Synchronous Movements Manipulation and Collision Avoidance Planning in a Dual-Robot Environment

Jacopo Mora · Matteo Salvalai

***Abstract*—Multi-robots systems, especially ones with arms close to each other or with lots of objects in the surroundings, suffer from the problem of having to avoid collisions between robots themselves as well as with the environment. In this paper we will be describing the study and implementation of a Collision Avoidance Planner, thought to be applied in an Agricultural setting. The system is capable of handling a set of given poses and of distributing them among a group of UR5 robot arms. Collisions are identified and corrected using the Bidirectional RRT algorithm, derived from the Rapidly-exploring Random Tree and the Bidirectional Search algorithms. The performance of the overall system suggests that our implementation is time-wise not acceptable due to the programming language used and to lackings in the used support libraries.**

I. Introduction

AGRICULTURE, developed around 12.000 years ago and the trigger of the *Neolithic Revolution*, changed fundamentally the society humans lived in. One would think that a practice so dear to us, given the rapid technological advancement of our era, must be easy to automate, and yet progress in this field is slow-paced. To test our capabilities as future engineers and to do our part we decided to take on the challenge. In particular, we focused our attention on *harvesting*: the task of harvesting a single product can be traduced, from an automation standpoint, in a task of Pick-and-Place. In a more general setting, optimizations for this task can take the form of a group of robots, working in a shared environment independently one to another. The object of this paper is to implement and to study a system capable of controlling such a group, capable to ensure that no collisions will ever happen between any of its members. In the *Architecture* section we will present the high-level structure of the overall system while in *Implementation* we'll enter more into details on how the architecture was effectively implemented. The *Rework Function* section presents the algorithms used to find a path between two joints configurations. *Overall Performance* will contain benchmarks and comparisons between the two programming languages for which the support libraries are available, `Python` and `C++`, and comparisons among developed algorithms.

II. Architecture

At a high level of abstraction, we identified three main components, a *Commander*, a *Planner* and a set of *GroupCommander*s, one for each controlled robot.

A. GroupCommander

The *GroupCommander* is a high-level description of an underlying group of joints, which constitute a robot. It allows querying for its current state, which provides information regarding the ability to accept and execute a new trajectory, and for its current pose, expressed as a configuration in the group's joints space.

Moreover, it is capable of computing a trajectory to reach a given pose, the position expressed in Cartesian coordinates while the orientation with quaternions, free of self-collisions and collisions against the surrounding environment. Each pose in the trajectory is expressed as a configuration in the joints space. The environment that the *GroupCommander* is aware of is a static model of the real environment: each object is stored with its state, registered at the moment the planning function is called; information regarding past or future states of a given object is excluded from this representation. The physical limitations of the joints group are kept in mind during the trajectory's generation process. Because of these properties, we'll refer to the resulting trajectory as *naive*.

Finally, it allows the execution of a given trajectory, with each point expressed as a joints configuration, giving feedback the moment the final pose is reached. For later references, the input trajectory will be kept in memory.

B. Planner

The main objective of the *Planner* is to build a collision-free trajectory, given a pose and the *GroupCommander* to which it has been assigned to. The *Planner* has to identify and avoid cases of self-collisions, collisions to other joints groups, and collisions to the ever-changing environment, while keeping track of the physical constraints of the input *GroupCommander* and estimating the amount of time required to produce the resulting trajectory. The total time estimation must be as precise as possible in order to reduce inefficiencies regarding potential pauses taken to wait until the produced trajectory becomes valid. The estimation is also necessary since the planning process is not instantaneous and the result must be valid.

The pseudo-code of the algorithm, called $Plan$, is presented in Figure II-B; as inputs, it requires the set of all *GroupCommander*s, i.e. $groups$, the *GroupCommander* to which the pose is being assigned, $gComm$, the pose itself, $pose$, and a description of the environment, env. The call of $CollisionChecker$ results in the creation of an object

that simulates the behavior of the selected *GroupCommander* in a model of its surrounding environment and of the other *GroupCommander*s. Thanks to this construct it's possible to find occupied portions of the workspace at any given time and, so, to identify collisions.

Once the naive trajectory, $plan$, is composed, the final trajectory, $result$, is initialized to an empty trajectory and a time estimate of the overall duration of the algorithm, $estTime$, is obtained by combining knowledge regarding past runs and properties of the naive trajectory. The function $EstimateReworkTime$ is responsible for the estimation: different strategies can be applied, we decided to adopt a very straightforward one,

$$\hat{t}_T := \lambda \cdot (t_{path} \cdot (|T| - 1) + t_{rework} \cdot N_{avgCollisions})$$

where T is the trajectory for which to estimate the duration, $\hat{t}_T$ is the estimation, t_{path} is the average time taken to check if the path between two points is collision-free, $N_{avgCollision}$ is the average number of collisions found in a single check, t_{rework} is the time needed by $Rework$ to compose a new trajectory between two points and λ is a scaling factor, applied to give robustness to the estimation process.

Checks are then performed to identify collisions in the initial and final point of the trajectory and to return an exception if one is found. The initial point of the naive trajectory is the current configuration of the *GroupCommander*, while the final point is the configuration corresponding to the input goal pose. To avoid collisions in the initial configuration we would have to find a new initial configuration, not in a collision state, compute a trajectory to it and execute it. All of this must be done by being careful not to exceed the time frame previously estimated, which is unlikely to happen. Changes in the final pose would defeat the point of the whole system.

With $SetStartTime$, the trajectory $result$ will be set to start at $estTime$ seconds after the current timestamp, the one registered by the system at the time the function is called. A $timer$ will keep track of the duration of the current run, which will be used to update the historical information with $UpdateEstimate$.

Let's now consider each point of the naive trajectory: a point is suitable for the $result$ trajectory if, and only if, the path between the predecessor of the current point and the current point itself is collision-free. The path is explored with the function $FindCollisions$. In case a collision is found when exploring a given path, the relative end point will be considered in collision and each consecutive point for which the path is in collision is removed from the naive trajectory. The path between the predecessor of the initially considered point and the successor of the last point found in this way can be considered, overall, in a state of collision and so must be rebuilt. The function $Rework$ allows to find a new path between the two considered points and, using $ApplyPhysicalConstraints$, the resulting trajectory will be adapted to the physical abilities of the *GroupCommander* considered. The $Rework$ function will be discussed in detail in the next chapter.

Finally, the correction will be introduced in the $result$ trajectory and the next point will be analyzed.

Fig. 1. *Planner*: planning of the trajectory

```
1:  function Plan(env, groups, gComm, pose)
2:      cc ← CollisionChecker(env, groups, gComm)
3:      plan ← NaivePlan(pose, gComm)
4:      result ← Trajectory()
5:      estTime ← EstimateReworkTime(plan)
6:      if not Executable(plan, estTime) then
7:          return NIL, 0
8:      end if
9:      SetStartTime(result, estTime)
10:     Start(timer)
11:     from ← Pop(plan)
12:     Append(from, result)
13:     while not Empty(plan) do
14:         to ← Pop(plan)
15:         if not FindCollisions(from, to, cc) then
16:             Append(to, result)
17:             from ← to
18:             continue
19:         end if
20:         to ← FirstValidPoint(to, plan, cc)
21:         r ← Rework(from, to, cc)
22:         if Empty(r) then
23:             Reset(timer)
24:             return NIL, 0
25:         end if
26:         r ← ApplyPhysicalConstraints(r)
27:         Extend(result, r)
28:         from ← to
29:     end while
30:     Stop(timer)
31:     UpdateEstimate(timer)
32:     return results, estTime − Duration(timer)
33: end function
```

When every point has been checked and every path in collision has been corrected, the resulting trajectory will be returned, together with the amount of time that remains to wait until the trajectory can be considered executable. The *Planner* has the responsibility to keep track of the overall planning duration, the average number of collisions found in a given naive trajectory, and the average duration of a rework.

C. Commander

The main responsibility of the *Commander* is to handle the interactions between the *GroupCommander*s and the *Planner*, acting as a front-end for the two other components. Poses can be submitted to the *Commander*, along with a priority and an indication regarding which group should satisfy it. They are stored in a dedicated queue, ordered by priority, waiting to be handled. The *Commander* allows clearing the waiting poses queue.

Moreover, in an effort to be more transparent, the *Commander* has to publish periodically its current status and the related *GroupCommander*s status, along with the queue and the current working pose information for each arm.

Fig. 2. *Commander*: dispatch a given pose

```
1: function Dispatch(env, groups, pose)
2:     if not CanHandle(pose) then
3:         return False
4:     end if
5:     G ← SelectGroup(pose)
6:     T, toWait ← Plan(env, groups, G, pose)
7:     if not isValid(T) then
8:         DecrementRetry(pose)
9:         Put(pose, queue)
10:    else if toWait < 0 then
11:        DecrementRetry(pose)
12:        Put(pose, queue)
13:    else
14:        Wait(toWait)
15:        Execute(T, G)
16:    end if
17:    return True
18: end function
```

Fig. 3. RRT

```
1: function RRT(start, end, checker)
2:     nodes ← [start]
3:     i ← 0
4:     while i < I do
5:         rnd ← RandomNode()
6:         nearest ← NearestNode(nodes, rnd)
7:         new ← Steer(nearest, rnd, checker)
8:         if Valid(new) then
9:             Append(new, nodes)
10:        end if
11:        last ← nodes[Length(nodes) − 1]
12:        final ← Steer(last, rnd, checker)
13:        if Valid(final) then
14:            SetParent(last, final)
15:            return ToRoot(final)
16:        end if
17:    end while
18:    return NIL
19: end function
```

The *Commander* handles each pose in the waiting queue, picking them by priority, in order to ensure that the most important pose is always preferred to the others. Every time a pose is removed from the queue, it is assigned a predefined number of possible retries: if the pose fails for any reason, it's added back to the queue with its number of retries decreased. If the number of remaining retries gets to zero, the pose is discarded in the case of a priority value equal to zero. Otherwise, the priority is decreased and the pose gets assigned the default number of retries again.

Extracted a pose from the queue, the *Commander* has to choose the best arm to assign the said pose to. The easiest case happens when the pose has information about which arm has to take care of it, and the arm is not moving and it's ready to start. On the contrary, if the arm in question is busy, the *Commander* waits for it to finish its job, and then assigns the pose to it. The other case that can occur is when a pose can be processed by any of the arms indifferently: if both arms are free, then the pose will be assigned to one at random, otherwise, it will be assigned to the only free arm. If both are busy, the pose will be assigned to the first arm that finishes its current job.

The algorithm used to handle a given pose, named $Dispatch$, is presented in Figure II-C. The needed inputs are a description of the environment, i.e. env, the *GroupCommander*s, $groups$, and a pose, $pose$.

An initial check on the priority and the available retries allows to remove from the queue poses that the overall system is incapable to reach. The function $SelectGroup$ returns the most suitable *GroupCommander* to handle the given pose. If all the *GroupCommander* in the system are busy, the *Commander* will wait until one is able to serve the current pose. Once a *GroupCommander* is selected, a trajectory, T, is composed by the *Planner* using the algorithm $Plan$. In case the *Planner* fails to find a trajectory or takes more time than estimated, the number of retries of the pose is decreased by one and the pose is put back into the queue. The pose's priority changes accordingly to the number of times that the pose has been previously handled by $Dispatch$ without a trajectory being found. If the trajectory is valid, it'll be given to the *GroupCommander* and actuated.

III. Rework function

At the very core of the *Planner*, the $Rework$ function has the responsibility to compose a path between given initial and final poses, both described as a configuration in a *GroupCommander*'s joints space. Due to its popularity in the realm of motion planning, we decided to adopt as the main strategy the *Rapidly-exploring Random Tree* (RRT) algorithm [1], described briefly in Figure 3.

The idea behind RRT is to build, iteratively, a sequence of poses that will lead to the goal, starting from the initial one. Each pose, described as a configuration in the joints space, can be interpreted as a node and the sequence of the poses as a tree, rooted in the initial one. At every iteration a new random node is composed: it'll give indications about the direction in the searched space to explore. A path between the new node and the nearest one is computed and, if no collision is found, the new node will be added to the tree as a child of the nearest one. Once the tree reaches the final node, the solution can be found recursively by traversing it, starting from the final node and following the chain of parent nodes until the root is reached.

At the heart of the RRT algorithm, the distance and the strategy used to generate a random node must be suitable to be applied in a high dimensional space, i.e. the joints configurations space. Both must be chosen carefully in order to limit the impact of the *Curse of Dimensionality*. For what concerns the distance, we opted for the *Euclidean distance*. As for the exploration strategy, ideally, the best one would generate, over all iterations, a group of points sparsely distributed over the whole search space. Given its high dimensionality, the *Curse*

of Dimensionality limits the ability to explore it in such a manner. The space considered is the joints space, which can be linked to the space of all the reachable poses by a given robot. By looking at the robot's physical structure, it is possible to observe that, in general, variations of the same entity on the joints don't contribute equally to the corresponding variation of the pose. By intuition, it's possible to conclude that a fine exploration conducted on the subspace related to the most influential joints corresponds to a coarse exploration in the original space. The more the subspace considered is close to the original one in terms of dimensionality, the more the exploration in the original space becomes finer.

The strategy we adopted wants to reduce the dimension of the search space while being able to operate with configurations extracted from the whole joints space. In a new random configuration, the value of each influential joint is extracted from a uniform distribution over all of its assumable values, while the value of each uninfluential one is set to the corresponding value in the desired goal configuration. By setting the value of the uninfluential joints in this way, the strategy guarantees to reach the given goal without the need to rework an eventual plan found by considering only the most influential joints. Given the anthropomorphic structure of the robots, we decided to explore the subspace generated from the first three joints, considered starting from the robot's bases.

In the RRT algorithm, better results are tightly linked to a sparse exploration of the considered search space. The idea behind *Bidirectional RRT* algorithm, a variation of RRT inspired by *Bidirectional Search* [3], is to build trees rooted in the initial and goal configurations and to explore them at the same time. Ideally, this strategy would lead to a solution in half the time of the original method since, at each iteration, the number of explored configurations is doubled.

The overall algorithm is described in Figure 4. To achieve such performances, a pool of two threads is being used, each running the expansion process of a given tree, called $Explore$.

The $Explore$ function is composed by the instructions in lines from 5 to 10 of the description of RRT, Figure 3. A solution is found when the configuration obtained by any of the expansion processes is contained in the tree handled by the other one. The trajectory is composed by traversing from the intersection to both of the roots.

A drawback of the RRT algorithm is the absence of guarantees on the quality of the solution. An alternative is the *Rapidly-exploring Random Tree Star* algorithm [2], where a heuristic function is used in conjunction with the distance to converge to an optimal solution, asymptotically. Using the same approach of *Bidirectional RRT*, the *Bidirectional RRT** algorithm was developed.

IV. Implementation

The system is implemented using the `Python` programming language, installed in a `ROS` *Kinetic* virtual machine, downloaded from *ROS-I Images* [4]. To test and simulate the correctness of the implemented system, the virtual machine comes with `Gazebo`, `MoveIt`, and `rViz`. While `Gazebo` serves as a simulator for the robot and its surrounding environment, `rViz` allows looking at the simulated environment from the robot's point of view. Used in conjunction with `MoveIt`, they form a powerful tool, well suited for the task of *Motion Planning*.

Fig. 4. BidirectionalRRT

1: **function** $BidirectionalRRT(start, end, checker)$
2: $pool \leftarrow ThreadPool$
3: $s2eRRT \leftarrow RRT(start, end, checker)$
4: $e2sRRT \leftarrow RRT(end, start, checker)$
5: $i \leftarrow 0$
6: **while** $i < I$ **do**
7: $s2eNode \leftarrow Submit(Explore(s2eRRT), pool)$
8: $e2sNode \leftarrow Submit(Explore(e2sRRT), pool)$
9: $nodes \leftarrow Nodes(e2sRRT)$
10: **if** $Intersection(s2eNode, nodes)$ **then**
11: $e2s \leftarrow Intersect(s2eNode, nodes)$
12: $result \leftarrow ToRoot(e2s, [])$
13: **return** $ToRoot(s2eNode, Reverse(result))$
14: **end if**
15: $nodes \leftarrow Nodes(s2eRRT)$
16: **if** $Intersection(e2sNode, nodes)$ **then**
17: $s2e \leftarrow Intersect(e2sNode, nodes)$
18: $result \leftarrow ToRoot(e2sNode, [])$
19: **return** $ToRoot(s2e, Reverse(result))$
20: **end if**
21: **end while**
22: **return** NIL
23: **end function**

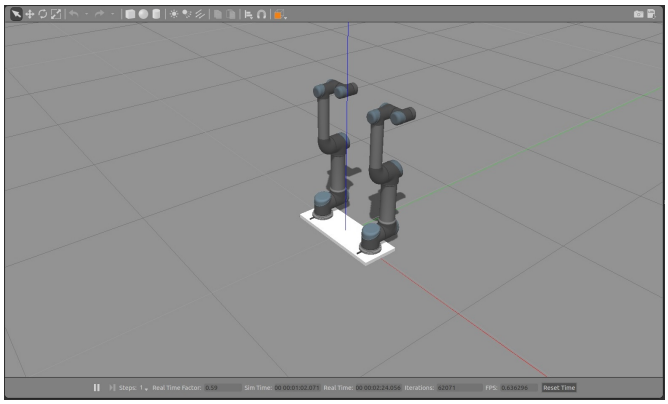

Fig. 5. The robot system loaded into Gazebo

All the code referenced in this section is included in a `GitHub` repository [5].

With the intention of having a simple and accessible way to test the system, a test suit `Python` script was written. The script lets the user choose among different pre-selected poses for the arms to reach, ranging from single-arm movement to situations that will trigger collisions. Besides single sets of poses, the script provides users an easy way to launch all tests at once: the so-called *Complete suit* test will execute all the listed tests in order, taking the arms back at their home position after every pose execution. Users can also choose to reach random poses: the poses will be randomly generated taking into account the robot joints limits, namely

$[-\pi, \pi]$, and the surrounding environment, which means that generated poses that violate those constraints, for instance an arm going underground, will be discarded beforehand. The test suit is included among all other scripts under the `two_arms` package.

With the purpose of debugging, a custom logging class was implemented. The logger is based on the default `ROS Python` logging class, included in `rospy`. The custom logger wraps the default `ROS` log with information about which module originated a particular log string. Moreover, it redirects the logs not only on the console but also to a file, named after the timestamp at which the main script was launched.

A. ROS interface

Since we wanted our focus to be on the commanders and the planner, we decided to keep the setup as simple as possible. We decided to implement the overall system as a *ROS Node* and to adapt as much as is achievable to the interface suggested by `ROS`.

For what concerns the robot model, we used two robot arms, both sharing the same model. We opted for the model *UR5*, provided by `ROS-Industrial` in the `Universal Robot` meta-package [7]. The description of the overall robot is defined in a single `URDF` file, which imports and builds upon the `URDF` description of the *UR5*, following the suggested best practices. Thanks to the `Setup Assistant` offered by `MoveIt`, it was possible to define and instantiate different joints groups and relative controllers, as well as to configure the whole underlying `MoveIt` ecosystem. The arms are linked together to a small box, laying underneath them and connected to the ground; to prevent the arms from going underground, a flat box is added beneath the robot system at run-time, leveraging the class `PlanningSceneInterface` provided by the `MoveIt Python` libraries. The whole robot model is included in the `two_arms_description` package; the simulation of the robot in `Gazebo` can be observed in Figure 5. Thanks to `Gazebo`, `MoveIt`, and `rViz` we were able to create and interact with simulated models, controllers, and actuators. Using nested namespaces we managed to distinguish each of the instantiated components and to call them appropriately.

In order to be able to interact with the *Commander* outside of the coding scripts with a `ROS`-like interface, three *ROS Topics* were instantiated, one for each of the *Commander*'s responsibility. The `\clear_queue` topic listens for a boolean value, which will be used to acknowledge when to clear the poses queue. The `\goal_pose` topic receives `GoalPose` objects, *ROS Message*s representing the poses that the robot arms must reach. Each `GoalPose` contains information about which arm has to take care of the pose, along with the priority value assigned to it.

The last one, `\status`, publishes *ROS Message*s containing the description of the *Commander*'s status, called `DualCommanderStatus`. A `DualCommanderStatus` message contains two `GroupCommanderStatus`, *ROS Message*s describing the status of the corresponding *GroupCommander*, as well as a list of `GoalPose`s, describing the content of the waiting queue. The `GroupCommanderStatus` message gives information about the currently handled trajectory, a boolean value stating whether the arm is moving or not, and the list of all goals processed by the *GroupCommander*.

All of the previously described topics are relative to the robot namespace, `\two _arms`; all the aforementioned messages are included in the `two_arms_msgs` package.

B. Commander, Planner and GroupCommander

While the *Commander* and the *GroupCommander* are implemented with a specific `Python` class, respectively `DualCommander` and `GroupCommander`, the implementation of the *Planner* has been split between the classes `DualCommander` and `TrajectoryReworker`.

The `DualCommander` class is responsible for the handling of the *ROS Topic*s previously described, being a subscriber for the `\clear_queue` and `\goal_pose` topics and a publisher for the `\status` one. A specific callback function is being assigned to each topic, able to handle the incoming messages. Since the callbacks are executed by different threads, shared objects were protected either by `Lock`s, thread synchronization primitives offered by the `multithreading` python standard library, or by choosing a thread-safe implementation; more in general, this strategy has been applied to all classes used by multiple threads. In particular, goal poses are stored in a priority queue, implemented using the `Python` class `PriorityQueue` from the library `Queue`, implicitly thread-safe.

The `GroupCommander` class is the implementation of *GroupCommander*. Having the problem to handle multiple poses at the same time, the `actionlib` package was used [8]. The package provides an action client/server implementation: while a single `ActionServer` is listening to the incoming requests on each joints group, every `GroupCommander` has its own `ActionClient`, which will send the given computed trajectory to the corresponding server. Different servers are identified thanks to the use of different namespaces. Besides taking care of sending the trajectories to the server, the client keeps track of the status of the goal pose that is being processed, making it possible to get feedback on what is going on under the hood. Using this mechanism, arms in the robot system can move independently one from another, without having to wait for other arms. For the purpose of planning the naive trajectory, `GroupCommander` wraps the methods `get_current_pose` and `plan` of the class `MoveGroupCommander`, offered by the `MoveIt` Python interface.

A fundamental part of a collision-avoidance system is to check if a determined robot state is, in fact, in collision with itself or with the surrounding environment. In order to make these controls, the system relies on a `MoveIt` service, i.e. *\check_state_validity*. The service receives a call requiring a `RobotState`, which is composed by the joints names of the whole robot and their values, the name of the planning group to check and potential constraints; the response states whether the inspected joint state is in collision and if that's

the case, information regarding contact points and the colliding robot links. Complications arise while trying to compare two different trajectories with the aim of finding likely collisions. Every trajectory is composed of a variable number of points, each representing a particular state (namely positions, velocities and accelerations) that the robot must reach in the specified timestamp. With that being said, two trajectories have, in general, a different number of points, each with a different timestamp, which means that a bare state checking comparison between two given trajectories is useless. Using the temporal information provided by each node, it is possible to solve eventual time mismatches between trajectories by interpolating the values. If the time deltas are small enough, velocities can be considered constant between two points and the interpolation strategy would describe the real movement with good precision. The `Python` class `StateChecker` allows performing such checks. Trajectories currently actuated by each *GroupCommander* are stored in a dictionary, updated when needed. By specifying the *GroupCommander* for which to check for collisions, all the trajectories of the other groups will be used as reference. Given a joints configuration and a timestamp, the `StateChecker` will find for each of the reference trajectories a pair of points that includes the input timestamp and, by linear interpolation, will compose the corresponding reference configuration, used to compose the overall state. The pair of points can be found using the *Binary Search* algorithm since a trajectory is basically a sequence of points, ordered by timestamp. Thanks to the properties of a trajectory and the *Binary Search* algorithm, a pair of points can be found with an average time complexity of $\mathcal{O}(n \log n)$, where n is the number of points in the trajectory. The so composed `RobotState` will be sent to the offered service and its result unpacked, interpreted, and returned. This feature assures that the trajectory to be taken as a reference can be dynamically changed during the execution and, moreover, lets the class be used without having to know its internal mechanism, abstracting the algorithm from the implementation. To optimize the initialization time, a single instance, properly prepared, of `StateChecker` is shared between different runs of the planning procedure.

The `TrajectoryReworker` class implements the checking for collisions loop and reworking functions of the *Planner*, while the composition of the naive path is delegated to the `GroupCommander` class and executed in the `DualCommander`. An instance of `StateChecker` is used to perform the checks. Each of the $Rework$ algorithms presented in the previous section is implemented in its own class. Historical information needed for the rework time estimation of a trajectory is stored using an *Exponential Moving Average*, using the update factor α defined as

$$\alpha \leftarrow \frac{2}{N+1}$$

where N is the number of previously stored items. To ensure that the physical constraints of each robot are respected, each composed naive trajectory is analyzed and the results are stored in a similar manner, to be used during the trajectory composition procedure. In particular, the analysis wants to obtain information regarding the average physically sound joints configuration variations, the time difference, and the distance between two consecutive points in a produced naive trajectory. The naive trajectory, built by `MoveIt`'s interface, respect such physical constraints. In the RRT and variations implementations, the goal is to produce nodes with a sound timestamp. In order to do that, the `steer` method allows obtaining an estimation of the correct time variation between two nodes by confronting the difference between the joints configuration variation of the input nodes and the historically registered difference. The so obtained factor will be applied to the historically registered time difference to compute a valid one. Once the points have appropriate timestamps, the resulting trajectory will be subject to a reworking procedure with the goal of composing points that will satisfy the physical constraints on the joints. To do that the historical joints variations and the variations between a pair of points are confronted and, if deemed appropriate, new nodes are introduced between the pair.

C. Limitations

During the implementation of the system, we encountered some roadblocks, of which we present a brief discussion.

1) MoveIt Python interface: The first limitation we came across during the development of our project is the scarcity of features available in the `MoveIt Python` interface, with respect to the `C++` one. The most problematic difference lays in the interface provided to interact with the robot planning scene: while in `C++` there are several methods at disposal, ranging from single state collisions checking to whole paths validity checking, `Python` does not have the same features. `Python` misses completely the `PlanningScene` class, which contains all the aforementioned useful methods. Given that, the language capability to interact with the planning scene is limited to the addition and removal of objects in the scene. Being in a multi-robot environment that aims to avoid collisions, the ability to check whether a given state of the system is in collision or not is cardinal. Provided this `Python` library limitation, the solution was to rely on a `MoveIt` service called *\check_state_validity*, although that led to non-negligible time consumption. We also discovered that the `PlanningSceneInterface`, offered by the `MoveIt`python interface for `ROS` *Kinetic*, included a bug that, if the robot was instantiated in a namespace, prevented the planning scene to correctly load and, so, to add or remove objects from it; we wrote a simple *UNIX* script to install the correct version of the class.

Another relevant limitation with respect to our objectives was that the `MoveIt` framework is not capable, at the time of writing, to execute synchronous movements. That translates to robot arms not being able to move at the same time: instead one has to wait for the other to complete its task before proceeding with its own pose. In a real multi-robots environment, that is of course not acceptable. Luckily, a workaround that does not rely directly on the `MoveIt` methods exists, namely the `actionlib` package. With the package, an `ActionServer` can receive multiple requests at once from

```
[ WARN] [1635347196.413703282, 0.147000000]: The default_robot_hw_sim
 plugin is using the Joint::SetPosition method without preserving the
 link velocity.
[ WARN] [1635347196.413736744, 0.147000000]: As a result, gravity wil
l not be simulated correctly for your model.
[ WARN] [1635347196.413752003, 0.147000000]: Please set gazebo_pid pa
rameters, switch to the VelocityJointInterface or EffortJointInterfac
e, or upgrade to Gazebo 9.
[ WARN] [1635347196.413762002, 0.147000000]: For details, see https:/
/github.com/ros-simulation/gazebo_ros_pkgs/issues/612
```

Fig. 6. Warning in Gazebo, related to the gravity bug

different clients, making it possible to execute synchronous movements. The last programming language limitation we had in our project is not relating to `MoveIt` libraries, but to the properties of the languages themselves: while `C++` is a compiled language, `Python` is on the contrary interpreted. This difference can lead to divergent code efficiency.

2) Gazebo gravity bug: The `ROS` version used in the project, that is `Kinetic`, contains a bug relating to `Gazebo` environment, especially for what concerns the simulation of gravity, as can be seen in Figure 6.

Relying on what the warning suggests, the solution could be to upgrade `Gazebo`, which would require the upgrade of the whole `ROS` version, or to switch the joint interface used from the default *PositionJointInterface* to one between *VelocityJointInterface* or *EffortJointInterface*. We briefly tried to switch to both the suggested interfaces but we couldn't manage to set the `PIDs` in such a manner to have it working properly. The impact of this bug was, everything considered, marginal: it seemed to kick in only when the model was just loaded, causing a minimal oscillation.

3) Virtual machine computational power: Being `ROS` a complex and vast set of software libraries and tools, the easiest solution to get started is to install a ready-to-use *Ubuntu* image, available on the `ROS` website, rather than manually installing all the required tools on a pre-existing operative system. This solution has however a compromise: the performance of the overall system is significantly slower, compared to `ROS` running on a native OS. This limitation hinders not only the efficiency of scripts but also in a significant way the usage of simulations environments: our tests and simulations were conducted in environments, like `Gazebo` and `rViz`, often running at less than one frame per second. We firmly believe that using a virtual machine is one of the causes of our non-acceptable re-planning time consumption. As native Windows users, we tried to operate using the *Windows Subsystem for Linux* (WSL), hoping for better performances. The main problem we encountered was that we couldn't manage to make `Gazebo` work on a `XServer`, forcing us to the virtual machine for our simulations.

V. Overall performance

Having four possible algorithms at disposal to rework paths in a collision state, we decided to measure the performances of each of them on a well-defined common benchmark trial to understand which one could be the best fit for our case. The benchmark essentially requires the considered algorithm to find a trajectory between two configurations. Since we are working with only two robot arms, we decided to move the left arm to a predefined position and let the right arm be the rework subject. We decided to use as final configuration one such that the algorithm would need to work in close proximity to the other arm, potentially generating new configurations which result in collisions. Since all are variations of RRT, for which randomness plays a key role, we decided to run each instance five times and to statistically analyze them; the results are shown in Table I.

TABLE I
Algorithms benchmark results

Algorithm	t_1	t_2	t_3	t_4	t_5	μ	σ
RRT	18.10	77.99	97.82	46.81	17.88	**51.72**	35.76
RRT*	23.02	483.97	24.82	94.41	22.18	129.68	200.43
Bidirectional RRT	110.38	104.12	94.61	79.74	89.35	95.64	12.07
Bidirectional RRT*	88.23	181.04	194.99	253.88	140.10	171.65	61.95

The best performing algorithm as for this benchmark seems to be `RRT`. Also its heuristic counterpart `RRT*` seems to have good performances, although its mean is much higher due to an outlier. `RRT*` differentiates from `RRT` by the fact that the `steer` method, which takes the current node to the nearest one, is called much more frequently. The method appears to be the bottleneck of the class.

The bidirectional alternatives of those aforementioned algorithms do not act as expected. In fact, the time taken by the bidirectional algorithms is much more compared to their plain versions. Bidirectional algorithms require an additional check that must be done at every iteration: frontiers of both sides, i.e. the "start pose to end goal" path and the opposite "end goal to start pose", must be iterated in order to find possible intersections and, if one is found, the algorithm terminates successfully. Although that check needs to be done at every iteration, that does not seem the main cause of the severe time consumption, as the intersection search has a time complexity of $\mathcal{O}(n)$, where n is the number of iterations. The main cause has to be searched on the used programming language characteristics: in `Python`, only one thread at a time can hold control of the interpreter, regulated by the *Global Interpreter Lock*. That causes the thread execution not to be parallel at all, although still not sequential, basically making the bidirectional approach, implemented with threads, time-wise too expensive. Moreover, an overhead is introduced due to the synchronization between threads, synchronization that is required to check for intersections in the paths frontiers. Still, the observation on `RRT*` holds: between the `RRT` variant and the `RRT*` one, the latter is more time-consuming. We are quite comfortable in saying that a truly parallel implementation of the bidirectional variants could be time-wise interesting: as a rough estimate of its performance, we could simply halve the corresponding times, resulting in overall better performances in respect to the original version, making `Bidirectional RRT`, as for the average, the overall best. A truly parallel implementation could be achieved by using Python `Processes` instead of `Threads` or by using other programming languages, like `C++`.

As for the overall performances of the system, we found them to be extremely unsatisfactory in our tests: when collisions were found, the rework process consistently took more

time than expected, basically allowing the other moving arm to conclude its trajectory before reaching a solution, making the entire process pointless. We think that the use of python plays a good role in the disappointing performances: in the end, it's an interpreted programming language, suited more for scripting than for applications to be used in a real-time environment. More importantly, we found out that the main time-wise bottleneck resides in a workaround we implemented to overcome the limitations of the `MoveIt` interface.

The `MoveIt` library provides two different interfaces, one written in `C++` and one in `Python`. While the `C++` API is up-to-date and well maintained, we cannot say the same for the `Python` counterpart, which lacks lots of useful methods and classes. A clear example of that behavior can be seen while attempting to check the validity of two sets of joints, one for each robot arm: while the `C++` interface provides a convenient method, namely `checkCollision` included in the `PlanningScene` class, the `Python` one misses not only the said method but also the whole class. A workaround for this is to call a particular `MoveIt` service, *\check_state_validity*, although increasing the time required to complete the algorithm as we saw in previous sections. Our implementations of planning algorithms suffer from this problem, resulting in a project that is not feasible in a real multi-robots environment. To show the difference between the two programming languages libraries, we implemented two equal benchmark classes, one for each language. The task of the class is to load 100.000 different joint configurations from a given file and to check every entry against the language corresponding state validity control method. In particular, every entry is composed of twelve elements, i.e. the six joint values of the left arm and the six joint values of the right one.

Looking at the results of the conducted test, we can assert that the method executed with `C++` is almost sixty times faster than the `Python` counterpart. Talking numbers, we have ~ 10 seconds `C++` total time execution against the ~ 570 seconds required by `Python`: given the results, we can state that our planning implementation is limited by the language chosen and the corresponding available library. Note that the total time is the only metric of comparison available: by looking at the duration of a single state validation call, the `C++` implementation is so fast that it's consistently equal to 0, while the `Python` implementation gives times in the order of the milliseconds. As we have seen beforehand, to re-plan a trajectory in order to avoid some predicted collision, we have to "circumnavigate" the said collision point by finding new nodes and building intermediates nodes between them to form a final path. Using that procedure requires every iteration of the algorithm to check the validity of the states of all generated nodes, which affects severely the total time required by the algorithm to complete. The benchmark scripts are included in the `two_arms_benchmarks` package.

VI. Conclusion

In this paper, we propose a system able to handle synchronous movement and to avoid collisions in a dual-robot environment.

The whole process is possible thanks to a series of *Group-Commander*s, one for each arm in the environment and one master commander in charge of managing all separate robots. A *Planner* is used in order to compute trajectories that are free of collisions, starting from a goal pose that must be reached, while a *Commander* allows the two components to cooperate.

The system makes use of interpolation for the comparison between different trajectories and, in case a collision is found, a new path is computed using a version of the Rapidly-exploring Random Tree algorithm.

As shown by the results, *RRT* is the best algorithm in our use case. *RRT** provides a path that is assured to be optimal, but that's not the main focus of the problem, as the quickness of generating a solution is preferred to its optimality. *Bidirectional* versions show the potentials of being an improvement over the studied basic algorithms, although bounded by the programming language used.

Most of the work is limited by virtual machines as well as the programming language chosen, `Python`, which besides being an interpreted language and not compiled, lacks a complete and maintained *API*.

Future developments could include a full rewriting of the code in `C++`, executed on a native OS. Improvements on the proposed *Bidirectional RRT* algorithm could be focused on how paths frontiers are iterated to search for interceptions: optimizing the code on the said part could underline the potential advantages of the bidirectional implementation with respect to the plain *RRT* version.

References

[1] LaValle, Steven M and others (1998), *Rapidly-exploring random trees: A new tool for path planning*, Ames

[2] Sertac Karaman and Emilio Frazzoli (2010), *Incremental Sampling-based Algorithms for Optimal Motion Planning*, arXiv

[3] Pohl, Ira (1971), *Bi-directional Search*, Meltzer, Bernard

[4] ROS-I Images, [Online]. Available: https://rosi-images.datasys.swri.edu

[5] GitHub repository for the implementation, [Online]. Available: https://github.com/jacknot/sync_move_collision_avoidance_robot_planner

[6] MoveIt documentation, Python interface related to ROS Kinetic [Online]. Available: http://docs.ros.org/en/kinetic/api/moveit_tutorials/html/doc/move_group_python_interface/move_group_python_interface_tutorial.html

[7] ROS-Industrial Universal robot package [Online]. Available: https://github.com/ros-industrial/universal_robot

[8] actionlib package [Online]. Available: http://wiki.ros.org/actionlib

Jacopo Mora Student at University of the Studies of Brescia

Matteo Salvalai Student at University of the Studies of Brescia