Tasho: A Python Toolbox for Rapid Prototyping and Deployment of Optimal Control Problem-Based Complex Robot Motion Skills

Ajay Suresha Sathya[1⋆], Alejandro Astudillo[1⋆], Joris Gillis[1], Wilm Decré[1], Goele Pipeleers[1], and Jan Swevers[1]

***Abstract*— We present Tasho (Task specification for receding horizon control), an open-source Python toolbox that facilitates systematic programming of optimal control problem (OCP)-based robot motion skills. Separation-of-concerns is followed while designing the components of a motion skill, which promotes their modularity and reusability. This allows us to program complex motion tasks by configuring and composing simpler tasks. We provide templates for several basic tasks like point-to-point and end-effector path-following tasks to speed up prototyping. Internally, the task's symbolic expressions are computed using CasADi and the resulting OCP is transcribed using Rockit. A wide and growing range of mature open-source optimization solvers are supported for solving the OCP. Monitor functions can be easily specified and are automatically deployed with the motion skill, so that the generated motion skills can be easily embedded in a larger control architecture involving higher-level discrete controllers. The motion skills thus programmed can be directly deployed on robot platforms using the C-code generation capabilities of CasADi. The toolbox has been validated through several experiments both in simulation and on physical robot systems. The open-source toolbox can be accessed at: https://gitlab.kuleuven.be/meco-software/tasho**

I. Introduction

Optimization-based motion generation frameworks have been highly successful in robotics since many tasks can be effectively programmed through constraints [1], because the control trajectory or inputs for a given robot that achieves the specified task can be automatically computed by solving a constrained optimization problem. This methodology provides a high degree of generalizability to different robot platforms and task parameters. However, programming these tasks is an iterative and time-consuming process that involves modeling, tuning parameters and adding/removing constraints. Furthermore, the workflow from task definition to solution deployment often involves prototyping in a high-level programming language for simulations, to later program in a low-level language for deployment on a real robot.

To address the aforementioned issues, we present Tasho – a Python toolbox for **Ta**sk **s**pecification for receding **ho**rizon control – which (i) provides several features and design patterns to facilitate the iterative design process in constraint-based task specification for robots and (ii) fosters the synergy between relevant open-source modules for robotics and optimization, aiming at defining an efficient and direct workflow for fast deployment of constraint-based tasks on robot hardware.

The authors would like to gratefully acknowledge the support from the Flanders Make SBO project MULTIROB: "Rigorous approach for programming and optimal control of multi-robot systems" and the FWO projects G0A6917N and G0D1119N of the Research Foundation - Flanders (FWO - Flanders).

[1]The authors are with the MECO Research Team of the Department of Mechanical Engineering, KU Leuven and the DMMS-M Core Lab, Flanders Make, 3001 Leuven, Belgium {ajay.sathya, alejandro.astudillovigoya}@kuleuven.be

⋆ The first two authors contributed equally to this work.

A. Related Work

At one end, constraint-based task control can be achieved with an instantaneous and reactive approach, as in eTaSL [2] and Stack of Tasks [3], where a quadratic program (QP), is solved to obtain instantaneous control inputs. However, this approach does not simulate the system over a prediction horizon and provides limited guarantees related to optimality and constraint satisfaction. At the other end, tasks can be solved as a trajectory optimization problem and executed in open loop in a nonreactive manner. Receding horizon control, also known as model predictive control (MPC), combines the best of both approaches, where the trajectory optimization problem is solved repeatedly in a loop in real-time to obtain both reactivity and favourable properties related to optimality and constraint satisfaction [4].

Multiple software frameworks have been developed recently for robot trajectory optimization such as the ADRL Control Toolbox [5], ALTRO [6] and Crocoddyl [7]. They focus on providing fast optimization solvers using algorithms based on Riccati recursion like iterated linear quadratic regulator (iLQR) and differential dynamic programming (DDP), which are necessary for achieving high MPC loop frequencies. Drake [8] is a C++ toolbox providing access to a large number of mature solvers and dynamics simulators for optimization-based control systems design.

In these software frameworks, however, one needs to formulate a new mathematical program from scratch each time a new task is defined. There is limited support for composing tasks to obtain a new task. Except for Drake, they also do not facilitate developing reusable constraint models. For instance, it is desirable to mathematically model the complex constraint that encodes that an object of a particular shape is inside a box and reuse it across different tasks that require such constraints for particular objects and boxes. Even if the other frameworks provide Python bindings, there is no support for a unified and simplified workflow from prototyping in Python to deployment with limited or no additional coding in a low level language like C++.

B. Approach and contributions

Tasho is a Python toolbox that aims to ease the workflow from task definition to controller deployment in robot appli-

cations through various design choices. It leverages existing open source libraries to provide users with easy access to relevant functionalities. Tasho is released under the GNU LGPLv3 license and relies on the powerful optimization and algorithmic differentiation (AD) framework CasADi [9] for expression handling, algorithmic differentiation, serialization, code-generation and interfacing optimization solvers and algorithms. Rockit [10], an open-source package built on top of CasADi by the MECO Research Team, is used within Tasho to transcribe, solve and post-process OCPs arising in MPC implementations. Please not that the use of Tasho does not represent an additional overhead with respect to the direct use of CasADi or Rockit during MPC executions. Moreover, Tasho interfaces the rigid-body dynamics library (RBDL) Pinocchio [11] to generate efficient functions of rigid-body dynamics and kinematics (and their derivatives). In addition to this, support is provided for physics simulation and visualization through PyBullet [12].

It aims to make task specification intuitive, modular and reusable by encouraging users to model constraints separately and define a task from bottom-up as pre-, per- and post-conditions of these constraints, similar to how predicates and actions are defined in the PDDL domains [13]. Flexibility is provided to modify and compose existing tasks both in parallel and sequence to quickly define new and more complex tasks. It aims to eliminate additional and often time-consuming coding in low level languages like C++ for experimental deployment, through code-generation and motion skill modeling which can be parsed by robot control middleware like Orocos [14] and ROS [15] to automatically generate a controller component. Tasho's scope is completely restricted to continuous motion skills. However, it provides support for specifying monitor functions which evaluate boolean functions of continuous expressions and communicate events. This feature is essential for a high-level discrete controller/planner to coordinate change in the behaviour at the continuous level. This choice of separation between computation and coordination respects suggested best practices in robotics [16].

The main contributions of this work are (i) a user-friendly open-source toolbox that provides a unified workflow that streamlines and automates many steps from specification to experimental deployment to enable rapid prototyping of OCP/MPC based motion skills, (ii) the proposal of a design methodology for continuous task specification that enhances modularity and reusability, and (iii) the definition of a functionality to easily program complex tasks by modifying existing task templates, by composing simpler tasks or by reusing specific submodules of other tasks.

II. Structure of Tasho

This section presents the overall structure and modules of Tasho and their dependencies as shown in Fig. 1 and described below.

1) Robot model: Efficient subroutines to compute robot kinematics, forward and inverse dynamics, and analytical derivatives of such dynamics and kinematics directly from

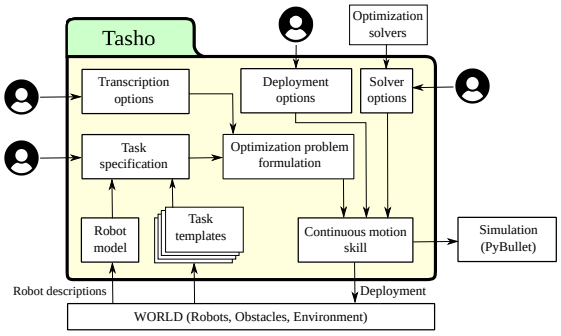

Fig. 1: Overall structure of Tasho showing the interaction of its internal components.

robot description (URDF) files are provided through an interface with the state-of-the-art RBDL Pinocchio.

2) Task specification: The task specification module forms the core of the toolbox and allows for reusable constraint-based tasks models to be specified by the user. This module is described in more detail in Section III.

3) Transcription options: Through the Rockit package, this module provides access to *direct methods* [17] for converting the infinite-dimensional OCP associated with the task to a finite-dimensional approximation that can then be solved as a nonlinear program (NLP). The user can choose amongst multiple-shooting, single-shooting, direct collocation or B-splines.

4) Task templates: This module provides templates for various tasks and constraints. They serve as example for task specifications and can also be reused for specifying more complex tasks. The use of templates is explained further in Section III-E.

5) Optimization problem formulation: This module performs the actual transcription from a constraint-based task specification into an NLP using the aforementioned transcription options.

6) Deployment options: This module sets the options relevant for deployment of the controller by defining messaging ports, properties and monitor functions. It is explained in detail in Section IV.

7) Solver options: This module exposes to the user the interface to several numerical optimization solvers and allows the user to provide relevant solver options, e.g., maximum number of iterations, convergence tolerance, and step-size. Through CasADi, several mature optimization solvers are made available, including IPOPT [18], SNOPT [19], KNITRO [20], and CasADi's implementation of the SQP method, to name a few. Support for ACADOS [21], a popular OCP solution framework for embedded applications is provided through Rockit.

8) Continuous motion skill: The continuous motion skill module generates a controller component that is ready to be deployed either in a simulated environment (through a simple interface to the PyBullet library) or in a real robot (through an interface provided with Orocos). It handles the deployment of both the OCP-based robot controller and the

monitor functions. This module is presented in more detail in Section V.

III. Task Specification Module

The core module of Tasho, i.e., the task specification module, is detailed in this section. The first subsection presents all elements within a task component. We then give details on the task composition capabilities of Tasho, on the flexibility to remove or substitute entities within a task, and on the task templates that are provided.

A. Elements of a Task Specification Component

A task component contains all the information relevant to a continuous task whose schematic diagram is shown in Fig. 2. It consists of variables, expressions, constraint expressions and constraint instances, and provides several functions to manipulate these objects.

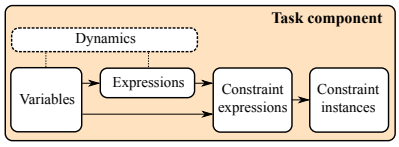

Fig. 2: Structure of a task specification component.

In Tasho, a task specification component is an instantiation of the `Task` class which composes the task specification module and requires two arguments: (i) a `name`, the name given to the task by the user, and (ii) a `mid` (model id) specifying a type for the task. These string arguments are required for all entities in a task component and their concatenation should be a unique identifier for every entity within a task. We explain different elements of a task component through code-snippets taken from a point-to-point (P2P) motion task specification, where a specific link on a robot needs to reach a goal pose specified in Cartesian-space. This task is first created as follows:

```
task = Task(name = rob.name, mid = 'P2P')
```

1) Variables: Variables are the atomic entities of a task. All the expressions of a task are functions of these variables. The variables themselves can be of five types: states, controls, parameters, free variables and magic numbers.

States are variables with *dynamics* and evolve over time based on a continuous-time or discrete-time *dynamics* function which must be provided for every state variable as either another variable or an expression, e.g., the generalized joint positions and velocities of a robot.

Controls are variables that also evolve over the time horizon of the task, but this evolution is not constrained by a higher-order derivative unlike the state variable, e.g., the actuated joint torques of a robot.

Parameters are placeholder expressions whose numerical values must be provided by the user everytime they solve the optimization problem associated with the task. They can be used to supply state information from perception modules, e.g., current joint pose of the robot or the pose of an object that needs to be picked up. Parameters can be either fixed or also varying over the horizon.

Free variables are other decision variables of the optimization problem that do not vary over time. For instance, the base position of a manipulator can be left as a free variable for the pick-and-place tasks to also compute an associated optimal base position.

Magic numbers are variables initialized to important constants like controller gains or tolerances. Explicitly declaring magic numbers instead of hard-coding constants allows programmers to access, inspect and modify them in an existing task component for reconfiguration. Unlike parameters, magic numbers cannot be modified after code-generation.

We present a code-snippet that shows the declaration of three variables `q`, `qd`, and `qdd` within the P2P task. Along with `name` and `mid`, the declaration takes as arguments (i) the `type` and (ii) a `shape` which sets the two-dimensional size of the variable.

```
q   = task.create_variable(rob.name,
       'q',   type = 'state', shape = (rob.nd,1))
qd  = task.create_variable(rob.name,
      'qd',  type = 'state', shape = (rob.nd,1))
qdd = task.create_variable(rob.name,
      'qdd', type = 'control', shape = (rob.nd,1))
```

In this example, the variables `q` $:= q \in \mathbb{R}^{n_{\text{dof}}}$ and `qd` $:= \dot{q} \in \mathbb{R}^{n_{\text{dof}}}$ are states representing the generalized joint angles and velocities, respectively, `qdd` $:= \ddot{q} \in \mathbb{R}^{n_{\text{dof}}}$ is defined as a control input and represents the generalized joint accelerations, and n_{dof} is the number of dof of the robot ($n_{\text{dof}} = 7$). The double integrator *dynamics* involving q, $\dot{q}$ and $\ddot{q}$ can be defined within the task as

```
task.set_der(q, qd)
task.set_der(qd, qdd)
```

where $\dot{q}$ is set as time-derivative of q, and $\ddot{q}$ is set as time-derivative of $\dot{q}$.

2) Expressions: Expressions are the result of applying mathematical functions on variables and/or other expressions. Since CasADi is used internally to represent these expressions, we support the same wide range of mathematical functions available in CasADi for their definition. In the following example, two expressions are defined: `pose`, which represents the pose of the end-effector computed by the forward kinematics function $\mathbf{fk}(q) : \mathbb{R}^{n_{\text{dof}}} \rightarrow SE(3)$, and `trans_error`, which is the substraction between the position of the end-effector $p_{\text{ee}}(q)$ (the first three components of the last column of `pose`) and a goal position p_{goal} (the first three components of the last column of a variable `goal_pose_franka_panda` $\in SE(3)$ of type magic number).

```
pose = Expression(
  rob.name, mid = 'pose_7',
  expr_fun = lambda q: fk(q), q)
trans_error = Expression(
  rob.name, mid = 'trans_error_pose_7_vs_goal',
  expr_fun = lambda e, r: -e[0:3,3] + r[0:3,3],
  pose_7, goal_pose_franka_panda)
```

Besides the required arguments `name` and `mid`, an expression requires an argument `expr_fun`, which is a Python lambda function defining the actual expression, and a set of parent variables (or expressions).

3) Constraint expressions: Constraint expressions encode the mathematical program constraints on expressions and variables and implicitly defines their feasible set. The constraints can be equality, inequality or double-sided inequality constraints. We allow two levels of priority (or hardness) – 'hard' and 'soft'. For soft constraints, slack variables to permit constraint violation are automatically created and the user must specify a penalty function (ℓ_1, ℓ_∞ and ℓ_2^2 norms are currently supported) and a multiplicative penalty weight. Specifying more than two levels of strict priority between constraints of an OCP is still a research topic [22] and is not currently supported in Tasho. We also make the design decision choice that any components of the objective function of a task must be specified as soft constraints.

The following code snippet is used to build a constraint expression based on expression `trans_error`.

```
trans_con = ConstraintExpression(
  rob.name, mid = 'trans_con_pose_7_vs_goal',
  expression = trans_error,
  constraint_hardness = 'hard',
  reference = [0, 0, 0])
```

Here, a hard constraint is specified on `trans_error` in order to impose the constraint $p_{\text{ee}}(q) - p_{\text{goal}} = \mathbf{0}$, without specifying the time instant at which the constraint must be met.

4) Constraint Instances: Constraint instances are the imposition of a constraint expression as either initial, path or terminal constraints. As the name suggests, initial constraints must be satisfied at the initial time instant of the predictive horizon, path constraints throughout the horizon and terminal constraints at the final time instant of the horizon.

Continuing with the example, the constraint expression `trans_con` is instantiated at the final time instant of the predictive horizon by using the following code.

```
task.add_terminal_constraints(trans_con)
```

B. Composing Tasks

The `Task` class provides a `compose()` function that combines two subtasks `task1` and `task2` to create a new task `task_new = compose(task1, task2)`. The constraint-based task specification approach substantially simplifies this composition step because the intersection of the feasible set associated with a set of constraints is implicitly the feasible set of the concatenation of all constraints in the set.

Task composition performs a union operation of all the internal components of the individual subtasks. This requires detecting duplicate (if any present) variables, expressions, constraint expressions and constraint instantiates. Otherwise, the task can have multiple variable or expressions representing same physical quantities that may be assigned different values by the optimization solver which is not physically feasible. Having duplicate constraints will affect the efficiency of the optimization problem.

To detect duplicates, we check the unique identifier (defined by `name` and `mid`) of each entity in the task component. For instance, if there is a `Variable` identified as `q` in both `task1` and `task2`, both variables are considered to be duplicates. Only the `q` from `task1` is included in `task_new` and all the children expressions of `q` in `task2` are assigned to the new variable `q` in `task_new`. A similar approach is followed for expressions, constraint expressions and constraint instances. Therefore, we require users to assign unique `name` and `mid` to entities to prevent name-clashes.

C. Sequential composition

Tasho supports sequential composition of tasks and converts it into a multi-stage OCP using Rockit, where the constraints, dynamics and even variables can be different in different stages. Continuity constraints are automatically imposed on state variables that are common between the tasks that are sequentially composed. This is useful in many cases, e.g., in a bin picking task where the first phase involves reaching the object and a second phase involves a Cartesian motion such that the object is between the grippers, or for legged robots where certain constraints are valid only during some gait phases.

D. Removing or substituting entities

Since Tasho aims to promote reusability of tasks and their components, it provides options to modify an existing task to suit a new task. Users can add or remove any entity (variable, expression, constraint expression or constraint instance) to a task. When removing an entity, all its children are also deleted from the task. Conversely, during an entity substitution, the new entity also substitutes the old entity as parent of its children within the task.

E. Task Templates

Various task templates are provided that act as a base and also as examples for constrained-based task specification in Tasho. They can be used as-is or be reused to build other tasks, e.g., composing other tasks or being extended with additional entities.

The P2P task template specification partly shown through the code snippets in this section has additional constraints, e.g., orientation constraints, box limits on `q`, `qd` and `qdd`. This template takes a robot object and a link as arguments and returns a P2P task object. The source code of the specification is available in Tasho's repository[1]. An autogenerated task component graph of a P2P task for the seventh link of a Franka Panda robot is shown in Fig. 3.

The tasks thus specified are transcribed into an NLP after the transcription options are specified. The solver object of this NLP is then either code-generated or serialized using CasADi depending on the solver options.

IV. Deployment Options

The solver object from the previous section can be called in any existing control framework or middleware like ROS written in C/C++ for deployment. But since our aim with

[1]The source code of the P2P template can be accessed at `https://cutt.ly/Tasho_P2P_Template`. Other templates are available at `https://cutt.ly/Tasho_Templates`

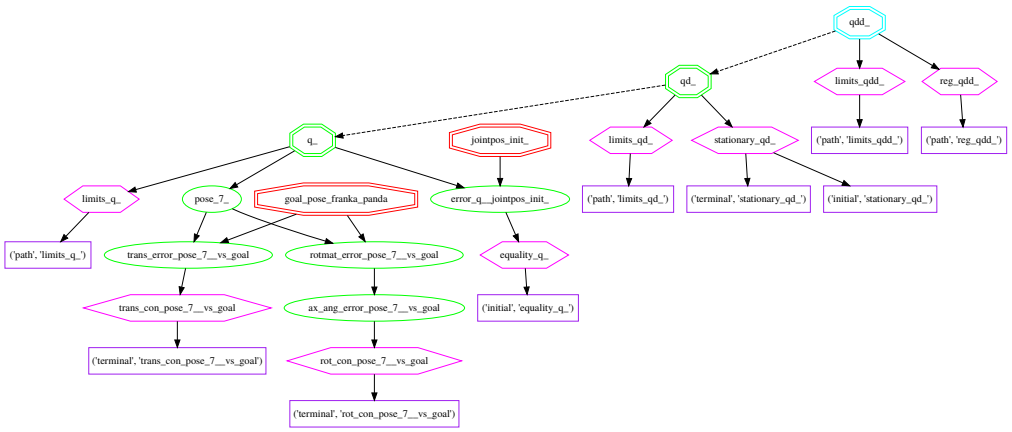

Fig. 3: Task component graph for a point-to-point motion task. In this graph, the octagons represent variables, whose type depends on the color: state - green, control - cyan, magic number - red, parameter - blue (not shown), free variable - yellow (not shown). The dashed directed line represents the system dynamics pointing from the derivative to the state. All the expressions are shown within green ellipses, constraint expressions in magenta hexagons and constraint instances in purple boxes. All entities are labeled by their `mid` in the graph.

Tasho is to program tasks in Python at a level of detail high enough to minimize or eliminate additional coding in a low-level language like C++, we support additional features to make this possible. For this, the user must specify additional deployment options which are explained below.

A. Properties and ports

The controller needs to communicate with other programs, e.g., for configuration, to receive feedback from estimators or to apply control actions. To model this, we specify two types of communication channels similar to Orocos properties and ports which are explained further below.

1) Properties: Properties are meant for communicating information to the controller that does not change during the motion execution and is primarily for configuration purposes, e.g., joint acceleration limits, soft-constraint weights, and goal pose for P2P motions.

2) Ports: Input and output ports are meant for high-frequency communication. These can be used to communicate at the frequency of the controller or even at higher frequencies for monitor functions.

One must declare an input port or a property for every `Variable` of type `parameter` in the task model. One can declare an output port for any `Variable` or `Expression` in the task model. For `Variable` entities of type `free variable`, `state` or `control`, an input port can be declared, which will provide the initial guess for these variables. Such initial guess may be critical for the solver convergence.

B. Monitor functions

Monitor functions register the changes in the truth-value of a boolean expression based on `Variables` and `Expressions`. They register events during task execution, e.g., collisions, constraint violation, solver infeasibility, completion of task, which may be relevant for changing the robot behaviour. In Tasho, we also introduce the option of specifying *predictive monitors*, which monitor not only the current value of an expression, but also its predicted value along the horizon. This implementation of monitor functions represents a bridge that can enable a rich interaction between a continuous-level controller and a discrete high-level controller since such high-level controller can (i) listen to these monitors to coordinate different tasks and (ii) react in anticipation to events that are predicted to occur in the future. Monitors can be of four types: `Current` (they monitor the state of the system at the current time instant), `Anytime` (they turn true if the monitored expression is true at any step in the horizon), `Throughout` (they are true only if the expression is true at all steps of the horizon) and `Final` (they monitor the expressions at the final instant of the horizon). This is only possible for predictive MPC controllers as opposed to instantaneous controllers. All the options supplied here are saved in a json file to be parsed by the continuous motion skill component described in the next section.

V. Continuous Motion Skill

This section presents the continuous motion skill module of Tasho. This module is in charge of taking the generated solver object and the deployment options to automatically create an executable robot controller ready to be deployed either in simulation or on a real robot. It aims to automate several recurring aspects of the deployment of robot skills that are time consuming when written manually for each new task.

A. Simulation

Being able to simulate and visualize the robot controller in action is extremely useful for prototyping a controller. For this, we provide an interface to the PyBullet library.

B. Deployment on a Real Robot

This component further simplifies the experimental deployment of an OCP/MPC controller for complex robot motion skills. At the time of writing, it implements an interface

with Orocos in C++. We plan to build a similar interface for ROS as well in the future. We present below the deployment component for an MPC controller. A similar component is also implemented for the simpler OCP controller, where the computed trajectory is followed without reactivity.

The deployment of the continuous motion skill module is implemented as a life cycle state machine with five states: initialization, configuration, update, stop and cleanup. The steps involved in each of these states are discussed below. We refer the program that calls the motion skill hereafter as *client*.

1) Initialization: The motion skill component in Orocos enters this state upon creation. In this state, it performs the following actions in order: loads the specified json options file, loads the OCP solver object, allocates working memory, create ports and properties for future communication. After these steps, the skill is ready to be configured by the *client* (possibly a discrete controller).

The client must assign values to properties and connect the ports of the skill to ports of other programs. For instance, the joint velocity output port of the motion skill should be connected to the joint velocity input port of the Orocos robot driver (assuming that the robot is joint velocity controlled). The client can trigger the transition to the configuration state after connecting the ports.

2) Configuration: In the configuration state, the skill (i) reads all properties and messages from the input ports, (ii) sets appropriate numerical values and initial guesses from messages to the corresponding OCP parameters and decision variables, respectively, and (iii) solves the OCP. For an MPC solution, it also loads the MPC solver if a different OCP solver is provided for solving MPC problems. This is desirable if one wants to use a slow but robust solver for solving the OCP to convergence and then a fast and warm-started solver for fast MPC execution, e.g., solving one SQP iteration per MPC iteration as in the real time iteration scheme [23]. The skill is now ready to be deployed.

3) Update: This state deploys the MPC controller and runs in a loop at the rate of the MPC frequency. It receives all input messages and assigns them to the right parameters and variables. It predicts the future state of the system by simulating the system by one MPC timestep for the applied control input to account for MPC computation delay. During the first iteration of the loop, the solution from the OCP solved in the configuration state is used. The task parameters are initialized to these states and the MPC problem is solved. The monitor functions are computed. The control actions and monitor function updates are written to the output ports. The MPC solution is shifted by one timestep for all task states and control variables to prepare for warm-starting during the next iteration.

4) Stop: The *client* can trigger the transition of the skill to the stop state from the update state depending on the event messages sent by the skill – e.g., termination criteria reached or solver infeasibility. In this state no more messages are written to the ports. The *client* can trigger a transition to either the configuration state for executing another motion task with different parameters or to the cleanup state.

5) Cleanup: In the cleanup state, the allocated memory is freed and the solver objects are deleted.

VI. EXPERIMENTS

In this section we demonstrate the deployment of Tasho for two robotic applications: bin-picking and dual-arm laser contouring. In these applications we employ task templates to leverage their reusability.

A. Time optimal bin-picking application

A time optimal bin-picking application, where a robot transferred elements from one bin to another, was deployed. Each pick motion was programmed as a sequential composition of three P2P motion stages, each of which was quickly programmed by modifying a base P2P template. The first P2P stage substituted the 6D goal pose constraint with a terminal constraint requiring the gripper to be above the bin. The stationarity constraints were removed at the intersection of the stages in the P2P tasks allow smooth transitions. The second stage required the robot to reach a pre-grasp pose and the final P2P stage enforced a normal approach motion to maximize successful grasp chances with the suction cup. The last two stages avoided collision with the bin walls while the first stage's collision avoidance constrained simply modelled the bin as a solid box, with the remaining collision avoidance constraints being common for all the stages.

This application was deployed on a UR10 robot shown in Fig. 4. Free time option, multiple shooting discretization, prediction horizon of 10 OCP steps for the P2P stage and 5 OCP steps for the approach motion were provided as transcription options, while time optimality was selected as a control option within the task specification to optimize bin picks per hour. Using IPOPT and MA57 [24], fast motion trajectories were computed successfully in $\sim 100\ \mathrm{ms}$ and had an average execution time of $\sim 500\ \mathrm{ms}$. This allowed to reach a bin-pick cycle time of $\sim 5\ \mathrm{s}$, where the bottleneck was the perception module that returned the pose of the objects to pick.

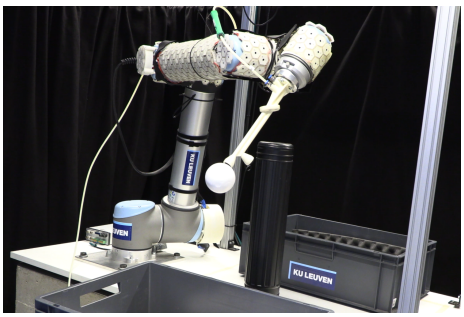

Fig. 4: Setup of the bin-picking task using a 7-dof UR10 robot manipulator with a suction cup.

B. Dual arm laser contouring

A complex dual arm laser contouring task was deployed on a dual arm ABB Yumi robot. Specifying this task is

significantly more complex than specfifying a bin-picking task. The left gripper of the robot points a laser at a workpiece held by the right gripper. The task requires the laser to trace a curve on the workpiece. Along with the usual constraints on system limits, the task requires the angle of incidence of the laser and the distance of the pointer from the workpiece to be within specified bounds. Furthermore, the user can specify a desired velocity of the contouring as well. The motion skill computed the jointspace velocity commands to achieve this desired task which was sent to the robot via the provided Orocos interface. For an MPC frequency of 20 Hz and a horizon of 13 sampling steps, equivalent to 650 ms, we obtained MPC computation times of $\sim$15 ms on average using IPOPT and MA57, and $\sim$30 ms in the worst case, which is well within the $\sim$50 ms MPC sampling time.

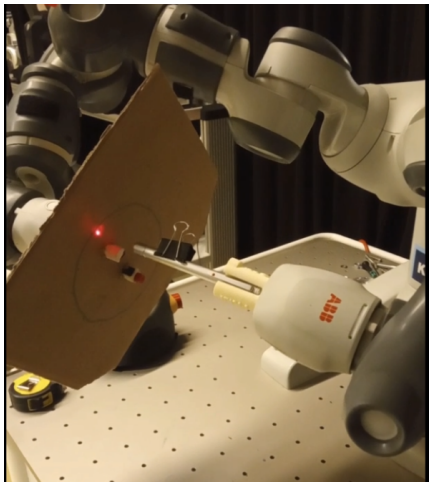

Fig. 5: Setup for the dual-arm laser contouring experiment performed by using an ABB Yumi robot.

VII. Discussion

As already mentioned in Section I, there exist multiple software frameworks or packages aimed for robot control and optimization. Some of them, such as CT, ALTRO and Crocoddyl provide implementations of fast OCP solution algorithms. Tasho, however, has a different focus aiming at the facilitating programming of constraint-based tasks and providing a fast solver is out of its scope. Nevertheless, we do aim to interface it with FATROP [25] in the near future, which is a fast solver for optimal control being developed in our group. Drake is another framework that has a similar focus on modeling as Tasho, but it is more mature with a significantly larger development team and scope (supports sum-of-squares programming, mixed integer programming and simulating deformable objects, to name a few features) compared to all the other packages mentioned. We will only consider the relevant features of Drake that overlap with Tasho's scope in the discussion below.

A comparison between Tasho and the aforementioned software packages is shown in Table I.

From the compared packages, only Crocoddyl and Tasho provide an interface with the *RBDL* Pinocchio which, unlike RobCoGen [26], RobotDynamics.jl [27] and Drake's RBDL implementation, implements efficient formulations of analytical derivatives of rigid-body dynamics. These analytical derivatives are thus used by Tasho and Crocoddyl within OCP solution algorithms and have been proven to reduce the solution times of OCPs in robotic applications [28].

Tasho relies on the general *AD tool* CasADi. Besides support for high order derivatives (not supported by Eigen's AutoDiffScalar [29]) and code-generation (not supported by AutoDiffScalar and ForwardDiff.jl [30]), CasADi offers great flexibility in terms of expression handling (with atomic variables optimized for memory footprint or performance), Jacobian overloading, serialization of expressions, and interfaces with efficient OCP solvers, unlike CppAD [31].

Features to add complex *custom constraints*, i.e., reusable constraint definitions, directly to the mathematical program and defining *monitor functions* appear to be present only in Drake and Tasho. Also sequential composition of tasks to generate *multi-stage* OCPs is not supported by ALTRO or CT. Specifying multi-stage OCPs is possible in Drake and Croccodyl, but with a higher programming effort than in Tasho. In Drake, one needs to manually specify the different dynamics and objective functions at different phases at NLP level. In Croccodyl, it is not clear how easy it is to program multi-stage tasks different from the examples involving different phases in walking.

Moreover, software abstractions like *task composition* are not provided in the other software frameworks. Most importantly, Tasho provides a *simplified workflow* completely in Python, from task specification to solution deployment, which makes programming robot tasks easier than in these other frameworks in our opinion.

Modifying task specifications by adding or removing constraints in Tasho can only be done either offline or online, but not during real-time execution of a controller. This is because generating a new CasADi solver object can take hundreds of milliseconds. Such real-time modification is not performed by other MPC packages either to the best of our knowledge.

VIII. Conclusions and Future Work

We presented Tasho, an easy to use open-source Python toolbox for robot task specification and control using MPC. We proposed and developed a modular framework for declarative task specification in the context of optimal control, that is flexible enough to formulate a large class of robot OCP/MPC problems. This modularity enables reuse of large parts of code. Interfaces are provided to a wide range or relevant libraries in robotics and optimization. The streamlined workflow allow faster and easier prototyping of robot tasks compared to existing tools. The option for adding complex monitors for predictive control during task specification opens up opportunities for rich interaction between the continuous-level controllers and the discrete-level controller. As future work, we will add other important functionalities to Tasho, such as the deployment interface with ROS and ROS2,

TABLE I: Comparison of Tasho against other similar software packages.

Software package	Language	RBDL	AD tool	Custom constraints	Monitors	Multi-stage	Task composition	Implementation of fast solvers*
CT	C++	RobCoGen	CppAD	✗	✗	✗	✗	✓
ALTRO	Julia	RobotDynamics.jl	ForwardDiff.jl	✗	✗	✗	✗	✓
Crocoddyl	C++**	Pinocchio***	CppAD	✗	✗	✓	✗	✓
Drake	C++**	Own implementation	AutoDiffScalar	✓	✓	✓	✗	✗
Tasho	Python	Pinocchio***	CasADi	✓	✓	✓	✓	✗

* Provides its own implementation of fast OCP solvers.
** Provides Python bindings.
*** Leverages Pinocchio's efficient formulation of analytical derivatives of rigid-body dynamics.

a graphical user interface, the interface with the FATROP solver, and the support for contact dynamics and floating-base robots.

ACKNOWLEDGMENT

The authors would like to thank Sunhong Kim for working with a preliminary version of Tasho and providing valuable suggestions for improvement.

REFERENCES

[1] J. De Schutter, T. De Laet, J. Rutgeerts, W. Decré, R. Smits, E. Aertbeliën, K. Claes, and H. Bruyninckx, "Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty," *The International Journal of Robotics Research*, vol. 26, no. 5, pp. 433–455, 2007.

[2] E. Aertbeliën and J. De Schutter, "etasl/etc: A constraint-based task specification language and robot controller using expression graphs," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2014, pp. 1540–1546.

[3] N. Mansard, O. Khatib, and A. Kheddar, "A unified approach to integrate unilateral constraints in the stack of tasks," *IEEE Transactions on Robotics*, vol. 25, no. 3, pp. 670–685, 2009.

[4] D. Mayne, J. Rawlings, C. Rao, and P. Scokaert, "Constrained model predictive control: Stability and optimality," *Automatica*, vol. 36, no. 6, pp. 789–814, Jun. 2000.

[5] M. Giftthaler, M. Neunert, M. Stäuble, J. Buchli, and M. Diehl, "A family of iterative gauss-newton shooting methods for nonlinear optimal control," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 1–9.

[6] T. A. Howell, B. E. Jackson, and Z. Manchester, "Altro: A fast solver for constrained trajectory optimization," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2019, pp. 7674–7679.

[7] C. Mastalli, R. Budhiraja, W. Merkt, G. Saurel, B. Hammoud, M. Naveau, J. Carpentier, L. Righetti, S. Vijayakumar, and N. Mansard, "Crocoddyl: An Efficient and Versatile Framework for Multi-Contact Optimal Control," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2020.

[8] R. Tedrake and the Drake Development Team, "Drake: Model-based design and verification for robotics," 2019. [Online]. Available: https://drake.mit.edu

[9] J. A. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "Casadi: a software framework for nonlinear optimization and optimal control," *Mathematical Programming Computation*, vol. 11, no. 1, pp. 1–36, 2019.

[10] J. Gillis, B. Vandewal, G. Pipeleers, and J. Swevers, "Effortless modeling of optimal control problems with rockit," in *39th Benelux Meeting on Systems and Control, Date: 2020/03/10-2020/03/12, Location: Elspeet, The Netherlands*, 2020.

[11] J. Carpentier, G. Saurel, G. Buondonno, J. Mirabel, F. Lamiraux, O. Stasse, and N. Mansard, "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)*. IEEE, 2019, pp. 614–619.

[12] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning," http://pybullet.org, 2016–2021.

[13] C. Aeronautiques, A. Howe, C. Knoblock, I. D. McDermott, A. Ram, M. Veloso, D. Weld, D. W. SRI, A. Barrett, D. Christianson *et al.*, "Pddl— the planning domain definition language," *Technical Report, Tech. Rep.*, 1998.

[14] H. Bruyninckx, "Open robot control software: the orocos project," in *Proceedings 2001 ICRA. IEEE international conference on robotics and automation (Cat. No. 01CH37164)*, vol. 3. IEEE, 2001, pp. 2523–2528.

[15] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[16] D. Vanthienen, M. Klotzbücher, and H. Bruyninckx, "The 5c-based architectural composition pattern: lessons learned from re-developing the itasc framework for constraint-based robot programming," *JOSER: Journal of Software Engineering for Robotics*, vol. 5, no. 1, pp. 17–35, 2014.

[17] J. B. Rawlings, D. Q. Mayne, and M. Diehl, *Model predictive control: theory, computation, and design*. Nob Hill Publishing Madison, 2017, vol. 2.

[18] L. T. Biegler and V. M. Zavala, "Large-scale nonlinear programming using ipopt: An integrating framework for enterprise-wide dynamic optimization," *Computers & Chemical Engineering*, vol. 33, no. 3, pp. 575–582, 2009.

[19] P. E. Gill, W. Murray, and M. A. Saunders, "Snopt: An sqp algorithm for large-scale constrained optimization," *SIAM review*, vol. 47, no. 1, pp. 99–131, 2005.

[20] R. A. Waltz and J. Nocedal, "Knitro 2.0 user's manual," *Ziena Optimization, Inc.[en ligne] disponible sur http://www. ziena. com (September, 2010)*, vol. 7, pp. 33–34, 2004.

[21] R. Verschueren, G. Frison, D. Kouzoupis, N. van Duijkeren, A. Zanelli, R. Quirynen, and M. Diehl, "Towards a modular software package for embedded optimization," *IFAC-PapersOnLine*, vol. 51, no. 20, pp. 374–380, 2018.

[22] A. S. Sathya, W. Decre, G. Pipeleers, and J. Swevers, "A simple formulation for fast prioritized optimal control of robots using weighted exact penalty functions," in *2022 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2022, pp. 5262–5269.

[23] M. Diehl, H. G. Bock, and J. P. Schlöder, "A real-time iteration scheme for nonlinear optimization in optimal feedback control," *SIAM Journal on Control and Optimization*, vol. 43, no. 5, pp. 1714–1736, Jan. 2005.

[24] "Hsl - a collection of fortran codes for large scale scientific computation," 2019. [Online]. Available: http://www.hsl.rl.ac.uk/

[25] L. Vanroye, M. Vochten, E. Aertbeliën, W. Decré, and J. De Schutter, "Efficient optimization-based calculation of coordinate-invariant trajectory shape descriptors for on-line applications." in *Presented at IROS 2020 workshop on Bringing geometric methods to robot learning, optimization and control, Location: online conference*, 2020.

[26] M. Frigerio, J. Buchli, D. G. Caldwell, and C. Semini, "RobCoGen: a code generator for efficient kinematics and dynamics of articulated robots, based on Domain Specific Languages," vol. 7, no. 1, pp. 36–54, 2016.

[27] T. Koolen and contributors, "Rigidbodydynamics.jl," 2016. [Online]. Available: https://github.com/JuliaRobotics/RigidBodyDynamics.jl

[28] A. Astudillo, J. Carpentier, J. Gillis, G. Pipeleers, and J. Swevers, "Mixed use of analytical derivatives and algorithmic differentiation for NMPC of robot manipulators," *IFAC-PapersOnLine*, vol. 54, no. 20, pp. 78–83, 2021.

[29] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[30] J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in Julia," *arXiv:1607.07892 [cs.MS]*, 2016. [Online]. Available: https://arxiv.org/abs/1607.07892

[31] B. Bell, "Cppad: a package for c++ algorithmic differentiation," 2018. [Online]. Available: http://www.coin-or.org/CppAD