, u_k) = \mathbf{0}_{n_x}, && \text{(2c)}\\
& && \zeta(x_k, l_k) \in \mathcal{Z}, && \text{(2d)}\\
& && w \in W, \quad k \in [0, N-1], && \text{(2e)}
\end{aligned}$$

where N is the prediction horizon, $\{V_N(x_N), V(x_k, u_k)\}$ define a performance objective, $p \in \mathbb{R}^{n_x}$ is a parameter corresponding to the estimation of the current state vector, l_k are slack variables, $\zeta(x_k, l_k)$ are path constraints, $\mathcal{Z}$ is a closed set, and $w = \left[x_0^T \; u_0^T \; l_0^T \cdots u_{N-1}^T \; l_{N-1}^T \; x_N^T\right]^T \in \mathbb{R}^{n_w}$ is the vector of decision variables belonging to the set $W := \{w \in \mathbb{R}^{n_w} : w_{min} \leq w \leq w_{max}\}$. OCP (2) can be represented in a compressed manner as

$$\begin{aligned}
&\min_{w} && f(w, p) && \text{(3a)}\\
&\text{s.t.} && g(w, p) \in C, && \text{(3b)}\\
& && w \in W, && \text{(3c)}
\end{aligned}$$

where $f(w, p)$ and $g(w, p)$ are possibly-nonconvex smooth functions, and C is a closed convex set.

2.3 Robot modeling

For serial robots with n_{dof} degrees of freedom, the state vector is usually defined as $x = [q^T, \dot{q}^T]^T$, where $q, \dot{q} \in \mathbb{R}^{n_{\text{dof}}}$ are robot-joint angular positions and velocities respectively, while input vector is selected as $u = \tau$, where $\tau \in \mathbb{R}^{n_{\text{dof}}}$ is the joint torque vector. Serial robots are restricted by their kinematics, which for the j-th link of the robot may be described by the position vector $\mathbf{p}_j(q) \in \mathbb{R}^3$ and the rotation matrix $\mathcal{R}_j(q) \in \mathbb{R}^{3\times 3}$.

2.4 Proximal Averaged Newton-type method for Optimal Control

Let us define the bound-constrained optimization problem

$$\min_{w \in W} \quad \phi(w), \tag{4}$$

with $\phi(w)$ being a possibly-nonconvex smooth function, which may also depend on p. Problem (4) can be solved by means of the *projected gradient operator*

$$w^{\nu+1} = T_\gamma(w^\nu) := \mathbf{\Pi}_W(w^\nu - \gamma \nabla\phi(w^\nu)), \tag{5}$$

for some parameter $\gamma > 0$ and current iteration ν. If $\phi(w)$ is continuously differentiable, $\nabla\phi(w)$ is Lipschitz-continuous with Lipschitz-constant L_ϕ, and $\gamma \in (0, \frac{2}{L_\phi})$, then all accumulation points w^* of the *projected gradient operator* are also fixed points, *i.e.*, w^* is a zero of the *fixed-point residual operator*

$$R_\gamma(w) := w - T_\gamma(w). \tag{6}$$

The *proximal averaged Newton-type method for optimal control* (PANOC) is a matrix-free line-search method proposed in Stella et al. (2017) for iteratively solving problem (4), up to some tolerance $\epsilon > 0$, by using (5). PANOC approximates, with superlinear convergence, a solution w^* of (4) that satisfies

$$||R_\gamma(w^*)||_\infty < \epsilon. \tag{7}$$

The complete PANOC algorithm is shown in Algorithm 1. PANOC uses a globalization technique where the *forward-backward envelope* (FBE)

$$\varphi_\gamma(w) = \phi(w) - \frac{\gamma}{2}||\nabla\phi(w)||^2 + \frac{1}{2\gamma}\mathbf{dist}^2_W(w - \gamma\nabla\phi(w)), \tag{8}$$

is used as a merit function. FBE is a real-valued and continuous function which shares the same minima with (4) if $\gamma \in (0, \frac{1}{L_\phi})$. Thus, by solving the unconstrained minimization of (8), the solution of (4) is obtained.
To speed-up convergence, the solution update in PANOC combines averaged safe projected gradient updates from (5) and fast quasi-Newtonian directions d^ν, so that

$$w^{\nu+1} = w^\nu + \varrho_\nu d^\nu + (1 - \varrho_\nu)\gamma R_\gamma(w^\nu), \tag{9}$$

where ϱ_ν is a weight parameter, and

$$d^\nu = -H_\nu R_\gamma(w^\nu). \tag{10}$$

H_ν is an invertible linear operator, which encodes first-order information about R_γ and satisfies the *invert secant condition* $s^\nu = H_{\nu+1}y^\nu$, with $s^\nu = w^{\nu+1} - w^\nu$ and $y^\nu = R_\gamma(w^{\nu+1}) - R_\gamma(w^\nu)$. H_ν is calculated by means of the L-BFGS method.
The average parameter ϱ_ν is chosen to be the largest number in $\{1/2^i \; : \; i \in \mathbb{N}\}$ such that

$$\varphi_\gamma(w^{\nu+1}) \leq \varphi_\gamma(w^\nu) - \sigma||R_\gamma(w^\nu)||^2. \tag{11}$$

In case the Lipschitz constant L_ϕ is not known in advance, it can be evaluated with a backtracking procedure. The interested reader is referred to Stella et al. (2017) for more information on this procedure.

Algorithm 1 PANOC Algorithm

```
Input: Initial guess $\overline{w}^0 \in \mathbb{R}^{n_w}$, Tolerance $\epsilon > 0$, Max. iterations $\nu_{max}$, Estimate $L_\phi > 0$, Parameters $\gamma \in (0, 1/L_\phi)$, $\sigma \in (0, \frac{\gamma}{2}(1 - \gamma L_\phi))$
Output: Approximate solution $\overline{w}^*$
  for $\nu = 0, 1, ..., \nu_{max}$ do
    Evaluate $T_\gamma(\overline{w}^\nu)$ as in (5)
    Update $R_\gamma(\overline{w}^\nu)$ as in (6)
    if (7) is satisfied, exit
    Get $d^\nu$ from (10)
    Get $\varrho_\nu$ satisfying (11)
    Update solution candidate $\overline{w}^{v+1}$ as in (9)
```

2.5 Augmented Lagrangian Method

Since PANOC is only suitable for minimizing bound-constrained optimization problems (4), more general optimization problems as (3) must be cast into the form of (4). This cast procedure is done by using the *Augmented Lagrangian method* (ALM). ALM is able to solve constrained optimization problems by performing an *outer iterative procedure* which depends on the solution of an *inner optimization problem.* ALM is shown in Algorithm 2. The *inner optimization problem* casts problem (3) into problem

$$\min_{w \in W} \quad \phi(w, p, c, \lambda), \tag{12}$$

where $\lambda \in \mathbb{R}^{n_\lambda}$ is the vector of Lagrange multipliers, $c > 0$ is a penalty parameter, and

$$\phi(w, p, c, \lambda) = f(w, p) + \frac{c}{2}\mathbf{dist}_C^2(g(w, p) + c^{-1}\lambda). \tag{13}$$

Problem (12) can be solved by PANOC.
The *outer iterative procedure* updates λ, c and the *inner problem* tolerance ϵ. The update of the Lagrange multipliers λ is done in two steps. At every outer iteration ϑ, λ^ϑ is projected into a compact set $\Lambda \subseteq C^*$ as

$$\lambda^\vartheta = \mathbf{\Pi}_\Lambda(\lambda^\vartheta), \tag{14}$$

and after getting an updated *inner optimization problem* solution w^ϑ from PANOC, $\lambda^{\vartheta+1}$ is calculated according to

$$\lambda^{\nu+1} = \lambda^\vartheta + c(g(w^\vartheta) - \mathbf{\Pi}_C(g(w^\vartheta) + c^{-1}\lambda^\vartheta)) \tag{15}$$

The termination criteria of ALM algorithm depends on the λ-update, so that the condition

$$||\lambda^{\vartheta+1} - \lambda^\vartheta|| := z^{\vartheta+1} \leq c\delta, \tag{16}$$

where δ is a user-defined tolerance, must be satisfied.
The penalty parameter c is updated only if there has not been a sufficient decrease in z, defined by the used-defined parameter θ, so that

$$z^{\vartheta+1} \leq \theta z^\vartheta. \tag{17}$$

Then c is updated by a factor $\rho > 1$ so that

$$c^{\vartheta+1} = \rho c^\vartheta. \tag{18}$$

Inner problem tolerance ϵ is updated by a factor $\beta \in (0,\ 1)$ and lower bounded by the user-defined parameter ϵ_{min}, as

$$\epsilon^{\vartheta+1} = max\{\epsilon_{min}, \beta\epsilon^\vartheta\}. \tag{19}$$

3. METHODOLODY

This section describes the overall structure and workflow of the toolchain. First part presents the motivation and overview of the toolchain. Next, details on matters of every tool composing the toolchain are given in the corresponding subsections.

Algorithm 2 ALM Algorithm

```
Input: Parameters $p \in \mathbb{R}^{n_p}$, $\rho > 1$, $\beta \in (0, 1)$, $\theta \in (0, 1)$, Initial guess $\overline{w}^0 \in \mathbb{R}^{n_w}$ and $\overline{\lambda}^0 \in \mathbb{R}^{n_\lambda}$, Penalty $c^0 > 0$, Tolerances $(\epsilon^0, \epsilon_{min}, \delta) > 0$, Max. outer iterations $\vartheta_{max}$.
Output: Approximate solution $(\overline{w}^*, \overline{\lambda}^*)$
  for $\vartheta = 0, 1, ..., \vartheta_{max}$ do
    Project $\overline{\lambda}^\vartheta$ as in (14)
    Get $\overline{w}^\vartheta$ by solving (12) with Algorithm 1
    Get $\overline{\lambda}^{\vartheta+1}$ from (15)
    if (16) is satisfied, exit
    elseif (17) is not satisfied, update $c^\vartheta$ as in (18)
    Get $\epsilon^{\vartheta+1}$ from (19)
```

3.1 Toolchain Overview

This toolchain aims to determine a direct workflow from problem description to deployment of a solution in the framework of nonlinear MPC for serial robots. To the best of the authors knowledge, there is currently no open tool-chain which allows a user to easily deploy nonlinear MPC controllers on serial robots, without having to explicitly program most of the solution. This toolchain is composed by several modules based on open source software. These modules are CasADi, Pinocchio, Optimization Engine, and Orocos, and are described in the following subsections. These modules are currently supported by most Linux distributions.
The user-inputs required by the toolchain are

(1) Problem-definition script: the user must input the optimization problem (2) in which the MPC controller is based, as well as the definition of the decision variables vector w and set W. This is done in a Python script by setting CasADi functions.
(2) Unified robot description format (URDF) file: it is an XML-based, human-readable file that describes geometric and parametric features of robots. These features include, among others, mass, inertia, relative position and orientation of every link, limits and type of joints, and collision geometry. This file is commonly delivered by robot vendors or can be created by the user.

The structure of the toolchain is shown in Fig. 1.

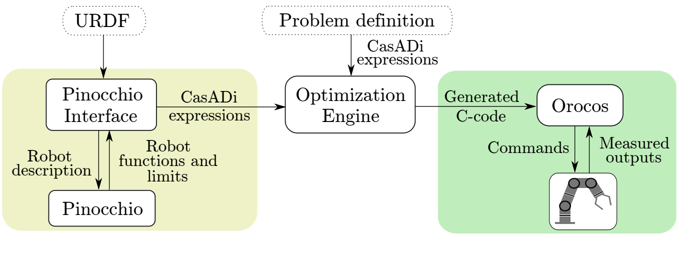

Fig. 1. Overview of the open toolchain for nonlinear MPC for serial robots.

3.2 CasADi

CasADi is a software framework for algorithmic differentiation (AD) and numerical optimization built in C++, with Python and MATLAB bindings. The core feature of CasADi is to perform AD on *expression* graphs, see Andersson et al. (2019). CasADi can import and export these *expressions* and generate self-contained C code for their efficient evaluation. CasADi allows for *expressions* handling, differentiation, and code-generation in the whole toolchain, as described later in this section.

3.3 Pinocchio

The toolchain relies on Pinocchio (Carpentier et al. (2019)), for generating computationally-efficient dynamics and kinematics functions for a serial robot described by a URDF file. Pinocchio is a rigid-body-dynamics library (RBDL) written in C++ and interfaced from Python. It computes *forward dynamics* functions by using the *articulated body algorithm*, while *forward kinematics* are computed based on coordinate transformations obtained from the spatial placements of each robot link. The computational efficiency of these functions relies on the sparsity handling with specific spatial operators for each joint type, and the use of static polymorphism, according to Carpentier et al. (2019).
We built an interface between CasADi and Pinocchio that converts functions generated by Pinocchio into CasADi *expressions*, which can then be exported to be used by the following tool in the chain. In addition to these *expressions*, the interface passes *robot limits*, retrieved from the URDF, to the following tool. These *limits* may include input limits $\tau_{min}, \tau_{max} \in \mathbb{R}^{n_{\mathrm{dof}}}$ and joint limits $q_{min}, q_{max}, \dot{q}_{min}, \dot{q}_{max} \in \mathbb{R}^{n_{\mathrm{dof}}}$, and are needed to define W.

3.4 Optimization Engine

Optimization Engine is a nonconvex optimization framework. This framework is implemented in Rust and can be interfaced from Python and MATLAB, see Sopasakis et al. (2020). It relies on Algorithm (1) and Algorithm (2) for fast solution of nonconvex optimization problems with low computation and memory requirements. The workflow of Optimization Engine inside the toolchain is defined as follows. First, problem-definition is set by the user by using CasADi *expressions*, while robot *expressions* and *limits* are retrieved from Pinocchio. Next, it uses CasADi to cast the optimization problem into the form of (3). To prepare the execution of Algorithm (2), CasADi constructs and code-generate *expressions* for $g(w, p)$, $\phi(w, p, c, \lambda)$, and $\nabla\phi(w, p, c, \lambda)$. Afterwards, Optimization Engine code-generates and compiles the solution of the problem in Rust with C bindings.

3.5 Open robot control software (Orocos)

The open robot control software (Orocos) is a robotics control framework satisfying real-time execution constraints, introduced in Bruyninckx (2001). Its real-time toolkit (RTT) allows users to deploy easily configurable, component-based control applications while satisfying hard timing constraints. In the toolchain, Orocos receives the code-generated solution of the problem from Optimization Engine and executes it inside a *control* component, thus computing the control input u^0 to be applied to the robot. An *interface* component, capable of simulating robot dynamics and interfacing the driver of a real robot, sends the control input to the robot and reads measured data from it. The measured data is then sent to the *control* component, which updates parameter p and solves the optimization problem again.

3.6 Parallelization

The use of multiple shooting as method to set dynamics constraints (2c) along N subintervals $t_k \in T_{ms}$, enables the possibility to evaluate the resulting N decoupled constraints in parallel, see Leineweber et al. (2003). A decoupled objective function (2a) and path constraints (2d) can also be evaluated in a parallel manner. CasADi implements a *map* function, which instructs the parallel evaluation of a *expression* by using the *open multi-processing* (OpenMP) API. The *map* function is used in the problem-definition script. Since forward dynamics of serial robots are computationally expensive *expressions*, parallelization of multiple shooting constraints represents a major speed-up in the computation of nonlinear MPC for such robots. This is demonstrated by means of a numerical example shown in next section.

4. NUMERICAL EXAMPLE

In this section, we present a numerical example built by using the toolchain presented in Section 3 and compare its results with other state-of-the-art tools. This example was executed on a desktop PC with Intel Core i9-9900X processor, with up to 20 threads, and 16 GB RAM.

4.1 Problem Definition

We consider a 7-degrees-of-freedom *Kinova Gen3* serial robot performing a common task such as path following. Path-following MPC aims to steer a robot from an initial end-effector pose, described by $\{\mathbf{p}_{ee}(q^0), \mathcal{R}_{ee}(q^0)\}$, along a path described by a reference position vector $\mathbf{p}_{ref}(s) \in \mathbb{R}^{3\times1}$, reference rotation matrix $\mathcal{R}_{ref}(s) \in \mathbb{R}^{3\times3}$, and reference path velocity $\dot{s}_{ref}(s) \geq 0$, where $s \in [0, 1]$ is the path progress parameter. $\mathbf{p}_{ref}(s)$, $\mathcal{R}_{ref}(s)$, and $\dot{s}_{ref}(s)$ are known a priori.
In order to introduce path following into the MPC formulation, we follow the approach described in Van Duijkeren (2019). This approach introduces path-related state vector $\chi = [s, \dot{s}]^T \in \mathbb{R}^2$, path-related input $\mu = \ddot{s} \in \mathbb{R}$, and path dynamics $\xi_{path}(\chi, \mu, t)$ corresponding to a double integrator. Recall that $x = [q^T, \dot{q}^T]^T$ and $u = \tau$. We consider augmented dynamics $\dot{\hat{x}} = \hat{\xi}(\hat{x}, \hat{u}, t)$, where $\hat{x} = \begin{bmatrix} x^T & \chi^T \end{bmatrix}^T \in \mathbb{R}^{n_{\hat{x}}}$, $\hat{u} = \begin{bmatrix} u^T & \mu \end{bmatrix}^T \in \mathbb{R}^{n_{\hat{u}}}$, and $\hat{\xi}(\hat{x}, \hat{u}, t) = \begin{bmatrix} \xi(x, u, t)^T & \xi_{path}(\chi, \mu, t)^T \end{bmatrix}^T$. The augmented dynamics are discretized as

$$\hat{x}_{k+1} = \hat{\xi}_d(\hat{x}_k, \hat{u}_k), \tag{20}$$

featuring the multiple shooting method with a sample time $\delta_t = 2.5\ ms$. The path progression error is defined as $e_{\dot{s}} = \dot{s} - \dot{s}_{ref}$, while path constraints are defined by $e_{pos} = \mathbf{p}_{ee} - \mathbf{p}_{ref}$, and $e_{rot} = (\mathcal{R}^{(n)}_{ref} \times \mathcal{R}^{(n)}_{ee} + \mathcal{R}^{(s)}_{ref} \times \mathcal{R}^{(s)}_{ee} + \mathcal{R}^{(a)}_{ref} \times \mathcal{R}^{(a)}_{ee})/2$, where a rotation matrix is divided in three column vectors as $\mathcal{R}_j = \begin{bmatrix} R^{(n)}_j & R^{(s)}_j & R^{(a)}_j \end{bmatrix}$. These constraints are relaxed allowing the end-effector to deviate from the path reference so that

$$\begin{bmatrix} ||e_{pos}(\hat{x}_k)||^2 \\ ||e_{rot}(\hat{x}_k)||^2 \end{bmatrix} \leq \begin{bmatrix} \rho_{pos}^2 \\ \rho_{rot}^2 \end{bmatrix} + l_k, \tag{21}$$

where ρ_{pos}, ρ_{rot} are maximum deviations allowed for position and rotation errors respectively, and $l_k \in \mathbb{R}^2$ is a slack variable that further relaxes the path constraints.

The following optimization problem is considered for the rest of the section

$$\min_{w} \left[\left\| \begin{bmatrix} \dot{q}_N \\ e_{pos}(\hat{x}_N) \\ e_{rot}(\hat{x}_N) \\ s_N - 1 \\ \dot{s}_N \end{bmatrix} \right\|^2_{\mathbf{Q_N}} + \sum_{k=1}^{N-1} \left\| \begin{bmatrix} e_{\dot{s}}(\hat{x}_k) \\ e_{pos}(\hat{x}_k) \\ e_{rot}(\hat{x}_k) \\ \hat{x}_k \\ \hat{u}_k \end{bmatrix} \right\|^2_{\mathbf{Q_m}} + \mathbf{Q}_l l_k \right] \tag{22a}$$

$$\begin{aligned} \text{s.t.} \quad & \hat{x}_0 - p = \mathbf{0}_{n_x}, && (22b) \\ & (20), && (22c) \\ & (21), && (22d) \\ & w \in W, \quad k \in [0, N-1], && (22e) \end{aligned}$$

where $\mathbf{Q}_N = diag(10^{-5} * \mathbf{1}_{7\times1}, \mathbf{1}_{6\times1}, 10^{-5}, 10)$, $\mathbf{Q}_m = diag(10, 0.1 * \mathbf{1}_{6\times1}, 10^{-5} * \mathbf{1}_{24\times1})$, and $\mathbf{Q}_l = diag(10, 10)$ are diagonal weighting matrices, $N = 16$, $n_{\text{dof}} = 7$, $n_{\hat{x}} = 2n_{\text{dof}} + 2$, $n_{\hat{u}} = n_{\text{dof}} + 1$,

$$w = \begin{bmatrix} \hat{x}_0^T \ \hat{u}_0^T \ l_0^T \cdots \hat{u}_{N-1}^T \ l_{N-1}^T \ \hat{x}_N^T \end{bmatrix}^T \in \mathbb{R}^{n_w}, \tag{23}$$

$n_w = n_{\hat{x}}N + n_{\hat{u}}(N-1) + 2(N-1)$, and W is defined by considering $q_{min} \leq q_k \leq q_{max}$, $\dot{q}_{min} \leq \dot{q}_k \leq \dot{q}_{max}$, $0 \leq s_k \leq 1$, $\dot{s} \geq 0$, $\tau_{min} \leq \tau_k \leq \tau_{max}$, and $l_k \geq \mathbf{0}_{2\times1}$. Constraints (22c) are defined to be evaluated in parallel, in up to $N_c = 16$ cores. Problem (22) is set as input of the toolchain in a Python problem-definition script [1] described in Section 3.1.

4.2 Robot dynamics and kinematics

CasADi *expressions* for forward dynamics and forward kinematics, as well as *limits* of the robot, are obtained by using Pinocchio's interface as shown in Section 3.3. This tool receives the *Kinova Gen3* URDF file [2] as input. The resulting *expressions* are code-generated in C, and compared in terms of evaluation time with those generated with other RBDLs such as Robotran (Docquier et al. (2013)), Spatial_V2 (Featherstone (2012)), and Robotics System Toolbox (RST) (Mathworks (2016)).

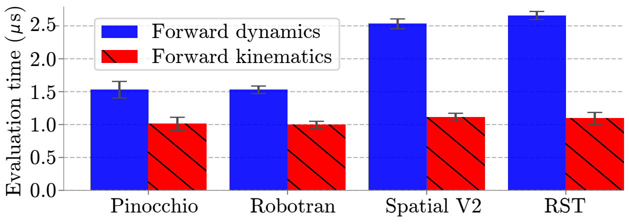

Fig. 2. Comparison of the time needed to evaluate dynamics and kinematics CasADi *expressions* generated with four RBDL.

[1] Available on https://git.io/JeoJT
[2] Available on https://github.com/Kinovarobotics/ros_kortex

As shown in Fig. 2, the evaluation time of the resulting dynamics *expressions* from Pinocchio's interface are among the lowest, only comparable with that from Robotran. However, Robotran requires the user to manually build a model of the robot, which is a more complex process than the one using a URDF file. Evaluation times of kinematics *expressions*, on the other side, are similar for every tested RBDL.

4.3 Solution generation and deployment

Once problem-definition and robot *expressions* are set, a solution is code-generated with Optimization Engine, as described in Section 3.4. The generated C-code is then executed by the Orocos *control* component, while the *interface* component simulates the dynamics of the robot. The actual path followed by the end-effector of the *Kinova Gen3* robot during the simulated test is shown in Fig. 3.

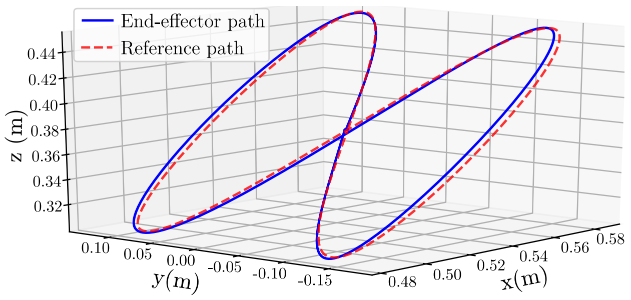

Fig. 3. Result of the path followed by the end-effector compared with the reference path.

Fig. 4 shows the evolution of error norms $\{|e_{\dot{s}}|, ||e_{pos}||, ||e_{rot}||\}$ along the path parameter $s \in [0, 1]$. Position and orientation of the end-effector are allowed to deviate from the reference up to $\rho_{pos} = \rho_{rot} = 0.01$ while satisfying constraint (22d). Also, since $|e_{\dot{s}}|$ is heavily penalized in the objective (22a), its value is kept low.

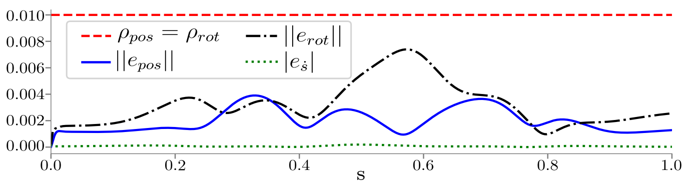

Fig. 4. Evolution of position, rotation, and path progression error norms.

We also compare the time needed to solve problem (22) in one time-step, including function evaluations, by using the presented toolchain (with no parallelization of constraint (22c), $N_c = 1$), with the one using a common approach such as the *sequential quadratic programming* (SQP) method with the *real-time iteration* (RTI) scheme. Also, the SQP approach is solved with three state-of-the-art *quadratic program* (QP) solvers: QRQP (Andersson and Rawlings (2018)), OSQP (Stellato et al. (2018)) and HPIPM (Verschueren et al. (2019)). It is worth mentioning that with our toolchain, problem (22) is completely solved up to the defined tolerance $\epsilon = 10^{-4}$, while the SQP method with RTI returns a suboptimal solution after truncating its algorithm when solving just one QP subproblem. This comparison is shown in Table 1.

Table 1. Average solution time of problem (22) with $N_c = 1$: measured and estimated(*)

Solver	**Solution time**	Function evaluation time	QP solver time
Toolchain	4.613 ms	-	-
SQP-QRQP	4.692 ms	2.133 ms	2.559 ms
SQP-OSQP	4.720 ms	2.133 ms*	2.587 ms
SQP-HPIPM	6.376 ms	2.133 ms*	4.243 ms

As shown in Table 1, solution time of problem (22) with this toolchain is comparable to that by using the fastest solver in SQP method, when using one core.
Now, in Fig. 5, we evaluate the toolchain and SQP-QRQP in terms of solution time of problem (22), while parallelizing (22a), (22c) and (22d) in up to $N_c = 16$ cores.

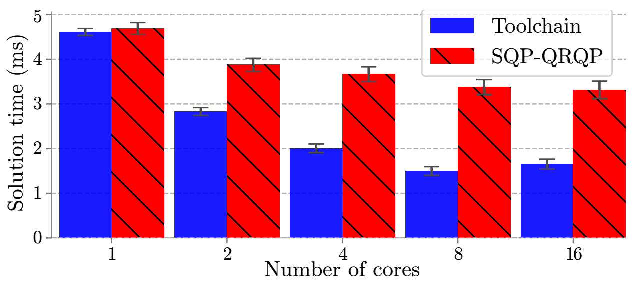

Fig. 5. Comparison of the evaluation time needed to solve problem (22) with the proposed toolchain and SQP-QRQP.

The solution time achieved with the presented toolchain is lower than the one achieved using SQP-QRQP for all tests using $N_c \geq 2$ cores to parallelize the evaluation of multiple shooting constraints. The greatest difference between the toolchain and SQP-QRQP in terms of solution time is obtained when using $N_c = 8$ cores, where the solution with the toolchain is 2.25 times faster than that with SQP-QRQP. Also, the greatest speed-up due to parallelization is achieved with $N_c = 8$ cores, having a speed-up of 3.07x with respect to implementation with $N_c = 1$ core. The results obtained while using $N_c = 16$ cores do not represent a speed-up regarding those using $N_c = 8$ cores due to overhead in multicore communication with OpenMP.
Two remarks are made regarding these results. Firstly, the path velocity reference $\dot{s}_{ref}$ defined for this test is small in magnitude since solution of problem (22) is not converging within ϑ_{max} iterations for $\dot{s}_{ref} > 0.2$, while it does converge to a suboptimal solution by using the SQP method for $\dot{s}_{ref} > 0.2$. Secondly, the initial guess $\{w^0, \lambda^0\}$ passed to both Algorithm (2) and SQP method, is computed a priori by solving problem (22) for a previous time-step and parameter p, with the interior point method solver IPOPT.

5. CONCLUSION

In this paper we presented an open toolchain that has several advantages to existing MPC and optimization frameworks regarding its use on nonlinear MPC for serial robots. The modules composing the toolchain were described and their objective within the toolchain was detailed. This toolchain is able to generate and deploy C code for nonlinear MPC solution based on a URDF file describing the robot, and a problem-definition script where the user sets the objective and constraints that the solution must consider. It also showed that evaluating multiple shooting constraints in parallel leads to a practical speed-up of up to 3.07x in terms of evaluation time of the solution.
Future work will aim to develop a core module for the toolchain, which should be able to interface multiple numerical optimization solvers, including Optimization Engine, to widen the scope of the toolchain. We will then validate the toolchain with a real robot application.

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.

Andersson, J.A. and Rawlings, J.B. (2018). Sensitivity Analysis for Nonlinear Programming in CasADi. *IFAC-PapersOnLine*, 51(20), 331–336.

Bock, H. and Plitt, K. (1984). A Multiple Shooting Algorithm for Direct Solution of Optimal Control Problems. *IFAC Proceedings Volumes*, 17(2), 1603–1608.

Bruyninckx, H. (2001). Open robot control software: the OROCOS project. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation*, volume 3, 2523–2528. IEEE.

Carpentier, J., Saurel, G., Buondonno, G., Mirabel, J., Lamiraux, F., Stasse, O., and Mansard, N. (2019). The Pinocchio C++ library : A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives. In *2019 IEEE/SICE International Symposium on System Integration (SII)*, 614–619. IEEE.

Docquier, N., Poncelet, A., and Fisette, P. (2013). ROBOTRAN: A powerful symbolic gnerator of multibody models. *Mechanical Sciences*, 4(1), 199–219.

Englert, T., Völz, A., Mesmer, F., Rhein, S., and Graichen, K. (2019). A software framework for embedded nonlinear model predictive control using a gradient-based augmented Lagrangian approach (GRAMPC). *Optimization and Engineering*, 20(3), 769–809.

Featherstone, R. (2012). Spatial Vector and Rigid-Body Dynamics Software. URL `https://cutt.ly/spatial_v2`.

Janeček, F., Klaučo, M., Kalúz, M., and Kvasnica, M. (2017). OPTIPLAN: A Matlab Toolbox for Model Predictive Control with Obstacle Avoidance. *IFAC-PapersOnLine*, 50(1), 531–536.

Leineweber, D.B., Bauer, I., Bock, H.G., and Schlöder, J.P. (2003). An efficient multiple shooting based reduced SQP strategy for large-scale dynamic process optimization. Part 1: theoretical aspects. *Computers & Chemical Engineering*, 27(2), 157–166.

Mathworks (2016). Robotics System Toolbox. URL `https://cutt.ly/rst-matlab`.

Sopasakis, P., Fresk, E., and Patrinos, P. (2020). OpEn: Code Generation for Embedded Nonconvex Optimization. In *21st IFAC World Congress*. Berlin, Germany.

Stella, L., Themelis, A., Sopasakis, P., and Patrinos, P. (2017). A simple and efficient algorithm for nonlinear model predictive control. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, 1939–1944. IEEE.

Stellato, B., Banjac, G., Goulart, P., Bemporad, A., and Boyd, S. (2018). OSQP: An Operator Splitting Solver for Quadratic Programs. *2018 UKACC 12th International Conference on Control, CONTROL 2018*, 339.

Van Duijkeren, N. (2019). *Online Motion Control in Virtual Corridors - For Fast Robotic Systems*. Ph.D. thesis, KU Leuven.

Verschueren, R., Frison, G., Kouzoupis, D., van Duijkeren, N., Zanelli, A., Novoselnik, B., Frey, J., Albin, T., Quirynen, R., and Diehl, M. (2019). acados: a modular open-source framework for fast embedded optimal control.