Contents lists available at ScienceDirect

Robotics and Computer-Integrated Manufacturing

journal homepage: www.elsevier.com/locate/rcim

Full length article

Extending the motion planning framework—MoveIt with advanced manipulation functions for industrial applications

Pablo Malvido Fresnillo [a,*], Saigopal Vasudevan [a], Wael M. Mohammed [a], Jose L. Martinez Lastra [a], Jose A. Perez Garcia [b]

[a] FAST-Lab, Faculty of Engineering and Natural Sciences, Tampere University, Tampere, Finland
[b] Design and Fabrication in Industrial Engineering, University of Vigo, Vigo, Spain

ARTICLE INFO

Keywords:
Robot manipulation
MoveIt
Speed control
Dual-arm synchronization
Automatic tool changing

ABSTRACT

MoveIt is the primary software library for motion planning and mobile manipulation in ROS, and it incorporates the latest advances in motion planning, control and perception. However, it is still quite recent, and some important functions to build more advanced manipulation applications, required to robotize many manufacturing processes, have not been developed yet. MoveIt is an open source software, and it relies on the contributions from its community to keep improving and adding new features. Therefore, in this paper, its current state is analyzed to find out which are its main necessities and provide a solution to them. In particular, three gaps of MoveIt are addressed: the automatic tool changing at runtime, the generation of trajectories with full control over the end effector path and speed, and the generation of dual-arm trajectories using different synchronization policies. These functions have been tested with a Motoman SDA10F dual-arm robot, demonstrating their validity in different scenarios. All the developed solutions are generic and robot-agnostic, and they are openly available to be used to extend the capabilities of MoveIt.

1. Introduction

Robot manipulators have been utilized in industrial environments since the 1960s, and their usage has been growing exponentially in the following years, till present [1]. These industrial robots were initially a niche class of machinery that perform manual operations with high levels of accuracy and repeatability, boosting the productivity of the manufacturing lines. However, with the increasing manufacturing complexity of products and their production environments (with the arrival of complex sensing and actuation elements), robotic manipulators have grown in variety and functionality to best cater to the required needs. These robots needed to communicate with other industrial devices, which have their own proprietary system architectures and communication protocols [2]. Furthermore, the different manufacturers of the robot manipulators themselves had their own proprietary UIs, programming, and interfacing requirements. This was creating limitations for roboticists to efficiently utilize the best robot for the situation, as they required knowledge about robots from many manufacturers and the expertise to work with them. Additionally, the developed solutions were not easilytransferable and scalable, and mostlyrequired each individual integration scenario to be handled from scratch. To overcome these shortcomings, a generic robot framework called Robot Operating System (ROS) was introduced in 2007.

ROS is a meta-operating system for robots, being a crossover framework between an OS and a middleware. It has open-source repositories of proprietary software libraries and user developed system solutions and implementations, which can be used by the global user community to develop their automation applications [3]. Majority of the manufacturers of automation systems and devices develop and maintain libraries for their products, which can be used to utilize the device functionalities and integrate it with the other elements of the system. ROS provides features which include hardware abstraction, concurrent-resource handling, inter-platform operability, low-level device control, synchronous and asynchronous communication between processes, and package management—to create highly modular systems [4].

Regarding its applications, ROS is not just limited to research but is also perfectly valid for industrial applications, taking advantage of the high-level functionality of ROS and the reliability and safety of industrial robot controllers. In particular, there is a project called ROS-Industrial [5], whose aim is to extend the advanced capabilities of ROS for industrial hardware and applications. This project counts with big support and there is even a European project called ROSIN [6] working on improving it and amplifying its impact.

* Corresponding author.
E-mail address: pablo.malvidofresnillo@tuni.fi (P. Malvido Fresnillo).

https://doi.org/10.1016/j.rcim.2023.102559
Received 22 September 2022; Received in revised form 27 December 2022; Accepted 2 March 2023

There is a vast amount of ROS-based packages, which can be installed in ROS providing different functionalities, such as navigation, motion planning, coordinate frames tracking, robot controllers interfacing, or 3D simulation and visualization, to highlight a few. One of the fundamental functions in robotics is motion planning [7]. This functionality was originally implemented in ROS by the Arm Navigation packages [8]. However, this was a preliminary development which had several shortcomings, including communication bottleneck while transferring heavy 3D data to multiple nodes simultaneously; and synchronization issues between separate nodes. This caused inconsistencies between nodes, resulting in failed motion plans. These issues were addressed with the development of MoveIt.

The MoveIt framework is a set of software packages integrated with ROS that incorporates the latest advances in motion planning, manipulation, kinematics, 3D perception, control, and navigation [9]. Due to its reliability and its active user community, which keeps upgrading the existing features and extending its avenues of utilization, MoveIt is utilized in over 150 robots of all categories from some of the most marquee technical companies in the world, such as NASA [10], ABB, Yaskawa, or Fanuc [11]; and this compatibility catalog is still growing. Additionally, MoveIt has a BSD license, which enables its free utilization for research, commercial and industrial applications. However, MoveIt is still recent and some important functions for robotic manipulation have not been developed yet. In particular, some features required for many manufacturing applications, such as the automatic tool changing or the control of the end effector's speed [12], are not implemented. As it is open-source, the community has to help develop these necessary functions, and individual contributions are very welcome. The goal of this paper is, therefore, to analyze the current state of MoveIt to find its main current necessities and develop valid solutions for them.

The following sections of the paper are structured as follows: Section 2 delves deeper into the workings of MoveIt, highlighting its system architecture and capabilities; and showcases some of the gaps in its available features which require further development. These shortcomings are solved with the functions developed and presented in Section 3. The theoretically proposed functions are tested and evaluated for reliability and proof of concept in Section 4. Finally, Section 5 talks about the various insights gained during the development of these feature extending functions and its experimentation, along with possible avenues to be explored as future work.

2. MoveIt

2.1. Library background and insights

MoveIt is the primary software library for motion planning and mobile manipulation in ROS for both industrial [13,14] and research applications. Thus, MoveIt is also the core motion planning library for ROS-Industrial. One of its main characteristics is that it is robot-agnostic, and it can be used with a vast amount of robots from different vendors and with different structures, including single and dual-arm robots, humanoid robots, and mobile manipulation systems [15]. This is possible thanks to the configuration files of the MoveIt package of each robot, from which all the necessary information for the robot's motion planning is automatically extracted and managed by MoveIt. There are two main configuration files, the Unified Robot Description Format (URDF) and the Semantic Robot Description Format (SRDF). The URDF is an XML specification that describes the kinematic chains of the robot, including, among other properties, the dimensions and collision models of their links, and the type and limits of their joints [16]. The SRDF is another XML specification that complements the URDF file with additional information such as joint motion groups, predefined robot configurations, and the Allowed Collision Matrix (ACM), which specifies which pairs of links do not need to be checked against each other to reduce the computation time [17].

MoveIt has a central node called *move_group*, which integrates all its capabilities using a plugin-based architecture. This node can be interfaced by the users, calling its actions and services. There are two ways to do this, through a GUI in Rviz (the ROS visualizer), or through code, either in C++ or Python, being the interfacing packages very similar for both languages. Regarding the Python interface, the *moveit_commander* package [18] is used to access to the different *move_group* capabilities. This package has three main classes: *RobotCommander*, the outer-level interface to the robot, *PlanningSceneInterface*, the interface to the world surrounding the robot, and *MoveGroupCommander*, the interface to one group of joints (e.g. a robot arm, two robot arms, or the robot's torso). The motion planning is performed by the defined *MoveGroupCommander* objects, moving the joints of the group in question to achieve the target pose or configuration. This class offers three possibilities to plan and execute a movement: defining a target configuration for the joints of the group (*set_joint_value_target*, or *set_named_target* if the configuration has been predefined in the SRDF file) or a target pose for its end effector (*set_pose_target*) and then, planning and executing the movement towards that target, or planning a cartesian path (*compute_cartesian_path*) and executing it.

In the first two motion types, in which a target is defined for the group of joints, either in the cartesian or the joint space, the motion is free, meaning that there is no control on the path followed by the end effector (EEF) to reach the target pose. Due to this, the generated motion plans can be unexpected or seem unintuitive. Additionally, MoveIt normally uses randomized motion planners from the Open Motion Planning Library (OMPL), such as Rapidly-exploring Random Trees (RRTs), which are unrepeatable. Therefore, even the motion planning between two tested poses can generate an unexpected trajectory [19]. On the other hand, in the third motion planning option, *compute_cartesian_path*, there is more control on the end effector's path, as it generates a motion plan in which the end effector moves in straight line segments following the specified sequence of waypoints.

Using any of the three previous options, the motion planning retrieves a *RobotTrajectory* message [20], which contains a *JointTrajectory* message, composed by a sequence of *JointTrajectoryPoints*. Each of these *JointTrajectoryPoints* contains the joint values, velocities, accelerations, and efforts of all the joints of the planned group of joints (e.g., the joints of a robotic arm), as well as the time (since the beginning of the motion) at which that point is reached. Finally, this message, containing the motion plan, is sent to the ROS controllers, typically to a *joint_trajectory_controller*, that interfaces the robot to execute the planned trajectory [21].

2.2. Main necessities of MoveIt

Despite all the possibilities and the flexibility that MoveIt offers to the users, some important features have not been implemented yet. Several researchers have contributed to fixing this, providing solutions to improve or extend MoveIt capabilities. Examples of this are [22], that extended MoveIt to perform resolution-optimal inverse kinematics along a specified workspace trajectory, and [23], which provided ROS action interfaces for its existing capabilities and added the capability of defining parameterized cartesian paths like circles and helixes. With this objective, this paper presents a set of functions that extend MoveIt with some of its most needed capabilities. These capabilities are presented and analyzed below.

- **Automatic tool changing (ATC):** What a robot can do is largely determined by its end effector [24]. If a robot is used to perform a single specific task, the end effector can be directly bolted to its arm. However, if the product requires multiple or complicated manipulations, an automatic and quick way to change the tool is probably needed [25]. This is the case of many industrial processes, such as electronic assembly, where products have many variations that require their own tools; injection molding, where a single robot work with several molds, each of them requiring a different suction cup gripper [24]; drilling, to create holes of different sizes [26]; or die polishing, that requires changing the polishing pad [27]. Additionally, ATC demand is in continuous growth, with a projected market size value of 928 million USD by 2028 [28]. However, even being an extremely necessary functionality, especially for manufacturing applications [29,30], there is no official solution to do an ATC in MoveIt. The kinematic structure of the robot is described in its URDF configuration file, specifying the end effector that is attached to the robot's wrist. However, modifying this file in runtime is not an easy task, as the URDF has many consumer nodes and this should ensure that all these nodes are in a consistent state, using the same version.

Some implementations of ATC in MoveIt have been reported, but none of them update the collision models and the homogeneous transformation matrix to the actuation frame of the new tool. In [31] a robotic arm uses an automatic tool changer to switch between a three fingers gripper and a vacuum cleaner. The movements of the robot are planned using MoveIt, but there is not mention of any update in the kinematic chain or the collision models. Additionally, the authors suggest that the performance would improve significantly adding collision models in MoveIt, leading to safer, and more flexible arm movements. Something similar is done in [32], where a UR10 robot changes between a parallel and a magnetic gripper using an ATC. As in the previous approach, MoveIt is used for the motion planning, and a C++ node is used to control the tool changing, as well as to configure the tool center point and the payload.

End effector speed control: As introduced in Section 2.1, there are three options in MoveIt to plan movements for a group of joints, setting a target in the cartesian or the joint space, or indicating a list of waypoints through which the end effector of the group has to pass, moving in straight line segments. However, none of these options allow to control the speed at which the end effector moves. This velocity control is very important for many robotic applications, such as welding, to control the joint penetration weld [33], precision tasks that require slow end effector movements, or multi-robot synchronization [34]. Several authors have tried to address this problem, but none of the approaches offer a complete solution.

In [35], the robot velocity is controlled using the *set_max_velocity_scaling_factor* method of the *MoveGroupCommander* class, however, this function cannot be applied for cartesian paths, and it just sets a scaling factor to reduce the maximum joint velocities. In [36], the trajectory provided by MoveIt is modified, smoothing the curves of the joint angles and velocities using a quintic polynomial interpolation. This way, the end effector movements are more reliable and steady, however, there is no control over its cartesian velocity. The same is done in [37], but using an optimized S-shaped trajectory planning algorithm to smooth the angle, velocity and acceleration curves of the joints.

Both [38] and [39] focus on the cartesian speed control of robots in teleoperation applications, using speed commands to move them. In [38], these end effector speed commands are converted into goal joint positions or velocities using the inverse Jacobian method of the *moveit_servo* package of MoveIt. However, these two approaches are not applicable for trajectories that are planned in advance, as they just work with motion commands that are specified online (e.g., teleoperation and reactive locomotion tasks).

The solution proposed in [40] is to use an optimization-based cartesian controller to control the end effector speed. For this, the authors use a cost function that considers the quadratic error between the real cartesian velocity (calculated from the joint velocities) and the ideal cartesian velocity, where the constraints are the speed limits of the joints. This solution allows the control of the end-effector speed but requires an additional cartesian controller, instead of directly obtaining a motion plan with cartesian velocity control.

In [12], a numerical integration method is used to generate a motion plan with the optimal tool speed for a given path in a robotic machining application. The optimal tool speed for this application is defined as the one that allows finishing the process in the minimum time possible without exceeding any of the kinematics constraints. These constraints are the velocity, acceleration and jerk limits of the robot joints, and the velocity and acceleration limits of the tool in the cartesian space, as they are related to the cutting force and the modal excitation of the robot. The speed optimization is done in the cartesian space and MoveIt is used to compute the inverse kinematics. The resultant motion is very smooth, however, the high order of the constraints makes this approach significantly slower. In particular, the motion planner was tested for two use cases. In the first one, it took about 30 s to compute a trajectory, which MoveIt calculated in a negligible time; and in the second one, the planning time was about 10 min. This issue makes this approach inappropriate for robotic applications in which online planning is required (e.g., vision-based manipulation).

Dual-arm cartesian motion: Coordination and temporal synchronization of multiple robots [41] or of the arms of a dual-arm robot is required in many industrial applications [42]. The motion planning and inverse kinematics solving plugins of MoveIt work only for connected chains [43]. A dual-arm group is composed of two sub-groups that do not form a chain; thus, none of the current functions of MoveIt can be used directly for planning its motion. However, there is a way of specifying dual-arm goal targets in MoveIt, which consists of setting a target for each arm group individually, in the cartesian or the joint space, and then executing the movement of both arms simultaneously through the dual-arm group [44]. This is done in some approaches for planning dual-arm [45,46], or even upper-body (arms + torso) [47] coordinated movements in MoveIt. However, it has many limitations, as there is no control over the arms' trajectories, and it is impossible to coordinate their movements. Something more advanced is done in [43], where, in order to have synchronized dual-arm movements, a cartesian path is planned for one of the arms, and then, the joint values of the generated plan are mirrored for the second arm. This has still many limitations, as the arms configuration must be symmetrical all the time, including their initial position.

To have more control over the trajectory of both arms, [48] evaluates the possibility of computing a cartesian path plan for each arm and then executing both simultaneously and independently. The author concludes that this is not possible because each trajectory is calculated independently, without taking into account the movement of the other arm, and due to this, the collision detection is not reliable. One of the problems derived from this happens when, in a dual-arm motion, one of the arms is going to pass through the initial position of the other. In this case, as the motion planner of this arm is not aware of the simultaneous motion of the other arm, it will consider that it will not move for the collision checking. Therefore, when it tries to plan the motion to this position, it will detect a collision between arms, returning an error.

A solution for this is presented in [49]. In this approach, first, all the waypoints of both arms are calculated. Then, the sequence of waypoints of each arm is analyzed in order and, if this situation is detected (i.e., a waypoint of one arm closer than a certain threshold distance to the initial position of the other arm), the waypoints are separated in two consecutive lists. Then, a plan for the first list of waypoints is computed for each arm independently, ensuring no planning errors due to self-collisions. Next, the two resultant plans are merged into a single dual-arm plan, and it is executed. When the execution finishes, the process is repeated for the rest of the waypoints. This approach overcomes some of the problems of the previous approaches, however, this is just a patch for the erroneous dual-arm collision-checking, but it does not solve the problem, as the collisions derived from the simultaneous motion of both arms will not be detected. Additionally, if the trajectory needs to be split in successive motion plans, the arms will stop in the final poses of every plan, plan the next trajectory, and start moving again. Thus, even if the motion planning is very fast, the robot arms will be constantly accelerating and decelerating and their movement will not be smooth.

A complete solution to add the dual-arm capability to MoveIt is presented in [50]. This approach applies a single-query bidirectional RRT planner extended to closed kinematic chains. To achieve this, the exploration tree uses a fast dual-arm constraint sampler package that generates valid configurations for the torso and arms of a robot to reach a given dual-arm target pose, and checks their validity with respect to collisions. The problem with this approach is that it is tailored for generating free-collision motion plans for closed-chain pick and place operations, and it does not give the user too much flexibility for defining the trajectory of the arms. Therefore, this approach would not be appropriate for applications in which the relative pose between both end effectors is not constant, such as folding clothes, opening a jar, joining the pieces grasped by each hand, or performing independent but synchronized dual-arm motions (e.g., each arm picking different objects from the same box simultaneously).

Something similar happens in [51], where another complete framework for planning closed-chain pick and place operations is presented. Additionally, in this approach, the captured objects are moving and have high momentum, increasing its complexity. To achieve this, the dual-arm coordinated motion is computed in the null space of the master arm using the relative Jacobian. Thus, the required trajectory is calculated for the master arm and the slave arm coordinates with it. Moreover, impedance control is used to adjust the operational forces of the arms to compensate for the momentum of the manipulated objects. This approach shows excellent results, capturing objects of up to 10 kg, moving at 0.1 m/s. However, as with the previous approach, this dual-arm motion planning framework is targeted for a very specific application, not being applicable for dual serial kinematic chain operations. Furthermore, this approach is not integrated with MoveIt, and therefore, it cannot leverage its enormous list of advantages, which were presented in Sections 1 and 2.1 (e.g., easy configuration, flexibility, compatibility with many ROS packages, support, and integration of a variety of tools for perception, collision checking, scene planning, kinematic calculations, etc.).

Finally, a complete and generic solution for implementing dual-arm manipulation in ROS is presented in [52], but as in the previous approach, MoveIt is not used for motion planning. A custom-built framework is used instead to coordinate robotic dual-arm systems. This framework integrates different modules for the synchronization and coordination of the arms, as well as for checking self-collisions and collisions with the environment, allowing it to be used for any dual-arm system and application. However, due to the use of a custom framework, this approach does not provide either a complete and generic solution for the dual-arm motion planning in MoveIt, being still one of its main necessities.

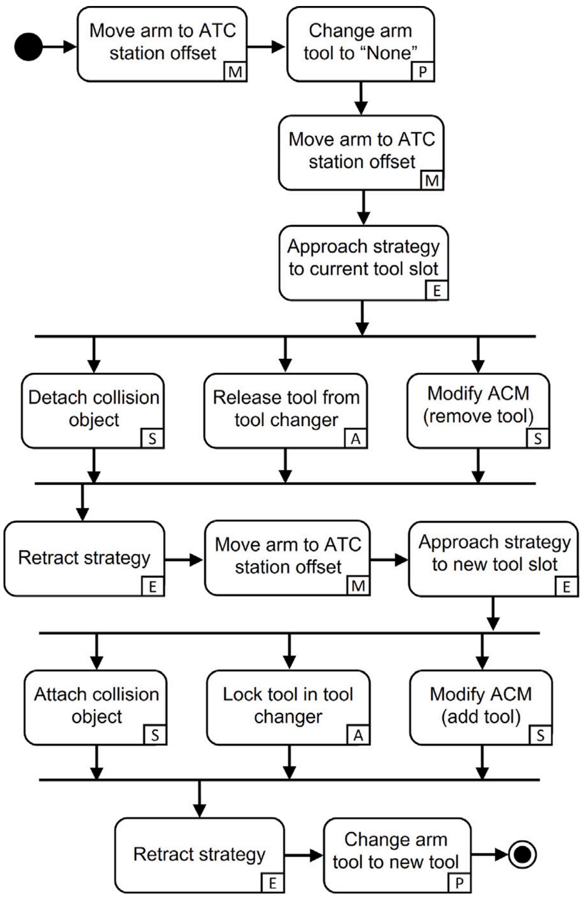

Move arm to ATC station offset M
Change arm tool to "None" P
Move arm to ATC station offset M
Approach strategy to current tool slot E
Detach collision object S
Release tool from tool changer A
Modify ACM (remove tool) S
Retract strategy E
Move arm to ATC station offset M
Approach strategy to new tool slot E
Attach collision object S
Lock tool in tool changer A
Modify ACM (add tool) S
Retract strategy E
Change arm tool to new tool P

Fig. 1. UML activity diagram of the ATC process. M: Arm motion using *MoveGroupCommander* methods, E: Arm motion with speed control using the MoveIt extended function presented in Section 3.2, S: Scene modification, P: Parameter modification, A: Actuation of the tool changer.

3. Advanced manipulation functions for MoveIt

This section presents the functions developed to fill the current gaps in MoveIt, analyzed in Section 2.2. These functions are generic and can be implemented in any robot with slight or no modifications. All these functions have been developed for ROS 1 and MoveIt 1, and they are publicly available.[1]

3.1. Automatic tool changing

The automatic tool changing capability is added to MoveIt with the *ATC* class, described in this section. For its integration with MoveIt, this class uses the *PlanningSceneInterface* and *MoveGroupCommander* instances that the user defines for the motion planning application. Thus, the *ATC* class can call the methods of these instances, as well as the extended MoveIt function defined in Section 3.2, to perform modifications in the scene and move the robot with and without speed control whenever required (see Fig. 1).

In the developed approach, the robot's end effectors are added to the scene as collision objects, which can be attached and detached to the last link of the required *motion group* by using the methods of the *PlanningSceneInterface* class. The models used for the collision objects are the STL meshes of the utilized tools. Simplified collision models of the tools can be used instead, reducing the computation time

[1] https://github.com/pablomalvido/Advanced_manipulation_moveit

significantly. This approach is not applicable to tools that can change their structure configuration and work in narrow spaces, as the collision models used are static. However, this is just a minor case, and the approach could be used for all the cases presented in Section 2.2.

The inclusion of these objects in the scene is managed by the class *constructor* method. Additionally, this method assigns and attaches the tools to any of the robot arms (if required), and stores information about the tools in class attributes. This information includes their names, the pose of their slots in the ATC platform, and the homogeneous transformation matrix from their bases to their actuation frames. All this knowledge about the tools is used later by the other methods of the class to configure the robot movements, correct the target poses, or provide information to external functions (e.g., the dimensions of the attached tool might be requested outside the class to define the waypoints of the robot's path).

changeTool is the main method of the *ATC* class. The inputs of this method are the name of the new tool, the *motion group* that is changing the tool, and the name assigned to this group; and it manages the robot movements during the tool changing process, the attachment and detachment of the collision objects, the communication with the automatic tool changer, the modification of the ACM, and the update of the name and actuation frame of the tool assigned to the robot arm. The motion strategy and the full activity diagram of the ATC process is shown in Fig. 1.

First, the arm in question moves to an offset position from the ATC platform. In case the robot has additional *motion groups*, slight modifications might be required, such as moving the robot torso, or moving away a second arm to leave space to the arm that is changing the tool to move. Then, the information of the arm is updated, indicating already that no tool is attached to it, as the rest of the movements will be planned for the wrist frame, and not for the actuation frame of the end effector. After this, the arm performs an approach motion to the pose of the tool slot, in order to release it there. The approach strategy considers a vertical insertion of the tools, but it might differ depending on the geometry of the slots of the ATC platform. The approach motion is done at reduced speed using the end effector speed control function of Section 3.2.

When the tool is in position, the collision object is detached and a message is sent to the tool changer to release the end effector. The communication protocol have to be modified to be compatible with the utilized tool changer. The current implementation calls a ROS service that allows to modify the value of the digital outputs of the robot controller. With one of these outputs, the pneumatic line connected to the tool changer is controlled. Additionally, the ACM matrix is modified, removing the released tool from it.

Then, the arm is retracted and moved again to the ATC station offset, so it can start approaching to the new tool. As for the first approach motion, all the approach and retract strategies depends on the ATC station geometry and might need to be modified. Additionally, all of them are performed with reduced speed, using the speed control function presented in Section 3.2. Once the arm is in place, the collision model of the new tool is attached, the tool changer is actuated to lock the tool, and the ACM is modified to include the allowed collisions for the new end effector. Finally, the arm is retracted and its information is updated with the name and actuation frame of the new tool.

In this approach, the *motion groups* of the robot arms do not contain any information about the end effector, hence, the motion planners calculate the movement of the robot wrist to the target poses. However, in most of the cases, the target poses have to be reached by the actuation frame of the end effector. The *correctPose* method of the *ATC* class was defined to manage these situations, correcting the target poses with the homogeneous transformation matrix from the frame of the robot wrist to the actuation frame of the tool. The matrix of each tool was defined in the class constructor, and the tool attached to each arm is updated everytime it changes, therefore, this method knows which transformation to apply for each arm, and the only arguments needed are the target pose and the name of the *motion group*. If the arm does not have any attached tool, the input target pose is not modified, so the movements are still planned to the robot wrist.

3.2. End effector speed control

This function allows to control both the path and the speed of an end effector's trajectory in MoveIt. The control of the path is achieved using the *compute_cartesian_path* method of the *MoveGroupCommander* class, which allows to specify a sequence of waypoints through which the end effector will move, doing straight line motions. The control of the linear and angular speed of the end effector is done by applying the equations of a trapezoidal velocity profile to the calculated path, determining the times, as well as the required joint velocities and accelerations, for the final motion plan. The selection of the velocity profile was done according to the guidelines of [53] and [54], balancing precision and computation time. Finally, a trapezoidal velocity profile was selected instead of a third or quintic-order profile because the effect of the acceleration slope would not be noticeable with the low sampling rate of the generated *JointTrajectory* message, which does not justify the increase in complexity and computation time. Additionally, other reasons that led to this decision were the broad use of trapezoidal profiles in the industry [55] and the fact that the jerk of the joints' actuators is not infinite, which smooths the real profile slightly. All the equations in this section are derived from the uniformly accelerated rectilinear motion equations [54].

Before continuing with the description of the approach, Table 1 must be checked. This table defines the nomenclature used in Sections 3 and 4. The terms described in this table will be used during the rest of the document, so it is crucial to understand them.

The function presented in this section allows the definition of trajectories with complex velocity profiles (i.e., trajectories with multiple target speed sections). This can be specified by the user with the following input parameters: a list of lists of waypoints (each sublist containing the sequence of waypoints of a target speed section), a list with the target linear speed of each section, and the maximum linear acceleration of the EEF. Additionally, the angular speed can be controlled in the same way, specifying a list with the target angular speeds for each target speed section, and the maximum angular acceleration of the EEF, but this is optional.

This function has three constraints. First, the target linear and angular speeds of each section of the trajectory can never be exceeded, prioritizing safety. Due to this, the transition between two target speeds must happen within the section with higher target speed. This means that the speed must decrease before reaching the first waypoint of the section with the lower target speed (in case of a deceleration), or that it must increase after the last waypoint of the section with the lower speed (in case of an acceleration). Second, the maximum linear and angular accelerations can never be exceeded, and third, all the joint velocities must be lower than their limits, which are read from the URDF file. Additionally, if the user wishes to virtually reduce these limits, this can be done by specifying the joint velocity limits' percentage (φ), which is 100% by default.

As introduced at the beginning of the section, the path of the end effector is calculated using the *compute_cartesian_path* method. This method returns a *RobotTrajectory* message with a motion plan, however, this does not have any control over the end effector's cartesian speed. Therefore, just the sequence of joint trajectory configurations is considered, guaranteeing the control in the end effector's path, and the trajectory times at which each of them will be reached are calculated by the developed function.

The first step for calculating the trajectory times is to determine the minimum distance ($\Delta X_{s_{min}}$) and angle ($\Delta\theta_{s_{min}}$) required to perform the target speed changes (ΔV_s) between sections. This is required to satisfy the first and second constraints.

Table 1
Nomenclature description.

Term	Definition
*Waypoints $[p_k]$	The sequence of poses specified by the user that the robot's EEF must follow. The waypoint's index is represented as k, and the total number of waypoints is represented as n.
*Target speed section (or section) $[s]$	Part of the trajectory with a common target linear and angular speed. The section's index is represented as s.
*Target linear speed/velocity (or target speed/velocity) $[V_s]$	Linear velocity (in the cartesian space) at which ideally the EEF should move within a section.
*Target angular speed/velocity $[\omega_s]$	Angular velocity (in the cartesian space) at which ideally the EEF should move within a section.
JointTrajectoryPoints	Points that compose the *JointTrajectory* ROS message. Each *JointTrajectoryPoint* is composed of a trajectory time and the joint trajectory values at this time. The index of the *JointTrajectoryPoints* is represented as i.
Trajectory times $[t_i]$	The time, from the beginning of the motion plan, at which the planned group of joints (e.g., all the joints of one arm for a single-arm motion plan) will have certain joint trajectory values.
Joint trajectory values $[q_{i,j}, \dot{q}_{i,j}, \ddot{q}_{i,j}]$	The configuration ($q_{i,j}$), velocities ($\dot{q}_{i,j}$) and accelerations ($\ddot{q}_{i,j}$) ($\forall j$) (in the joint space) that the planned group of joints will have at a certain trajectory time. The index of the joints is represented as j.
Trajectory configuration $[q_{i,j}]$	The configuration ($q_{i,j}$) of the planned group of joints at a certain trajectory time. The trajectory configurations are included in the joint trajectory values.
Trajectory pose $[x_i]$	The pose at which the EEF will be for at a certain trajectory time (i.e., the joint trajectory configuration converted to the cartesian space).
v_i	EEF linear velocity (in the cartesian space) at a certain trajectory time.
a_i	EEF linear acceleration (in the cartesian space) at a certain trajectory time.
*Maximum linear acceleration $[a_{max}]$	Maximum linear acceleration of the EEF (in the cartesian space). a_i cannot exceed this value.
*Maximum angular acceleration $[\alpha_{max}]$	Maximum angular acceleration of the EEF (in the cartesian space).
$\Delta t_{s_{min}}$	Time required to perform the target speed transition between two sections using the maximum acceleration (a_{max}).
ΔV_s	Target speed difference between two consecutive sections ($V_{s+1} - V_s$).
ΔX_{s_i}	Remaining distance (in the cartesian space) that the EEF must travel from the trajectory pose i until the end of the section.
$\Delta X_{s_{tot}}$	Total distance (in the cartesian space) traveled by the EEF within a section.
$\Delta \theta_{s_{tot}}$	Total angle (in the cartesian space) rotated by the EEF within a section.
*Step $[\varepsilon]$	Distance (in the cartesian space) between two consecutive trajectory poses.
$*\varphi$	Virtual joint speed limits. It is defined as a percentage of the real joint speed limits specified in the URDF file. It is 100% by default.

The terms with an * are input parameters of the functions presented in Sections 3.2, 3.3, 3.4.

$$\Delta t_{s_{min}} = \frac{\Delta V_s}{a_{max}} \tag{1}$$

$$\Delta X_{s_{min}} = V_s\, \Delta t_{s_{min}} + \frac{1}{2}\, a_{max}\, \Delta t_{s_{min}}^2 \tag{2}$$

Where s is the index of the section from which it is transitioning, V_s is the target speed of section s, a_{max} is the maximum end effector linear acceleration (specified as an input parameter), which is positive in case of a speed increase and negative otherwise, and $\Delta t_{s_{min}}$ is the minimum time required for the velocity transition. The equivalent equations apply to the angular movements and velocities.

As the velocity control is done in the cartesian space, all the obtained trajectory configurations are converted into poses (trajectory poses) using a forward kinematics solver. Then, as the plan is discrete, if the target speed of the next section is lower, the remaining distance from every trajectory pose until the end of the section (ΔX_{s_i}) is checked, to determine when the speed transition has to start (ΔX_{s_r}), as can be seen in Fig. 2. Then, the trajectory times at which every trajectory pose is reached are calculated according to the following equations:

$$\Delta X_{s_r} = \Delta X_{s_i} \,:\, \Delta X_{s_i} \geq \Delta X_{s_{min}} \wedge \Delta X_{s_{i+1}} < \Delta X_{s_{min}} \tag{3}$$

$$a_i = \begin{cases} 0, & \text{if } \Delta X_{s_i} > \Delta X_{s_r}; \quad (a) \\ \dfrac{2V_s \Delta V_s + \Delta V_s{}^2}{2\Delta X_{s_r}}, & \text{otherwise.} \quad (b) \end{cases} \tag{4}$$

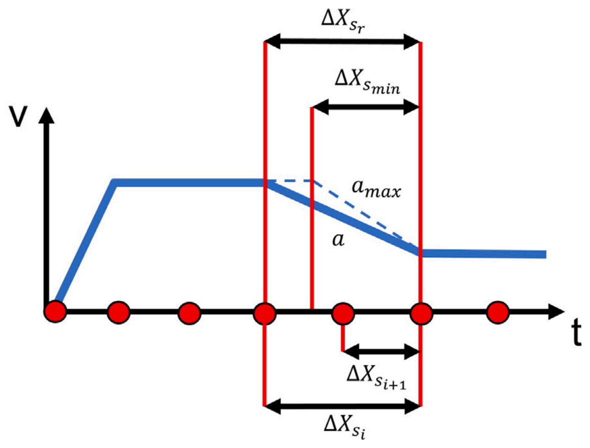


Fig. 2. Representation of the real and minimum distance required for a sections' transition. The red circles represent the trajectory times of the different trajectory poses. ΔX_{s_r} and $\Delta X_{s_{min}}$ are shown in the graph to represent graphically Eq. (3), however, the correct variables should be Δt_{s_r} and $\Delta t_{s_{min}}$ respectively, as the abscissa axis represents time.

$$\Delta t_i = \begin{cases} \dfrac{|x_i, x_{i-1}|}{V_s}, & \text{if } a_i = 0; \quad \text{(a)} \\ \dfrac{-v_{i-1} \pm \sqrt{v_{i-1}^2 + 2a_i\,|x_i, x_{i-1}|}}{a_i}, & \text{otherwise.} \quad \text{(b)} \end{cases} \tag{5}$$

$$t_i = t_{i-1} + \Delta t_i \tag{6}$$

$$v_i = v_{i-1} + a_i\,\Delta t_i \tag{7}$$

Where i is the index of the *JointTrajectoryPoint*, and x_i, v_i, a_i and t_i are the trajectory pose, the cartesian velocity and acceleration, and the trajectory time associated with this point. The sign of the square root in (5b) is the one that produces the minimum, positive time increase. In case there is a target velocity decrease, the speed transition starts at the beginning of the next section, and the equivalent equations are applied to calculate the times and speeds for every trajectory configuration.

Then, the calculated times are used to determine the joint velocities ($\dot{q}_i$) and accelerations ($\ddot{q}_i$) at each trajectory configuration (q_i) using the following equations, in which j represents the index of the joint:

$$\ddot{q}_{i,j} = \frac{2}{\Delta t_i^2}(q_{i,j} - q_{i-1,j} - \dot{q}_{i-1,j}\,\Delta t_i) \tag{8}$$

$$\dot{q}_{i,j} = \min(\dot{q}_{i-1,j} + \ddot{q}_{i,j}\,\Delta t_i,\ \varphi\,\dot{q}_{j_{max}}) \tag{9}$$

In case any of the resultant joint velocities is higher than its real or virtual (if reduced by φ) limit, it gets that value, as can be seen in (9), and the joint acceleration and the trajectory time are recalculated.

$$\ddot{q}_{i,j} = \frac{2\dot{q}_{i-1,j}(\dot{q}_{i,j} - \dot{q}_{i-1,j}) + (\dot{q}_{i,j} - \dot{q}_{i-1,j})^2}{2(q_{i,j} - q_{i-1,j})} \tag{10}$$

$$\Delta t_{i,j} = \frac{\dot{q}_{i,j} - \dot{q}_{i-1,j}}{\ddot{q}_{i,j}} \tag{11}$$

After calculating all the joint speeds and accelerations of a *JointTrajectoryPoint*, if $\Delta t_{i,j}$ had to be recalculated with (10) (11) for any of the joints, Δt_i is updated according to:

$$\Delta t_i = \max_j(\Delta t_{i,j}) \tag{12}$$

And all the joint speeds and accelerations of the *JointTrajectoryPoint* are recalculated for the updated trajectory time with (8) and (9). Finally, when none of the joint speed limits is exceeded, the calculated *JointTrajectoryPoint* is added to the motion plan message.

An additional case must be considered to ensure that the first constraint of the function is satisfied. Even if the maximum acceleration of the end effector is not enough to reach the target speed of a section, the target speeds can never be exceeded. Therefore, if the distance needed to accelerate and decelerate in a certain target speed section is higher than the total distance traveled on the section ($\Delta X_{s_{tot}} < \Delta X_{s-1_{min}} + \Delta X_{s_{min}}$), the end effector must stop accelerating before reaching the target speed, to guarantee that it will decelerate on time. This situation is represented graphically in Fig. 3, and the distance at which the end effector must start reducing its speed (ΔX_s) is calculated with the following equations:

$$\Delta X_{s-1} = V_{s-1}\Delta t_{s-1} + \frac{1}{2}\,a_{max}\,\Delta t_{s-1}^2 \tag{13}$$

$$V_s' = V_{s-1} + a_{max}\,\Delta t_{s-1} \tag{14}$$

$$\Delta X_s = V_s'\,\Delta t_s - \frac{1}{2}\,a_{max}\,\Delta t_s^2 \tag{15}$$

$$V_{s+1} = V_s' - a_{max}\,\Delta t_s \tag{16}$$

$$\Delta X_{s_{tot}} = \Delta X_{s-1} + \Delta X_s \tag{17}$$

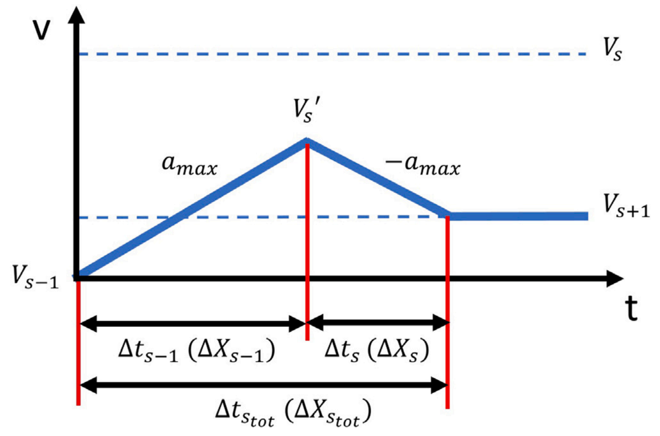


Fig. 3. Representation of the case in which the acceleration is not enough to reach the target speed of the section. The distance required to decelerate the end effector to the next target speed before the end of the section has to be calculated.

Where $\Delta X_{s-1}, \Delta X_s, \Delta t_{s-1}, \Delta t_s$ and V_s' are unknown, and the meaning of all the variables is graphically explained in Fig. 3. Solving this equation system, the following expressions are obtained:

$$\Delta t_s = \frac{-2V_{s+1} + \sqrt{2(V_{s+1}^2 + V_{s-1}^2 + 2a_{max}\Delta X_{s_{tot}})}}{2\,a_{max}} \tag{18}$$

$$\Delta X_s = (V_{s+1} + a_{max}\Delta t_s)\Delta t_s - \frac{1}{2}a_{max}\Delta t_s^2 \tag{19}$$

All the equations presented in this section are for the calculation of linear motions, but the equivalent equations are applied for the end effector angular motions and the angular target speeds. However, when there are both linear and angular speed targets, in most cases it is not possible to satisfy both at the same time, as the path is already defined and the linear and angular targets produce different times. Therefore, in these cases, the final time steps will be the highest of the linear (Δt_{X_i}) and angular (Δt_{θ_i}) time steps.

$$\forall i, \quad \Delta t_i = \max(\Delta t_{X_i}, \Delta t_{\theta_i}) \tag{20}$$

This will make one of the speeds constant at its target speed (the one with the highest time) and the other variable below its target speed. The whole process described in this section is summarized in Fig. 4.

3.3. Dual-arm cartesian motion

This function plans the trajectory of each of the arms individually, with (using the function of Section 3.2) or without (using the *compute_cartesian_path* method) velocity control, and then merges both plans, synchronizing the motion of the arms according to different policies. As mentioned in Section 2.2, computing the motion plan of each arm independently and executing them simultaneously makes the self-collision checking unreliable, as it does not take into account the simultaneous motion of the arms. However, collision checking is still useful to avoid collisions with the environment or with its torso. Therefore, the plan of each arm is generated using collision checking, but the allowed collision matrix is modified before to not take into account collisions between the arms. This kind of collision must be checked only when the plans are merged, otherwise the simultaneous motion of the arms will not be considered. Before continuing with the description of the approach, Table 1 must be checked, which defines the nomenclature that will be used throughout this section.

Once the motion plan of each arm is generated, the function merges them into a single plan for the dual-arm group, including the joint trajectory values of both arms. However, as the two single-arm motion plans were generated independently, their trajectory times are different, and hence, they cannot be merged directly. Due to this, their joint

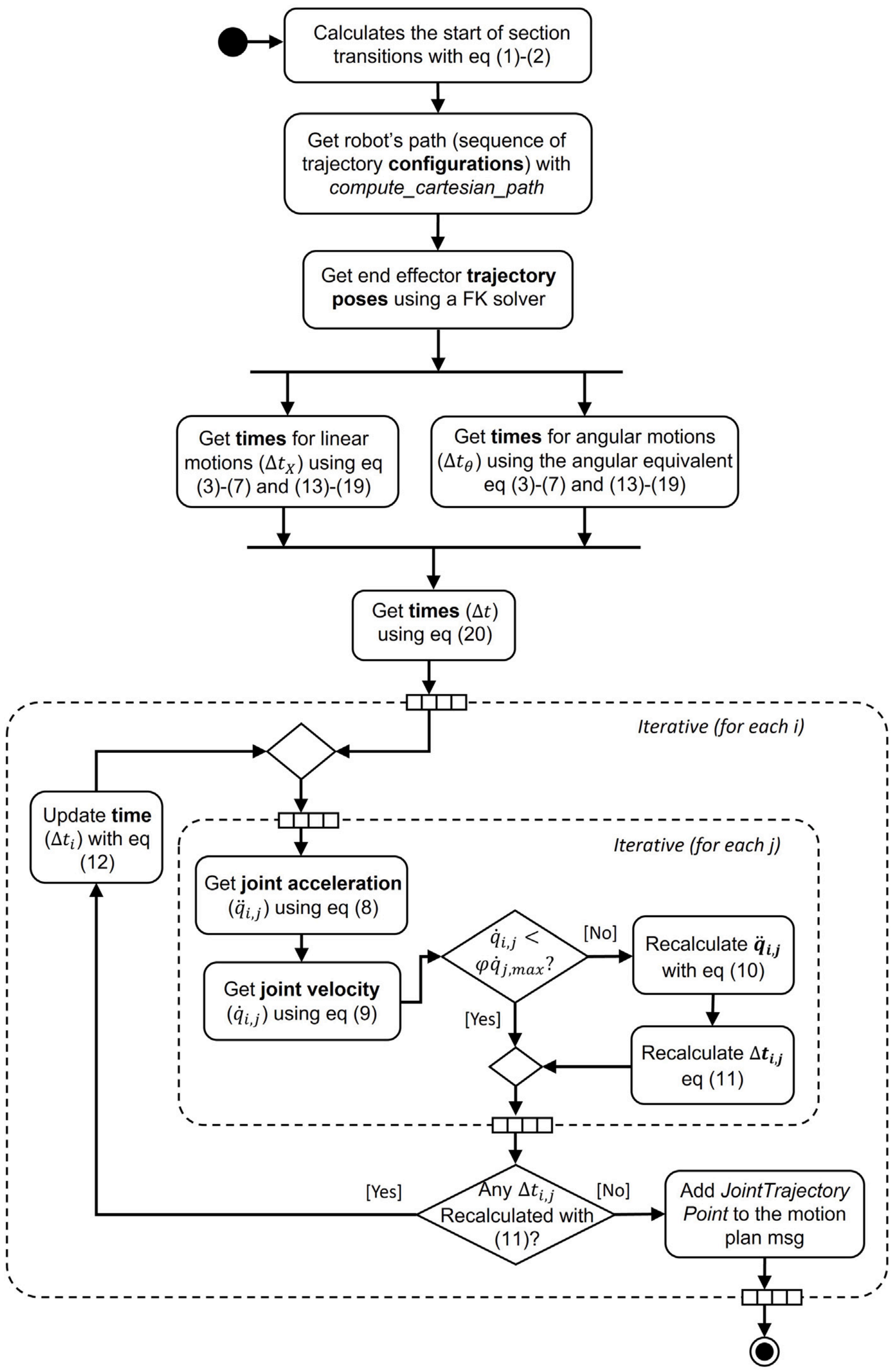

Calculates the start of section transitions with eq (1)-(2)
Get robot's path (sequence of trajectory configurations) with compute_cartesian_path
Get end effector trajectory poses using a FK solver
Get times for linear motions (Δt_X) using eq (3)-(7) and (13)-(19)
Get times for angular motions (Δt_θ) using the angular equivalent eq (3)-(7) and (13)-(19)
Get times (Δt) using eq (20)
Iterative (for each i)
Update time (Δt_i) with eq (12)
Iterative (for each j)
Get joint acceleration ($\ddot{q}_{i,j}$) using eq (8)
Get joint velocity ($\dot{q}_{i,j}$) using eq (9)
$\dot{q}_{i,j} < \varphi \dot{q}_{j,max}$?
[No]
[Yes]
Recalculate $\ddot{q}_{i,j}$ with eq (10)
Recalculate $\Delta t_{i,j}$ eq (11)
[Yes]
Any $\Delta t_{i,j}$ Recalculated with (11)?
[No]
Add JointTrajectory Point to the motion plan msg

Fig. 4. UML activity diagram of the end effector speed control function.

trajectory values must be recalculated for the trajectory times used in the merged plan. Three different synchronization policies can be selected to merge the motion plans: (i) starting and finishing the motion of both arms at the same time, (ii) moving each arm at the calculated speed, or (iii) reaching several waypoints simultaneously by both arms.

In the first policy, when both arms finish moving at the same time, the total execution time is the time of the slowest arm, and the speed of the fastest arm is adjusted accordingly. Therefore, to construct the merged plan, the trajectory times used are the ones of the plan of the slowest arm. Thus, the joint trajectory values of this arm that are required for the merged plan are already known. However, for the other arm, the joint trajectory values for each trajectory time of the merged plan are calculated by linear interpolation between its joint trajectory values at its previous and next trajectory times. To achieve a smooth motion of the arms, with a stable speed, the number of points of the individual motion plans to merge must be as similar as possible. This is done by adjusting the end-effector step parameter of the *compute_cartesian_path* method, for the arm with the shortest motion (i.e., the one that travels the shortest distance). This step (ε) defines the cartesian distance between the generated trajectory poses, and it is adjusted according to the following equation:

$$\varepsilon_B = \frac{\varepsilon_L \sum_{k=0}^{n_L-1} (|p_{L_k}, p_{L_{k+1}}|)}{\sum_{k=0}^{n_B-1} (|p_{B_k}, p_{B_{k+1}}|)} \tag{21}$$

Where the subindexes B and L represent the shortest (briefest) and longest plan respectively, n is the number of waypoints specified to

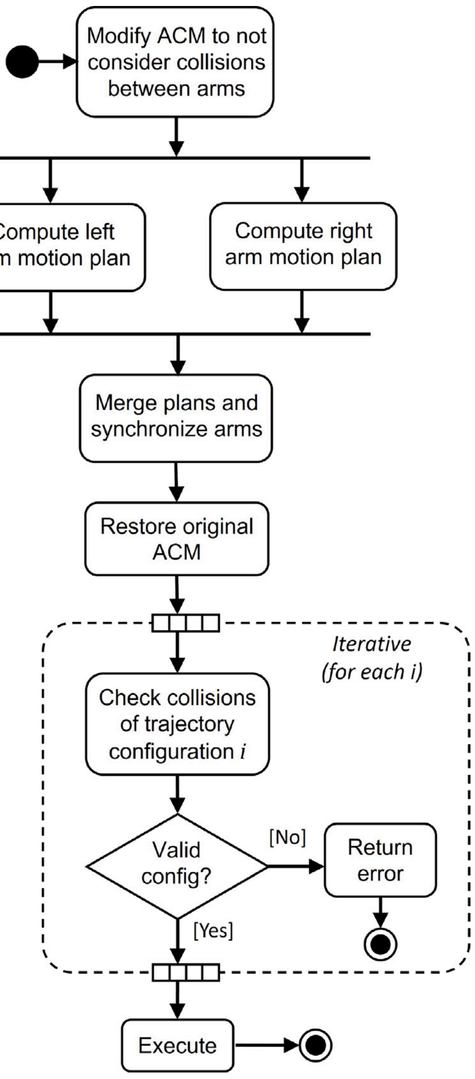

Modify ACM to not consider collisions between arms
Compute left arm motion plan
Compute right arm motion plan
Merge plans and synchronize arms
Restore original ACM
Iterative (for each i)
Check collisions of trajectory configuration i
Valid config?
[No]
Return error
[Yes]
Execute

Fig. 5. UML activity diagram of the dual-arm cartesian motion function.

generate the plan, k is the index of the waypoint, and p represents the list of waypoints.

In the second policy, each arm moves at its own target speed, which is targeted for un-coordinated manipulation applications [56]. The merged dual-arm plan contains all the trajectory times of both individual plans. For each arm, the trajectory values corresponding to the trajectory times of the other arm are calculated by applying Eqs. (6), (22), (8), and (9). For these calculations, the initial and final conditions considered are the trajectory values of the previous and next point with respect to the evaluated time. However, as each arm will finish its trajectory at a different time, these equations cannot be applied when, for one of the arms, the evaluated time is higher than all the trajectory times of its plan. In this case, for this arm, the new trajectory configuration will be the same as for its last JointTrajectoryPoint, and the new joint velocities and accelerations will be set to zero, so it remains in the final position while the other arm finishes its trajectory.

$$q_{i,j} = q_{i-1,j} + \dot{q}_{i-1,j}\,\Delta t_i + \frac{1}{2}\,\ddot{q}_{i-1,j}\,\Delta t_i^2 \tag{22}$$

The third and last policy, synchronizes both arms to reach a set of poses at the same time. A dual-arm plan is generated between every pair of waypoints, using the first synchronization policy, obtaining a sequence of plans that start and finish at the same time. Then, all these dual-arm plans are merged into one, recalculating the trajectory times and the joint speeds and accelerations from the beginning of the deceleration of the previous plan until the end of the acceleration of the next plan, using the Eqs. (5)–(9). This guarantees a smooth transition between the different target speeds of the plans, respecting the maximum acceleration constraints.

Finally, after the plans are merged according to one of these policies, it is necessary to check that the synchronized motion of the arms will not cause collisions between them. For this, the original values of the ACM are restored (so collisions between the arms are considered again), and the presence of self-collisions is evaluated in each trajectory configuration of the dual-arm plan. If any invalid configuration is detected, the plan is rejected and an error message is returned, otherwise, the plan is executed. To be safe, the step (ε) used to generate the individual arm plans must be small, as this is the resolution of the collision checking in the cartesian space. The full process can be seen in Fig. 5.

3.4. Master–slave dual-arm motion

This function uses the dual-arm motion function presented in Section 3.3 and adds the capability to perform identical and constant distance master–slave motions. This means that the user just needs to define the motion of one of the arms, the master, and the motion of the second arm, the slave, will be a function of it. Therefore, even though the temporal synchronization of the arms is achieved using one of the synchronization policies of the previous subsection, additional considerations are required to ensure their spatial synchronization. Fig. 6 shows how this function works.

In the constant distance master–slave motion, the slave arm moves to keep always a constant relative pose with respect to the master arm. This is especially interesting for closed chain bimanual manipulation tasks [56], in which both arms are manipulating the same object. When both grippers are grasping and moving an object, it is very important to keep the relative pose of both end effectors constant to not damage the object. To keep this relative pose constant, the initial homogeneous transformation matrix between the arms is computed (H_S^M), and then, all the slave waypoints ($H_{S_k}^W$) are calculated by applying this same transformation to the master waypoints ($H_{M_k}^W$), as can be seen in Fig. 6. The distance between consecutive waypoints must be small to ensure that the relative pose between arms will be constant during the whole trajectory.

In the identical master–slave motion, the slave arm performs the same cartesian displacements and rotations as the master arm. This is interesting when one arm has to move to a certain pose, but the other arm might be in the middle of its trajectory. With this motion, the slave arm will move away simultaneously, clearing the path for the master arm to move. Another application of this function is when both arms have to perform the exact same relative trajectory. In this case, just the master arm trajectory needs to be planned, and the slave arm will follow it. To achieve this, the same relative movements performed between consecutive master waypoints ($H_{M_k}^{M_{k-1}}$) are applied between consecutive slave waypoints ($H_{S_{k-1}}^W$ and $H_{S_k}^W$), as can be seen in Fig. 6. As in the previous case, the smaller the distance between waypoints, the more similar the arms trajectories will be. Finally, the dual-arm motion function is executed, sending the waypoints of both end effectors, and selecting the synchronization policy of finishing the trajectory both arms at the same time.

4. Experimental evaluation

The results of testing the developed functions with a real robot are presented in this section. These experiments were performed with a Motoman SDA10F dual-arm robot, using a PC with an Intel Core i5-8365U CPU @ 1.60 GHz and 16 GB RAM, and the results have

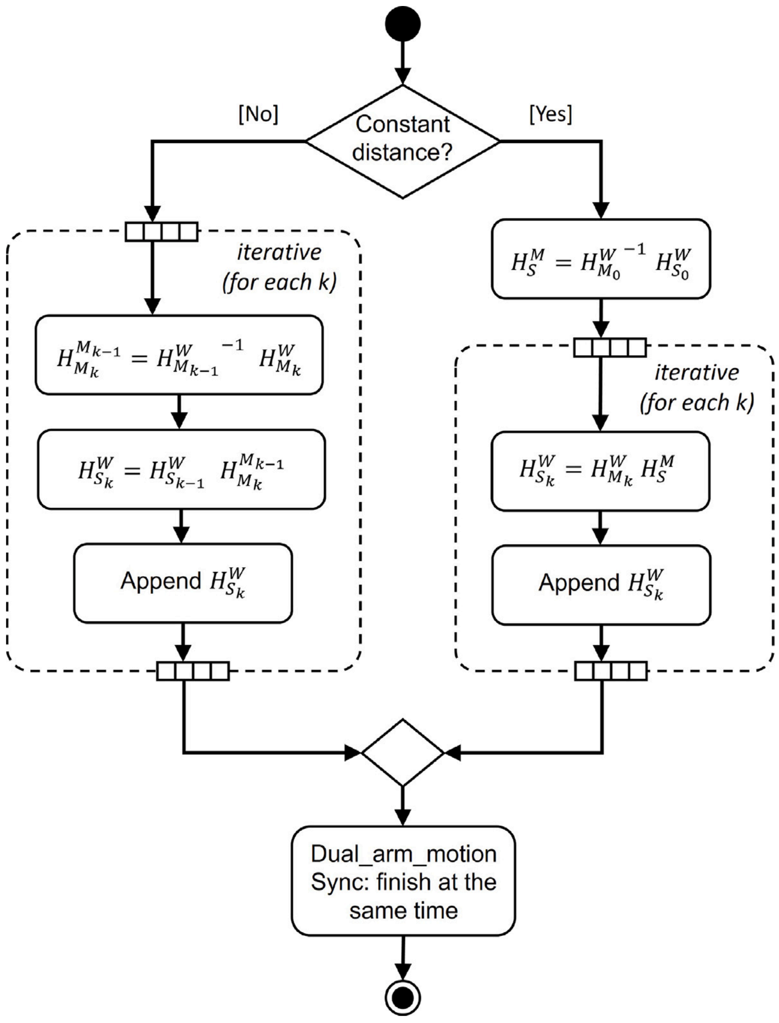

Constant distance?
[No]
[Yes]
iterative (for each k)
$H_{M_k}^{M_{k-1}} = H_{M_{k-1}}^{W\ -1}\ H_{M_k}^{W}$
$H_{S_k}^{W} = H_{S_{k-1}}^{W}\ H_{M_k}^{M_{k-1}}$
Append $H_{S_k}^{W}$
$H_S^M = H_{M_0}^{W\ -1}\ H_{S_0}^{W}$
iterative (for each k)
$H_{S_k}^{W} = H_{M_k}^{W}\ H_S^M$
Append $H_{S_k}^{W}$
Dual_arm_motion Sync: finish at the same time

Fig. 6. UML activity diagram of the dual-arm master–slave motion function. H = homogeneous transformation matrix from the frame of its super index to the frame of its subindex, M = Master, S = Slave, W = World$k$ = index of the waypoint.

been recorded and compiled in a video[2] All the EEF's position and velocity graphs of this section were generated by the data collected when running the tests in the real robot. These values were obtained by sampling the poses of the EEFs using the *get_current_pose* method of the *MoveGroupCommander* class. This section uses the nomenclature introduced in Section 3.

4.1. Automatic tool changing tests

The performed tests consisted of automatically changing one of the robot tools. Three tools were used for these experiments, two WSG-50 parallel grippers and an automatic taping gun, and all the tool change combinations were tested (i.e., change gripper to taping gun and vice versa for both arms). A pneumatic automatic tool changer was used to hook and release the tools, and the pneumatic line was controlled through the robot I/Os, whose values were changed using ROS services. The changes of tool were performed using a tool changing platform with slots specifically designed for the dimensions of each tool, as can be seen in Fig. 7. After calibrating the position of the tool changing platform, all the possible changes of tool for both arms were performed successfully. Fig. 7 shows the robot inserting the gripper in its correspondent slot, before releasing it.

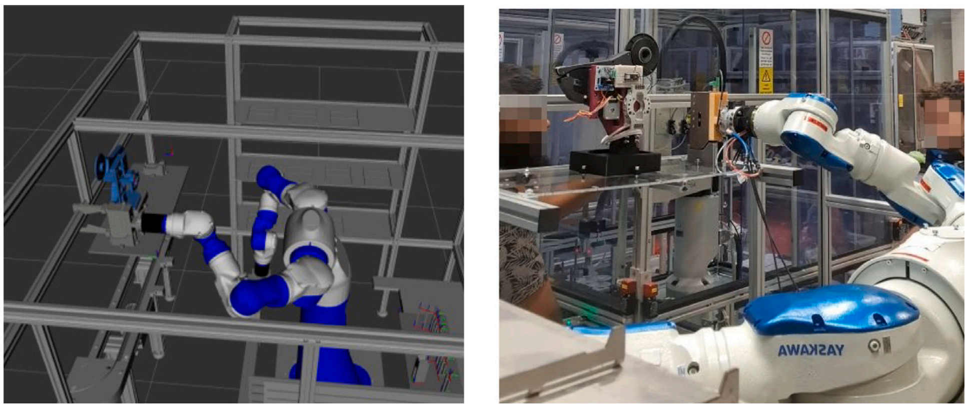
[2] https://youtu.be/DJXKWWmBquM

4.2. End effector speed control tests

The performance of the function to plan end effector trajectories with speed control was tested with eight experiments: A trajectory composed of different target speed sections (E1), a trajectory with slow acceleration able to reach the target velocity (E2), a trajectory with slow acceleration not able to reach the target velocity (E3), a circular trajectory (E4), a trajectory with linear and angular motion (E5-6), and a trajectory that exceeds the joint velocity limits (E7-8). All these experiments are described in Table 2. This table shows also the performance of each experiment with three metrics: the deviation between the target intermediate/final EEF poses and the ones obtained in the experiment (**test poses**), the average EEF speed (**test speed AVG**), and the maximum EEF speed error (**test speed EM**). The last two metrics are calculated without considering the acceleration/deceleration ramps. Additionally, the results of the experiments can be seen in Fig. 8 (E1-4), Fig. 9 (E5-6) and Fig. 10 (E7-8).

In the first experiment, a trajectory with a complex EEF speed profile was planned. This trajectory has three sections, all of them linear motions in the X axis. In the first one the EEF moves 150 mm with a target speed of 70 mm/s (E1.1), in the second 100 mm at 30 mm/s (E1.2), and in the third 150 mm at 70 mm/s (E1.3). As can be seen in Fig. 8, the EEF reaches the target speeds in each section, and the speed transitions happen within the sections with the highest target velocity.

In the second and third experiments, a trajectory with low acceleration is planned. In both cases, the motion is linear along the X



(a) RVIZ visualization (b) Real robot

Fig. 7. Insertion of the WSG-50 gripper in its slot in the tool changing platform during one of the ATC tests.

Table 2
EEF cartesian velocity control experiments.

Exp	Path type	Start (mm) (deg)*	Path definition (mm)(deg)*	Target speed (mm/s) (deg/s)*	Max accel (mm/s^2) (deg/s^2)*	Test poses (mm) (deg)*	Test speed AVG/EM (mm/s) (deg/s)*	Fig
E1	Linear X motion	X_0 = −130.3	X_f = −280.3	70	200	ΔX_f = −9.8	69.7/3.7	8
			X_f = −380.3	30	200	ΔX_f = 5.6	29.5/1.5	
			X_f = −530.3	70	200	ΔX_f = 0	68.4/−3.5	
E2	Linear X motion	X_0 = −530.3	X_f = −130.3	40	20	ΔX_f = 0	39.4/2.2	
E3	Linear X motion	X_0 = −130.3	X_f = −380.3	70	E5	ΔX_f = −0.1	V_s not reached	
E4	Circular XY motion	X_0 = −380.3 Y_0 = 506.8	X_f = −130.3 Y_f = 506.8 R = 125 θ^*_{Arc} = 180	80	200	ΔX_f = −0.1 ΔY_f = 0	79.0/−4.3	
E5	Simultaneous linear X motion + Z$_G$ rotation	X_0 = −130.3 $\theta^*_{Z_0}$ = 1.7	X_f = −480.3 $\theta^*_{Z_f}$ = 176.7	50 200*	200 140*	ΔX_f = −1.9$\Delta\theta^*_{Z_f}$ = 0.6	49.6/3.1 (24.5/-175)*†	9
E6	Simultaneous linear X motion + Z$_G$ rotation	X_0 = −130.3 $\theta^*_{Z_0}$ = 1.7	X_f = −480.3 $\theta^*_{Z_f}$ = 176.7	50 10*	200 140*	ΔX_f = −1.9 $\Delta\theta^*_{Z_f}$ = 0.7	(19.5/−32.2)† (9.6/−0.8)*	
E7	Linear X motion	X_0 = −130.4	X_f = −580.4	1000 φ = 10%	2000	ΔX_f = −0.1	(141/-886)†	10
E8	Linear X motion	X_0 = −580.4	X_f = −130.4	1000 φ = 5%	2000	ΔX_f = 0	(71.2/-937)†	

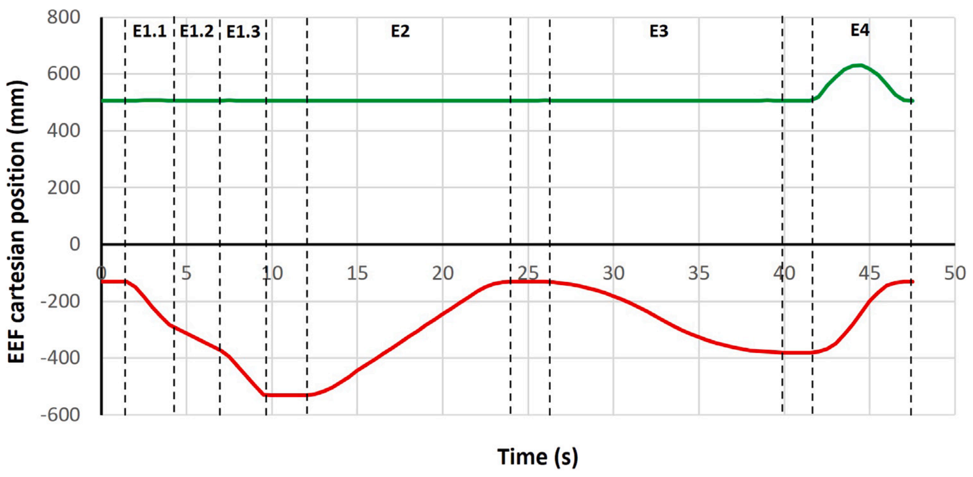
f: final, Z$_G$: Z axis of the gripper, R: Radius, †: High target velocity error because it is not reached (due to slow acceleration, joint speed limits exceeded, or linear-angular velocity control).

axis. In the second, the EEF moves 400 mm with a target speed of 40 mm/s and the acceleration limited to 20 mm/s^2. As can be seen in Fig. 8, the acceleration is slower than in the first scenario, and it takes more time to reach the target speed. However, in the third experiment the distance traveled by the EEF is reduced to 250 mm, the target speed increased to 70 mm/s, and the acceleration reduced to 5 mm/s^2, making it impossible to reach the target speed and then decelerate before the end of the trajectory. Therefore, the speed of the EEF starts reducing when it is slightly over 30 mm/s in order to stop its motion in the last waypoint of the trajectory.

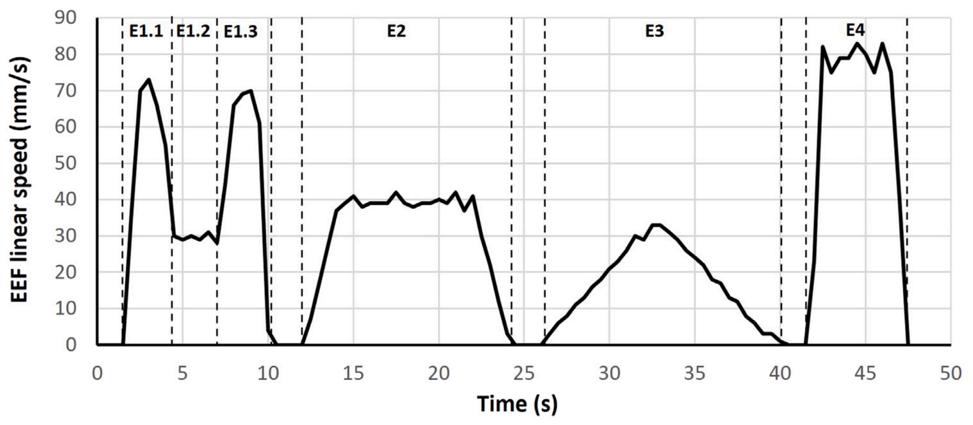
In the fourth experiment a simple velocity profile of 80 mm/s is tested on a circular XY trajectory, to prove that the function not only works for linear motions.

In the fifth and sixth experiments, an EEF trajectory composed of a simultaneous linear (along the X axis) and an angular (around the Z axis of the EEF) motion was tested. First (E5), the trajectory was executed with a high target angular speed (200 deg/s) and a target linear speed of 50 mm/s, and next (E6), with a very slow target angular speed (10 deg/s) and the same target linear speed as before. As the path of the trajectory is given, in most cases it is impossible to move at both target speeds. This issue is solved by ensuring that none of the target speeds are exceeded, fixing always the one that is reached first, as can be seen in Fig. 9. In the first case (E5), the target angular speed is too high, so it cannot be reached without exceeding the linear one. Therefore, the linear speed commands the motion and the angular speed varies below its target. However, in the second case (E6), as the target angular speed is low, it is reached before the target linear speed. Therefore, the angular speed is the one commanding the motion in this case and the linear varies below its target. Due to this, the speed oscillations are higher for the velocity that is not commanding the motion.

Finally, in the last two experiments (E7-8), a motion with a high target speed (1000 mm/s) and maximum acceleration (2000 mm/s^2) was performed to check if the function was keeping all the joint speed limits below their limits. Additionally, the joint speed limits were

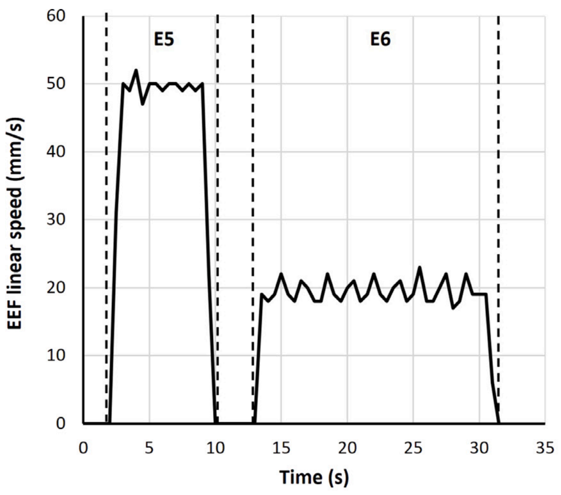


(a) EEF X (red) and Y (green) position

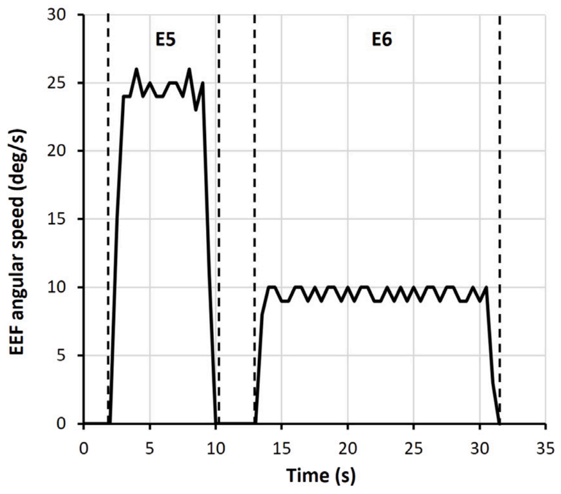


(b) EEF velocity profile

Fig. 8. Cartesian speed control results. Experiments 1-4.



(a) EEF linear velocity profile



(b) EEF angular velocity profile

Fig. 9. Cartesian speed control results. Experiments 5-6: linear + angular motion. In E5, the motion is commanded by the linear speed and the angular speed varies below its target, and vice versa in E6.

reduced virtually, first to a 10% (E7) and then to a 5% (E8), using the φ parameter. The specified target speed could not be reached in any of the experiments without exceeding some of the joint velocity limits. Therefore, the joint speeds were kept constant at their virtual limits, making the EEF cartesian speed vary below its target according to the Jacobian (as can be seen in Fig. 10).

All the graphs presented in this section were generated with data obtained during the real robot operation. The small speed oscillations

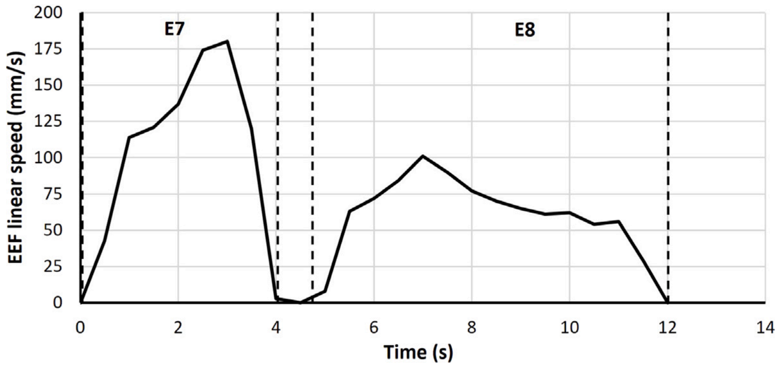


Fig. 10. Cartesian speed control results. Experiments 7-8: joint speed limits exceeded.

are due to the real behavior of the joint actuators, with an average error of 0.32 mm/s and a maximum error of 4.3 mm/s with respect to the target speed, which is much lower when the end effector speed and the acceleration is low. The same experiments were repeated with just the RVIZ visualization of the robot and the average error was reduced to 0.07 mm/s and the maximum error to 1 mm/s. This confirms that the calculated motion plan values are theoretically correct, as the errors are much lower when the effect of the actuators, weights, and inertias is not considered. The oscillations caused by the real actuators' behavior could be minimized using this function with a real-time control solution.

4.3. Dual-arm cartesian motion tests

Four different experiments were performed to test the dual-arm cartesian motion functions (E9-12). E9 and E10 test the three synchronization policies presented in Section 3.3, and they are described in Table 3. Whereas E11 and E12 test the master–slave motions presented in Section 3.4, which are described in Table 4. These tables show also the performance of the experiments using the three metrics presented Section 4.2. Additionally, two more metrics are used in Table 4 to measure the performance of the master–slave motions: the average variation of the relative pose between EEFs ($\boldsymbol{\Delta RP}$ **AVG**), and its maximum error ($\boldsymbol{\Delta RP}$ **EM**).

In E9, the same dual-arm trajectory was planned and executed both with synchronization policy 1 (E9.1) and 2 (E9.2). The results of this test can be seen in Fig. 11(a, c). Initially, synchronization policy 1 was used, setting the maximum EEF speed to 30 mm/s. As the distance traveled by the right arm is shorter, it moves slower than 30 mm/s, while the left arm moves at 30 mm/s, and both arms finish their trajectory at the same time. Then, the robot arms are moved back to their initial positions (without speed control), and the same trajectory is planned using synchronization policy 2. In this case, the right arm velocity is set to 50 mm/s and the left arm velocity to 30 mm/s. As can be seen in the figure, now each arm moves at its target speed and the right arm finishes its motion before the left one. Finally, the robot arms are moved back to their initial positions again (without speed control).

In E10, a trajectory is planned and executed both with synchronization policy 1 (E10.1-E10.2) and 3 (E10.3). This trajectory is defined by an intermediate and a final pair of waypoints, and each of them must be reached by the two EEFs at the same time. In the first part of the trajectory (from the initial to the intermediate pair of waypoints), the left EEF travels less distance, so it must move slower than the right EEF, while in the second part (from the intermediate to the final pair of waypoints), the right EEF is the one with the shortest path, and therefore, the one that must move slower. Regarding the target EEF velocity, it is set to 50 mm/s for both arms and synchronization policies.

When policy 1 is applied, the two parts of the trajectory are planned and executed separately to ensure that the two EEFs will reach the intermediate pair of waypoints at the same time. Due to this, the motion stops between the two trajectory parts. However, when policy 3 is applied, all the trajectory is planned at once, synchronizing the arms to reach every pair of waypoints simultaneously. Additionally, the speed changes between the different trajectory parts are smooth, and the motion never stops. In both cases, the target velocity is just reached by the arm with the largest path between waypoints, while the other arm has to adjust its speed to reach the next waypoint in the same amount of time. As in the previous experiment, movements without speed control are performed to bring the arms back to their initial positions after each experiment. The results of this experiment can be seen in Fig. 11(b, d).

The constant distance master–slave motion was tested in E11 with a set of trajectories executed while grasping a rigid object with both grippers. These trajectories included linear motions (E11.1, E11.5), circular motions (E11.2) (Fig. 12.a), and end effector rotations (E11.3. E11.4, E11.6) (Fig. 12.b), with different target velocities (Fig. 13.e, f). The euclidean distance between the grippers was kept constant at 250 mm, with a maximum variation of 3 mm (Fig. 13.a), and the maximum variation of their relative orientation was 0.7 degrees (Fig. 13.b).

Finally, to test the identical master–slave motion, a trajectory of the master arm that passes through the initial position of the slave arm was performed (E12). The trajectory was executed successfully, the slave arm copied the master's motions moving away and letting space for the master to move without collisions. This trajectory included a linear motion and EEF rotation, as can be seen in Fig. 12.c and Fig. 14.

5. Conclusions

MoveIt is the primary software library for motion planning and mobile manipulation in ROS, however, it is still quite recent, and some important functions have not been developed yet. Therefore, in this paper, the current state of MoveIt has been analyzed in order to identify these gaps. The three main necessities that were detected are: a generic solution to implement an ATC, a function to control the linear and angular cartesian speed of the end effector, and a solution to perform dual-arm synchronized motions. This paper presents a solution for each of these problems, which are publicly available at: https://github.com/pablomalvido/Advanced_manipulation_moveit.

All the developed functions and classes were tested both in RVIZ and with a real robot (a Motoman SDA10F dual-arm robot), showing satisfactory results. The ATC was performed successfully with both arms of the robot, using two parallel grippers and an automatic taping gun. All the possible tool changes were performed (i.e., change gripper to taping gun and vice versa for both arms) just by specifying the name of the tools to change.

The end effector speed control was tested in different scenarios and with different kind of trajectories, showing good control over the linear and angular speeds and accelerations of the end effector for linear and non-linear motions. The obtained velocity profiles showed a very accurate average speed, with an error under 1 mm/s, but with certain oscillations due to the behavior of the real joint actuators.

Table 3
Synchronized dual-arm motion experiments. The acceleration is 200 mm/s^2 for all the experiments.

Exp	Traj	Sync Policy	Arm	Path type	Start (mm)	Path definition (mm)	Target speed (mm/s)	Test poses (mm)(s)*	Test speed AVG/EM (mm/s)	Fig
E9	9.1	1	L	XY linear motion	$X_0 = -130.3$ $Y_0 = 506.9$	$X_f = -430.3$ $Y_f = 706.9$	30	$\Delta X_f = 0$ $\Delta Y_f = 0$ $t_f^* = 17.1$	29.5/−1.5	11(a, c)
			R	XY linear motion	$X_0 = 119.7$ $Y_0 = 506.9$	$X_f = 319.7$ $Y_f = 656.9$	30	$\Delta X_f = 0$ $\Delta Y_f = 0$ $t_f^* = 17.1$	(20.9/−10.1)†	
	9.2	2	L	XY linear motion	$X_0 = -130.3$ $Y_0 = 506.9$	$X_f = -430.3$ $Y_f = 706.9$	30	$\Delta X_f = 0.1$ $\Delta Y_f = 0$ $t_f^* = 38.2$	30.0/−1.5	
			R	XY linear motion	$X_0 = 119.7$ $Y_0 = 506.9$	$X_f = 319.7$ $Y_f = 656.9$	50	$\Delta X_f = 0$ $\Delta Y_f = 0$ $t_f^* = 31.5$	49.8/2.5	
E10	10.1	1	L	XY linear motion	$X_0 = -130.3$ $Y_0 = 506.9$	$X_f = -230.3$ $Y_f = 606.9$	50	$\Delta X_f = 0.1$ $\Delta Y_f = 0$ $t_f^* = 11.0$	(22.1/−28.7)†	11(b, d)
			R	XY linear motion	$X_0 = 119.6$ $Y_0 = 506.9$	$X_f = 369.7$ $Y_f = 706.9$	50	$\Delta X_f = 0$ $\Delta Y_f = 0$ $t_f^* = 11.0$	49.9/2.9	
	10.2	1	L	XY linear motion	$X_0 = -230.3$ $Y_0 = 606.9$	$X_f = -80.3$ $Y_f = 756.9$	50	$\Delta X_f = -0.1$ $\Delta Y_f = 0$ $t_f^* = 18.6$	50.2/2.4	
			R	XY linear motion	$X_0 = 369.7$ $Y_0 = 706.9$	$X_f = 269.7$ $Y_f = 756.9$	50	$\Delta X_f = 0$ $\Delta Y_f = 0$ $t_f^* = 18.6$	(26.6/−27.9)†	
	10.3	3	L	Sequence of XY linear motions	$X_0 = -130.3$ $Y_0 = 506.9$	$X_1 = -230.3$ $Y_1 = 606.9$ $X_f = -80.3$ $Y_f = 756.9$	50	$\Delta X_1 = 7.3$ $\Delta Y_1 = 5.2$ $t_1^* = 39.2$ $\Delta X_f = 0$ $\Delta Y_f = 0$ $t_f = 43.7$	$(22.0/29.1)_1^{\dagger}$ $(49.8/-2.0)_f$	
			R	Sequence of XY linear motions	$X_0 = 119.6$ $Y_0 = 506.9$	$X_1 = 369.7$ $Y_1 = 706.9$ $X_f = 269.7$ $Y_f = 756.9$	50	$\Delta X_1 = -6.2$ $\Delta Y_1 = 1.0$ $t_1^* = 39.2$ $\Delta X_f = 0$ $\Delta Y_f = 0$ $t_f^* = 43.7$	$(50.0/-4.9)_1$ $(26.2/-24.0)_f^{\dagger}$	

f: final, $[X_1, Y_1]$: Intermediate XY waypoint, $[X_f, Y_f]$: Final XY waypoint, †: High target velocity error for one of the arms because it is not reached (due to the use of the arms synchronization policies 1 and 3).

The developed dual-arm motion functions were tested with different synchronization policies: both arms moving at their own speeds, finishing at the same time, or reaching certain intermediate waypoints at the same time. All these tests were successful, without self-collisions and complying with the synchronization policies. Additionally, the master–slave dual-arm motion was tested. For this, the slave arm motion was calculated to either mimic the master arm motion or to keep the distance between end effectors constant. This last case was tested grasping a rigid object with both grippers. The object was moved without any damage or deformation, with a maximum variation of 3 mm in the distance between the grasping points during the whole trajectory.

After testing, it can be concluded that all the functions presented in this paper are valid solutions to overcome the main gaps identified in MoveIt. Additionally, these functions are generic and robot-agnostic, hence, they can be used to extend the current MoveIt capabilities for any robot and application.

Regarding future work, there are three objectives. The first one is to keep testing the developed functions, using them for different use cases and robots. The second one is to minimize the speed oscillations of the trajectories generated with these functions. As mentioned at the end of Section 4.2, this could be done by using these functions with a real-time control solution. Finally, the third objective is to migrate the developed functions to ROS 2 and MoveIt 2. Moreover, the repository will be maintained, keeping the functions updated with the modifications derived from future work.

Declaration of competing interest

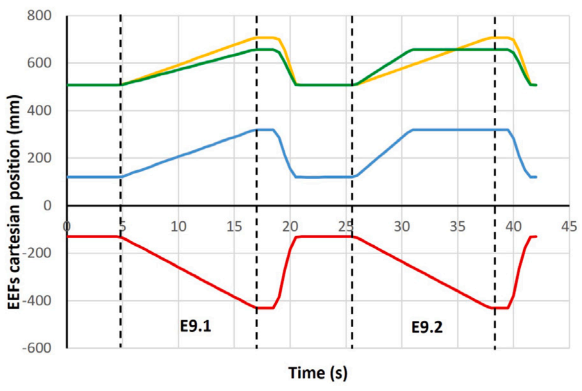
One or more of the authors of this paper have disclosed potential or pertinent conflicts of interest, which may include receipt of payment, either direct or indirect, institutional support, or association with an entity in the biomedical field which may be perceived to have potential conflict of interest with this work. For full disclosure statements refer to https://doi.org/10.1016/j.rcim.2023.102559. Pablo Malvido Fresnillo reports financial support, article publishing charges, and equipment, drugs, or supplies were provided by Horizon 2020. Saigopal Vasudevan reports financial support, article publishing charges, and equipment, drugs, or supplies were provided by Horizon 2020. Jose Luis Martnez Lastra reports financial support, article publishing charges, and equipment, drugs, or supplies were provided by Horizon 2020.

Data availability

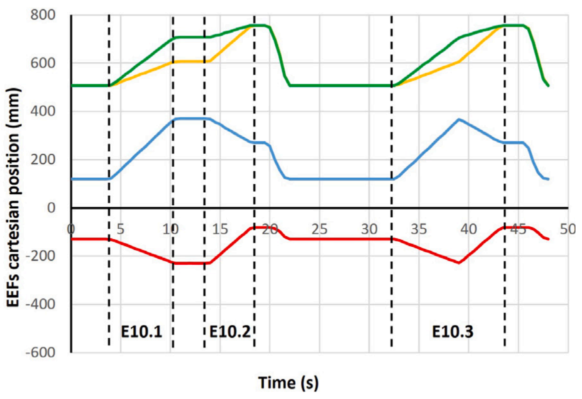
Data will be made available on request.

Acknowledgments

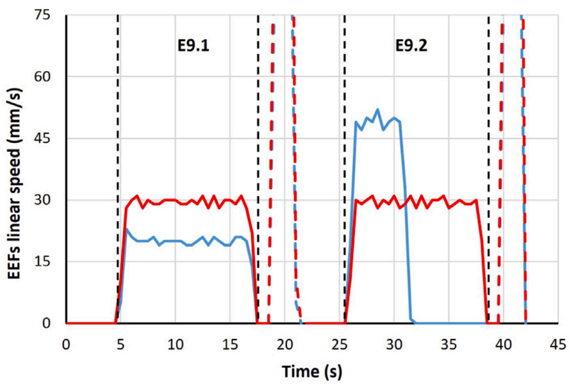
The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement no. 870133, correspondent to the project entitled REMODEL, Robotic tEchnologies for the Manipulation of cOmplex DeformablE Linear objects.



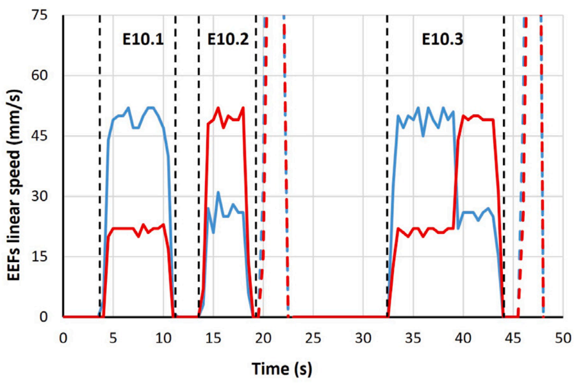
(a) EEFs' X (L: red, R: blue) and Y (L: green, R: yellow) positions



(b) EEFs' X (L: red, R: blue) and Y (L: green, R: yellow) positions



(c) EEFs' linear velocities (L: red, R: blue). The dashed lines represent non-controlled trajectories (out of the analysis)



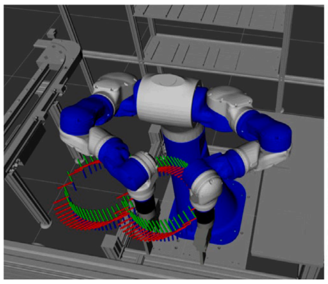
(d) EEFs' linear velocities (L: red, R: blue). The dashed lines represent non-controlled trajectories (out of the analysis)

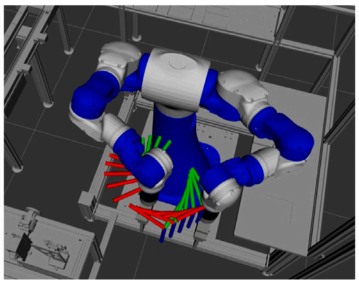
Fig. 11. Dual-arm synchronization results. Experiments 9 (a, c) and 10 (b, d). R: Right, L: Left.

Table 4
Master–slave experiments. The acceleration is 200 mm/s^2 for all the experiments.

Exp	Traj	Sync Policy	Master arm	Path type	Start (mm) (deg)*	Path definition (mm)(deg)*	Target speed (mm/s) (deg/s)*	Test poses (mm) (deg)*	Test ΔRP AVG/EM (mm) (deg)*	Test speed AVG/EM (mm/s) (deg/s)*	Fig
E11	11.1	MS Const RP	L	XY linear motion	$X_0 = -130.3$ $Y_0 = 506.9$	$X_f = -280.3$ $Y_f = 606.9$	40	$\Delta X_f = 0.1$ $\Delta Y_f = -0.1$	−0.9/−1.6	40.0/1.9	13
	11.2	MS Const RP	L	XY circular motion	$X_0 = -280.3$ $Y_0 = 606.9$	$X_f = -280.3$ $Y_f = 606.9$ $R = 150$ $\theta^*_{Arc} = 360$	60	$\Delta X_f = 0.1$ $\Delta Y_f = 0$	−0.1/3.0	59.9/−3.8	
	11.3	MS Const RP	L	Linear XY + Z$_G$ rot	$X_0 = -280.3$ $Y_0 = 606.9$ $\theta^*_{Z_0} = 1.7$	$X_f = -130.3$ $Y_f = 756.9$ $\theta^*_{Z_f} = 61.7$	30 21*	$\Delta X_f = 0$ $\Delta Y_f = -0.1$ $\Delta\theta^*_{Z_f} = 0$	0.3/1.0 (−0.2/−0.4)*	30.1/1.9 (8.5/−11.6)*†	
	11.4	MS Const RP	L	Linear XY + Z$_G$ rot	$X_0 = -130.3$ $Y_0 = 756.9$ $\theta^*_{Z_0} = 61.7$	$X_f = -280.3$ $Y_f = 606.9$ $\theta^*_{Z_f} = 1.7$	30 -21*	$\Delta X_f = 0.1$ $\Delta Y_f = 0$ $\Delta\theta^*_{Z_f} = -0.1$	−0.3/−1.1 (0.2/0.4)*	29.9/1.9 (−8.5/11.3)*†	
	11.5	MS Const RP	L	XY linear motion	$X_0 = -280.3$ $Y_0 = 606.9$	$X_f = -130.3$ $Y_f = 506.9$	40	$\Delta X_f = -0.1$ $\Delta Y_f = 0$	0.4/1.5	39.9/2.3	
	11.6	MS Const RP	L	Linear YZ + X$_G$ rot	$Y_0 = 506.9$ $Z_0 = 1197.2$ $\theta^*_{X_0} = -91.7$	$Y_f = 641.9$ $Z_f = 1332.3$ $\theta^*_{X_f} = -181.7$	30 -21*	$\Delta Y_f = -0.1$ $\Delta Z_f = -0.2$ $\Delta\theta^*_{X_f} = 0.2$	−0.1/−0.2 (0.3/0.7)*	29.0/−2.8 (−14.4/1.4)*†	
E12	12.1	MS Id	L	Linear X + Z$_G$ rot	$X_0 = -130.3$ $\theta^*_{Z_0} = 1.7$	$X_f = 69.7$ $\theta^*_{Z_f} = 89.7$	30 21*	$\Delta X_f = -0.5$ $\theta^*_{Z_f} = -0.4$	–	30.2/1.8 (13.0/−7.1)*†	14

f: final, Z$_G$: Z axis of the gripper, R: Radius, RP: Relative pose between EEFs, MS Const RP: Constant relative pose Master–Slave motion, MS Id: Identical Master–Slave motion
†: High target velocity error because it is not reached (due to the linear-angular speed control).


(a) Circular motion (E11.2)


(b) EEF rotation (E11.3)

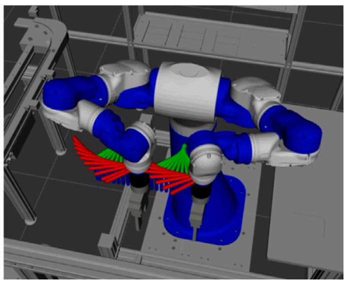
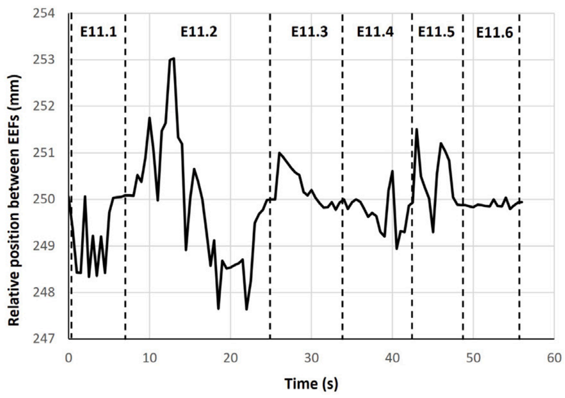
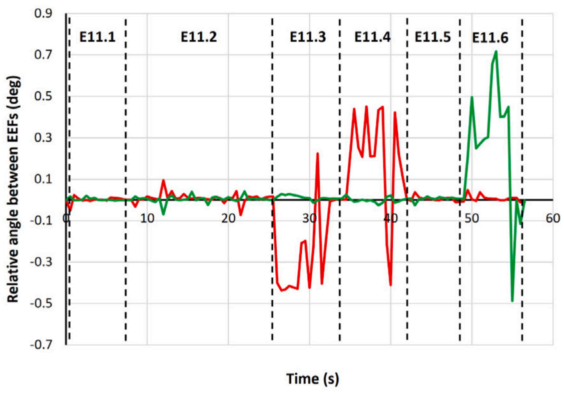

(c) Identical master-slave (E12)

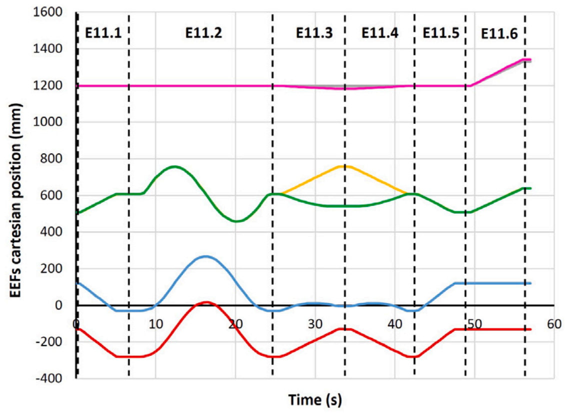
Fig. 12. RVIZ visualization of generated master–slave paths.



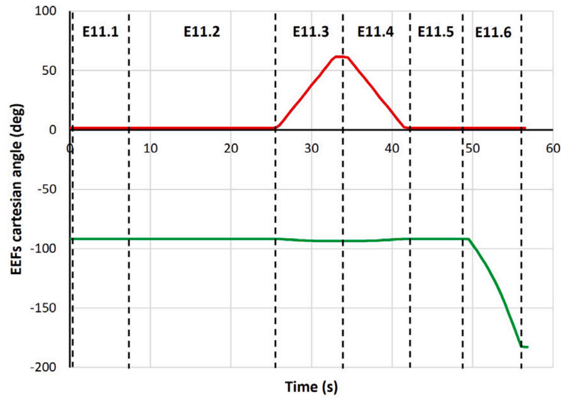
(a) Relative position between EEFs



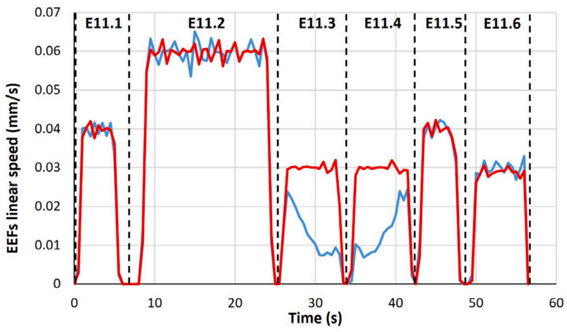
(b) Relative angle between EEFs



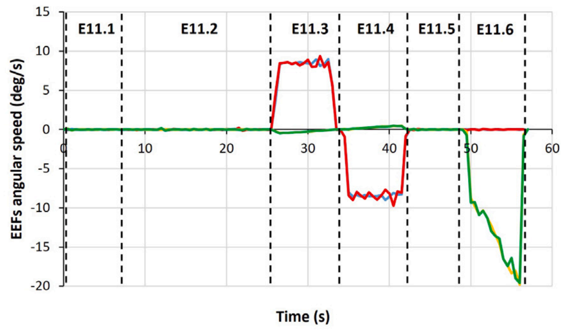
(c) EEFs' X (L: red, R: blue), Y (L: green, R: yellow), and Z (L: pink, R: gray) positions.



(d) EEFs' angles around their Z (L: red, R: blue) and X (L: green, R: yellow) axis.

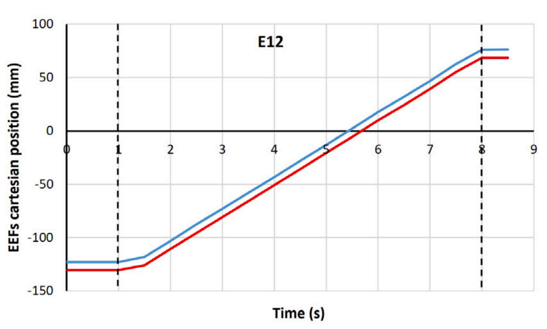


(e) EEFs' linear velocities (L: red, R: blue)

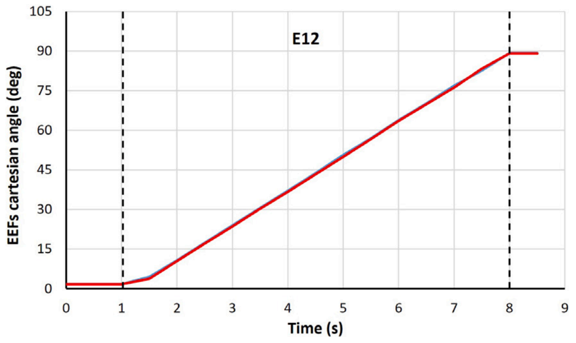


(f) EEFs' angular velocities around their Z (L: red, R: blue) and X (L: green, R: yellow) axis

Fig. 13. Constant relative pose Master–Slave motion results. Experiment 11. R: Right, L: Left. The left arm is the master.



(a) EEFs' X (L: red, R: blue) positions.



(b) EEFs' angles around their Z (L: red, R: blue) axis.

Fig. 14. Identical Master–Slave motion results. Experiment 12. R: Right, L: Left. The left arm is the master.

References

[1] B. Singh, N. Sellappan, P. Kumaradhas, Evolution of industrial robots and their applications, Int. J. Emerg. Technol. Adv. Eng. 3 (2013) 763–768.

[2] J. Wallén, The History of the Industrial Robot, Linköping University Electronic Press, 2008.

[3] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, A. Ng, ROS: an open-source Robot Operating System, in: ICRA Workshop on Open Source Software, Vol. 3, No. 3.2, Kobe, Japan, 2009.

[4] L. Joseph, J. Cacace, Mastering ROS for Robotics Programming: Design, Build, and Simulate Complex Robots using the Robot Operating System, second ed., Packt Publishing Ltd, 2018.

[5] ROS Industrial, ROS-Industrial description, 2022, Available at: https://rosindustrial.org/about/description. (Accessed 11 December 2022).

[6] ROSIN, 2022. Available at: https://www.rosin-project.eu/. (Accessed 11 December 2022).

[7] S.H. Tang, W. Khaksar, N.B. Ismail, M.K.A. Ariffin, A review on robot motion planning approaches, Pertanika J. Sci. Technol. 20 (1) (2012) 15–29.

[8] S. Chitta, I. Sucan, S. Cousins, Moveit![ROS topics], IEEE Robot. Autom. Mag. 19 (2012) 18–19, http://dx.doi.org/10.1109/MRA.2011.2181749.

[9] S. Chitta, Moveit!: An introduction, in: A. Koubaa (Ed.), Robot Operating System (ROS): The Complete Reference (Volume 1), in: Studies in Computational Intelligence, Springer International Publishing, Cham, 2016, pp. 3–27, http://dx.doi.org/10.1007/978-3-319-26054-9_1.

[10] J. Badger, D. Gooding, K. Ensley, K. Hambuchen, A. Thackston, ROS in space: A case study on robonaut 2, in: A. Koubaa (Ed.), Robot Operating System (ROS): the Complete Reference (Volume 1), in: Studies in Computational Intelligence, Springer International Publishing, Cham, 2016, pp. 343–373, http://dx.doi.org/10.1007/978-3-319-26054-9_13.

[11] MoveIt, MoveIt robots, 2022, Available at: https://moveit.ros.org/robots/. (Accessed 8 December 2022).

[12] L. Lu, J. Zhang, J.Y.H. Fuh, J. Han, H. Wang, Time-optimal tool motion planning with tool-tip kinematic constraints for robotic machining of sculptured surfaces, Robot. Comput.-Integr. Manuf. 65 (2020) 101969, http://dx.doi.org/10.1016/j.rcim.2020.101969.

[13] Y. Zhang, L. Li, M. Ripperger, J. Nicho, M. Veeraraghavan, A. Fumagalli, Gilbreth: A conveyor-belt based pick-and-sort industrial robotics application, in: 2018 Second IEEE International Conference on Robotic Computing, IRC, 2018, pp. 17–24, http://dx.doi.org/10.1109/IRC.2018.00012.

[14] R. Rahimi, C. Shao, M. Veeraraghavan, A. Fumagalli, J. Nicho, J. Meyer, S. Edwards, C. Flannigan, P. Evans, An industrial robotics application with cloud computing and high-speed networking, in: 2017 First IEEE International Conference on Robotic Computing, IRC, 2017, pp. 44–51, http://dx.doi.org/10.1109/IRC.2017.39.

[15] D. Coleman, I. Sucan, S. Chitta, N. Correll, Reducing the barrier to entry of complex robotic software: a MoveIt! case study, J. Softw. Eng. Robot. (2014) http://dx.doi.org/10.48550/arXiv.1404.3785.

[16] S. Hernandez-Mendez, C. Maldonado-Mendez, A. Marin-Hernandez, H.V. Rios-Figueroa, H. Vazquez-Leal, E.R. Palacios-Hernandez, Design and implementation of a robotic arm using ROS and MoveIt!, in: 2017 IEEE International Autumn Meeting on Power, Electronics and Computing, ROPEC, 2017, pp. 1–6, http://dx.doi.org/10.1109/ROPEC.2017.8261666, ISSN: 2573-0770.

[17] Z. Kingston, L. Kavraki, Robowflex: Robot motion planning with MoveIt made easy, in: 2022 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS, 2022.

[18] Moveit commander, Moveit_commander/MoveGroupCommander, 2022, Available at: http://docs.ros.org/en/jade/api/moveit_commander/html/namespacemoveit_commander.html. (Accessed 8 December 2022).

[19] A. Sharp, K. Kruusamäe, B. Ebersole, M. Pryor, Semiautonomous dual-arm mobile manipulator system with intuitive supervisory user interfaces, in: 2017 IEEE Workshop on Advanced Robotics and Its Social Impacts, ARSO, (ISSN: 2162-7576) 2017, pp. 1–6, http://dx.doi.org/10.1109/ARSO.2017.8025195.

[20] MoveIt msgs, Moveit_msgs/RobotTrajectory, 2022, Available at: http://docs.ros.org/en/noetic/api/moveit_msgs/html/msg/RobotTrajectory.html. (Accessed 8 December 2022).

[21] S. Chitta, E. Marder-Eppstein, W. Meeussen, V. Pradeep, A. Rodríguez Tsouroukdissian, J. Bohren, D. Coleman, B. Magyar, G. Raiola, M. Lüdtke, E. Fernandez Perdomo, Ros_control: A generic and simple control framework for ROS, J. Open Source Softw. 2 (20) (2017) 456, http://dx.doi.org/10.21105/joss.00456.

[22] E. Ferrentino, F. Salvioli, P. Chiacchio, Globally optimal redundancy resolution with dynamic programming for robot planning: A ROS implementation, Robotics 10 (1) (2021) http://dx.doi.org/10.3390/robotics10010042, Publisher: Multidisciplinary Digital Publishing Institute.

[23] D.C. Conner, D. Catherman, S. Enders, J. Gates, J. Gu, Flexible manipulation: Finite state machine-based collaborative manipulation, in: SoutheastCon 2018, (ISSN: 1558-058X) 2018, pp. 1–8, http://dx.doi.org/10.1109/SECON.2018.8478933.

[24] R. Little, Tool changers enhance robot versatility, Ind. Robot Int. J. 30 (4) (2003) 306–309, http://dx.doi.org/10.1108/01439910310479540, Publisher: MCB UP Ltd.

[25] D. Mourtzis, J. Angelopoulos, M. Papadokostakis, N. Panopoulos, Design for 3D printing of a robotic arm tool changer under the framework of Industry 5.0, Procedia CIRP 115 (2022) 178–183, http://dx.doi.org/10.1016/j.procir.2022.10.070.

[26] J. Atkinson, J. Hartmann, S. Jones, P. Gleeson, Robotic drilling system for 737 aileron, SAE Technical Paper 1 (8), 2007.

[27] B.-S. Ryuh, S.M. Park, G.R. Pennock, An automatic tool changer and integrated software for a robotic die polishing station, Mech. Mach. Theory 41 (4) (2006) 415–432, http://dx.doi.org/10.1016/j.mechmachtheory.2005.06.004.

[28] Business Research Insights, Robot tool changers market size, share, growth, and industry growth by type, by application, and regional forecast to 2028, 2022, Available at: https://www.businessresearchinsights.com/market-reports/robot-tool-changers-market-100781. (Accessed 20 December 2022).

[29] V. Tereshchuk, N. Bykov, S. Pedigo, S. Devasia, A.G. Banerjee, A scheduling method for multi-robot assembly of aircraft structures with soft task precedence constraints, Robot. Comput.-Integr. Manuf. 71 (2021) 102154, http://dx.doi.org/10.1016/j.rcim.2021.102154.

[30] G. Rosati, S. Minto, F. Oscari, Design and construction of a variable-aperture gripper for flexible automated assembly, Robot. Comput.-Integr. Manuf. 48 (2017) 157–166, http://dx.doi.org/10.1016/j.rcim.2017.03.010.

[31] R. Bormann, J. Hampp, M. Hägele, New brooms sweep clean - an autonomous robotic cleaning assistant for professional office cleaning, in: 2015 IEEE International Conference on Robotics and Automation, ICRA, (ISSN: 1050-4729) 2015, pp. 4470–4477, http://dx.doi.org/10.1109/ICRA.2015.7139818.

[32] S. Koukas, N. Kousi, S. Aivaliotis, G. Michalos, R. Bröchler, S. Makris, ODIN architecture enabling reconfigurable human – robot based production lines, Procedia CIRP 107 (2022) 1403–1408, http://dx.doi.org/10.1016/j.procir.2022.05.165.

[33] M. Ericsson, P. Nylén, A look at the optimization of robot welding speed based on process modeling, Weld. J. (Miami, Fla) 86 (2007) 238s–244s.

[34] Y. Bouteraa, J. Ghommam, Synchronization control of multiple robots manipulators, in: 2009 6th International Multi-Conference on Systems, Signals and Devices, 2009, pp. 1–6, http://dx.doi.org/10.1109/SSD.2009.4956742.

[35] S. Macenski, An Open Source Tele-Operation Application for Arbitrary N-DOF Manipulators | IDEALS (Undergraduate thesis), University of Illinois, 2017.

[36] X. Jing, Y. Xue, Y. Chen, A new approach to developing general manipulator control system application based on ROS, in: Y. Jia, J. Du, W. Zhang (Eds.), Proceedings of 2019 Chinese Intelligent Systems Conference, in: Lecture Notes in Electrical Engineering, Springer, Singapore, 2020, pp. 151–158, http://dx.doi.org/10.1007/978-981-32-9698-5_18.

[37] K. Miao, Z. Zhou, Y. Zhang, Trajectory planning and simulation of educational robot based on ROS, in: 2019 2nd International Conference of Intelligent Robotic and Control Engineering, IRCE, 2019, pp. 18–22, http://dx.doi.org/10.1109/IRCE.2019.00011.

[38] G. Gorjup, L. Gerez, G. Gao, M. Liarokapis, On the efficiency, usability, and intuitiveness of a wearable, affordable, open-source, generic robot teaching interface, in: 2022 Mediterranean Conference on Control and Automation, 2022.

[39] A. Laurenzi, E.M. Hoffman, L. Muratore, N.G. Tsagarakis, CartesI/O: A ROS based real-time capable cartesian control framework, in: 2019 International Conference on Robotics and Automation, ICRA, (ISSN: 2577-087X) 2019, pp. 591–596, http://dx.doi.org/10.1109/ICRA.2019.8794464.

[40] E. Brzozowska, O. Lima, R. Ventura, A generic optimization based cartesian controller for robotic mobile manipulation, in: 2019 International Conference on Robotics and Automation, ICRA, (ISSN: 2577-087X) 2019, pp. 2054–2060, http://dx.doi.org/10.1109/ICRA.2019.8793635.

[41] A. Montaño, R. Suárez, Coordination of several robots based on temporal synchronization, Robot. Comput.-Integr. Manuf. 42 (2016) 73–85, http://dx.doi.org/10.1016/j.rcim.2016.05.008.

[42] X. Zhao, B. Tao, L. Qian, Y. Yang, H. Ding, Asymmetrical nonlinear impedance control for dual robotic machining of thin-walled workpieces, Robot. Comput.-Integr. Manuf. 63 (2020) 101889, http://dx.doi.org/10.1016/j.rcim.2019.101889.

[43] M. Qasim Imran, Dual-Arm Manipulation in Robotic Waiter Use Case (Master's thesis), Tallinn University of Technology, 2020.

[44] D. Sepúlveda, R. Fernández, E. Navas, P. González-de Santos, M. Armada, ROS framework for perception and dual-arm manipulation in unstructured environments, in: M. F. Silva, J.L. Lima, L.P. Reis, A. Sanfeliu, D. Tardioli (Eds.), Robot 2019: Fourth Iberian Robotics Conference. Advances in Robotics, Volume 2, in: Advances in Intelligent Systems and Computing, Springer International Publishing, Cham, 2019, pp. 127–147.

[45] A. Suárez-Hernández, G. Alenyà, C. Torras, Interleaving hierarchical task planning and motion constraint testing for dual-arm manipulation, in: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS, (ISSN: 2153-0866) 2018, pp. 4061–4066, http://dx.doi.org/10.1109/IROS.2018.8593847.

[46] J.F. Buhl, R. Grønhøj, J.K. Jørgensen, G. Mateus, D. Pinto, J.K. Sørensen, S. Bøgh, D. Chrysostomou, A dual-arm collaborative robot system for the smart factories of the future, Procedia Manuf. 38 (2019) 333–340, http://dx.doi.org/10.1016/j.promfg.2020.01.043.

[47] C. Gkournelos, N. Kousi, A. Christos Bavelos, S. Aivaliotis, C. Giannoulis, G. Michalos, S. Makris, Model based reconfiguration of flexible production systems, Procedia CIRP 86 (2019) 80–85, http://dx.doi.org/10.1016/j.procir.2020.01.042.

[48] J. Österberg, Skill Imitation Learning on Dual-Arm Robotic Systems (Master's thesis), KTH Royal Institute of Technology, 2020.

[49] P.M. Fresnillo, S. Vasudevan, W.M. Mohammed, An approach for the bimanual manipulation of a deformable linear object using a dual-arm industrial robot: cable routing use case, in: 2022 IEEE 5th International Conference on Industrial Cyber-Physical Systems, ICPS, 2022, pp. 1–8, http://dx.doi.org/10.1109/ICPS51978.2022.9816981.

[50] Y. Solana, H.H. Cueva, A.R. García, S.M. Calvo, U.E. Campos, D. Sallé, J. Cortés, A case study of automated dual-arm manipulation in industrial applications, in: 2019 24th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, (ISSN: 1946-0759) 2019, pp. 563–570, http://dx.doi.org/10.1109/ETFA.2019.8869209.

[51] L. Yan, Y. Yang, W. Xu, S. Vijayakumar, Dual-arm coordinated motion planning and compliance control for capturing moving objects with large momentum, in: 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS, (ISSN: 2153-0866) 2018, pp. 7137–7144, http://dx.doi.org/10.1109/IROS.2018.8593853.

[52] C. Rodríguez, A. Rojas-de Silva, R. Suárez, Dual-arm framework for cooperative applications, in: 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation, ETFA, 2016, pp. 1–6, http://dx.doi.org/10.1109/ETFA.2016.7733704.

[53] D. Lee, C.-W. Ha, Optimization process for polynomial motion profiles to achieve fast movement with low vibration, IEEE Trans. Control Syst. Technol. 28 (5) (2020) 1892–1901, http://dx.doi.org/10.1109/TCST.2020.2998094, Conference Name: IEEE Transactions on Control Systems Technology.

[54] J.R. Garcia Martinez, J. Rodriguez Resendiz, M.A. Martinez Prado, E.E. Cruz Miguel, Assessment of jerk performance s-curve and trapezoidal velocity profiles, in: 2017 XIII International Engineering Congress, CONIIN, 2017, pp. 1–7, http://dx.doi.org/10.1109/CONIIN.2017.7968187.

[55] J. Kim, E.A. Croft, Online near time-optimal trajectory planning for industrial robots, Robot. Comput.-Integr. Manuf. 58 (2019) 158–171, http://dx.doi.org/10.1016/j.rcim.2019.02.009.

[56] C. Smith, Y. Karayiannidis, L. Nalpantidis, X. Gratal, P. Qi, D.V. Dimarogonas, D. Kragic, Dual arm manipulation—A survey, Robot. Auton. Syst. 60 (10) (2012) 1340–1353, http://dx.doi.org/10.1016/j.robot.2012.07.005.