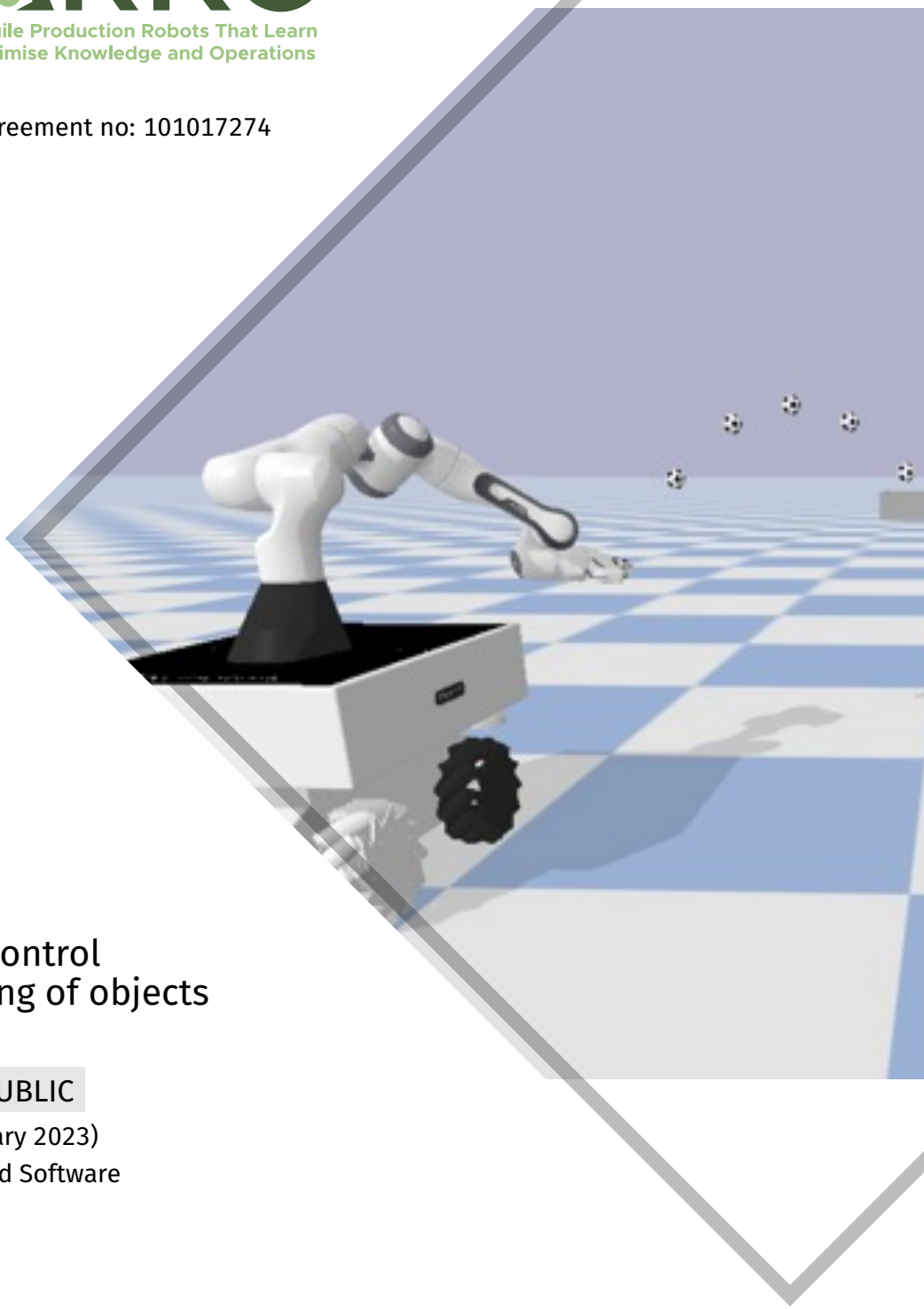
DARKO

Dynamic Agile Production Robots That Learn and Optimise Knowledge and Operations

H2020-ICT-2020-2 Grant agreement no: 101017274



DELIVERABLE 4.5

Preliminary robot control for dynamic throwing of objects

Dissemination Level: PUBLIC

Due date: month 26 (February 2023)

Deliverable type: Report and Software

Lead beneficiary: EPFL

1 Introduction

Throwing is a form of dynamic manipulation [14, 8] to augment the feasible work space of the robot and increase the efficiency of object manipulation. This is highly desirable in applications such as logistics and handling of goods, and thus forms an integral part of the DARKO project. With the emergence of machine learning, there is a growing popularity on using model-free methods that distil skills from offline training. With sufficient data coverage, the robot is able to handle a wide set of tasks once deployed. Despite impressive empirical result, most model-free methods in the literature (such as [22]) lead to algorithms which are difficult to generalize to new tasks and can not ensure successful task execution.

Consider a target box at a desired location and projectile motion of an object. The throwing problem can be reduced to the following simple question: *Find (if there exists) a feasible trajectory such that the object released at the end of the trajectory lands in the target box.* The object is subject to gravity (in case of projectile motion) and to air drag, depending on its shape and configuration during release. The throwing problem is therefore an optimization problem whose variables are: joint positions and velocities, base position and velocity. The constraints are twofold. One set of constraints arises from the limitation of dynamic capabilities of the robot and the other from the fact that after release the object must land in the target box.

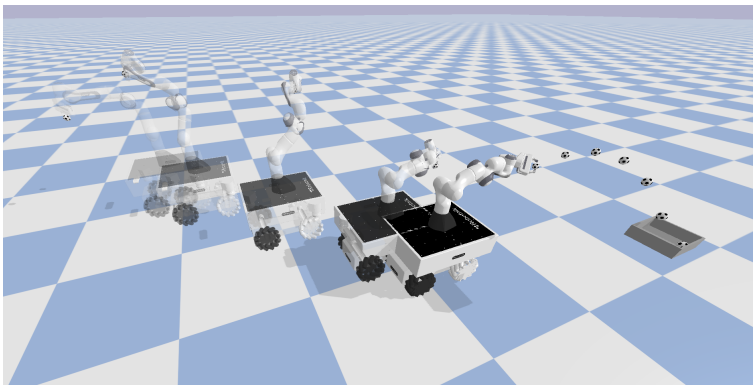


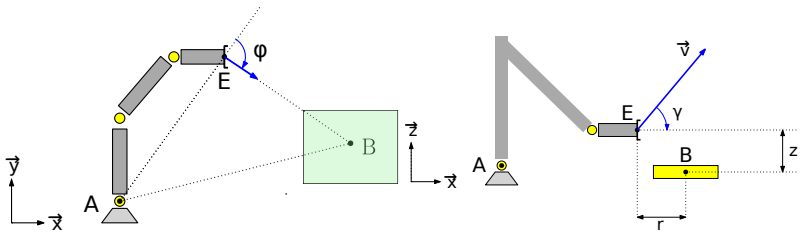
Figure 1: Snapshots of the implementation of EPFL throwing approach to enable a mobile manipulator to throw a ball in a bucket while moving.

1.1 Contributions

This deliverable reports on two contributions:

It first reports on the approach developed by EPFL to solve the planning of feasible throwing trajectories. The approach relies on computing a library of feasible throwing postures retrievable in real-time. The problem is formulated as a nonlinear optimization. The approach combines the strength of model-based methods and model-free methods: model-free learning to model the complex non-linear object flying dynamics [9] and model-based approach to formulate throwing as a feasibility problem. The approach was assessed in both simulation and on a real platform using the panda robot. We demonstrate the applicability of the throwing method to control real-time adaptation of the throw trajectory to adapt to live displacement of the target's box. The work was published in [11]. The code is released as open source, see Section 2.5.

In the second part of this develiverable, we report on advances made by UNIPI to improve throwing capabilities of standard actuators. Indeed, despite the promising results



(a) Top view (throwing triangle) **(b)** Side view (throwing plane)

Figure 2: Visualization of throwing triangle and throwing plane.

obtained by EPFL when using the Franka Emika Panda robot for throwing, the limited torques of the Franka Emika Panda severely limits the throwing workspace. In very dynamic tasks like throwing, the safety bound on the velocities at both at the joint and at the end-effector level are significant. With the aim of overcoming the intrinsic limitations of a collaborative robot as the Franka Emika Panda in very dynamic tasks, this deliverable also describes the initial implementation of the software needed for the use of a pneumatic hand-tool developed by UNIPI (see D2.1 for a description of the pneumatic hand-tool itself from a hardware point of view). The pneumatic hand-tool, which can easily grasped by the general purpose gripper developed within the DARKO project (see D2.1 for details) is mainly designed for picking objects placed in an intricate way inside the box or, alternatively, well-ordered objects close to one another in a way that the gripper have no physical space for grasping them. The hand-tool is indeed able of carrying out a gripping action on the surface of an object with a suction cup and a pneumatic circuit, which generates the vacuum by means of a Venturi pump. However, it is also able of carrying out a pushing action on the same object, by means of a parallel pneumatic circuit which inverts the airflow (regulating pressure and airflow opening time), to launch the object in a desired target box beyond the working area of the robot.

In what follows, we describe the methodology followed by EPFL and UNIPI separately (see Section 2 and 3, respectively). We also discuss aspects of the software that we aim to augment in the forthcoming deliverables.

2 Planning Feasible Throwing Trajectories (EPFL)

2.1 Background and literature survey

Most early works on robot throwing [13, 20, 16, 6, 18, 15, 10] worked on simple robots with low degrees of freedom [13, 20, 16, 15, 10] or had a specific object to throw [6, 18]. Works such as [12, 21] have tried to accurately model the flying dynamics of objects as well as the robot dynamics and determine the optimal robot throwing trajectory via numerical optimization. However, the optimization problem for throwing is hard to solve because of nonconvex constraints. In [21], it takes 0.5 s to generate a feasible throwing trajectory for a 3—DOF robot. Therefore, it presumably needs at least several seconds for the mobile manipulator with 7—DOF Franka Emika Panda arm along with 2-DOF omnidirectional Robotnik base and is hence unsuitable for online motion generation.

In the state of the art, there are numerous grippers, rigidly mounted on the robot, which allow gripping by means of the use of a suction cup and a vacuum pump. However, they are mainly used for quickly griping and releasing objects, without being able to launch them beyond the working area of the robot itself. However, there are some grippers that were initially designed for gripping objects and, only later, were also tested for throwing

objects within a short range [2, 1]. The gripper consists of an elastic membrane, filled with granular material (e.g.: coffee powder) which allows of grasping objects by inverting the pressure and to generate a pushing force that allows objects to fly ("jamming gripper") by introducing pressured air inside the elastic membrane.

2.2 Problem Formulation

The geometric relationship between robot mobile manipulator and task throwing is shown in Fig. 2a and Fig. 2b. We observe that there exist geometrically equivalent structures in mobile manipulator throwing, namely:

- Mobile manipulator alone is equivariant in horizontal position because of the mobile base with omnidirectional wheels.
- Throwing alone is equivariant in incident direction.
- Mobile manipulator throwing can be described by throwing triangle $\triangle AEB$, which is equivariant in rotation around Z-axis at target box B.

2.2.1 Backward Reachable Tube

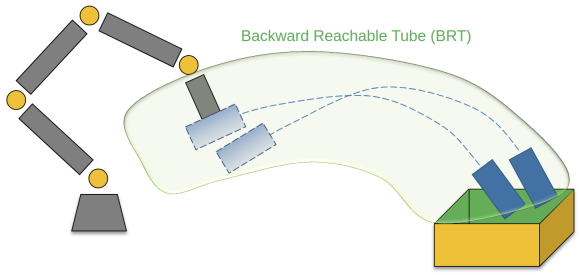


Figure 3: The Backward Reachable Tube models the set of valid throwing configurations, given the object's flying dynamics.

In the throwing plane, we denote the object flying state as $x = [r, z, \dot{r}, \dot{z}]^\top$. The flying dynamics is described by a first order differential equation $\dot{x} = f(x)$, $x \in \mathbb{R}^4$. We denote the flying trajectories of f starting from state x_0 as $\zeta_{f,x_0}(t) : [0, +\infty] \to \mathbb{R}^4$. We assume that a user is giving the robot a landing target set $\mathcal{X}_l \subseteq \mathbb{R}^4$, describing the allowed landing position slack and allowed range of landing velocities. In order to throw successfully, we only need to drive the robot into the BRT and release the object. The BRT is generated from object flying dynamics.

We sample landing configurations from the landing target set $\mathcal{X}_l$, and solve the flying dynamics backwards in time as a Initial Value Problem (IVP) and then aggregate the ODE solutions. We sample 2160 landing states inside $\mathcal{X}_l$, solve the flying dynamics backwards in time for 1 second to get the 2160 flying trajectories, then aggregate all the data points on the trajectories, filter out the data with high velocities that are for sure not feasible by the robot ($|\dot{r}| > 5.0$, $|\dot{z}| > 5.0$), finally yield 75000 throwing configurations in throwing plane coordinates.

2.2.2 Feasibility problem formulation

In order to find throwing configurations in robot joint space, there are two types of nonlinear constraints to be considered:

- Equality constraint: the throwing triangle $\triangle AEB$ defines a task manifold, i.e. for fixed $\overrightarrow{AB}$, the end-effector position $\overrightarrow{AE}$ must lie in the throwing plane defined by $\overrightarrow{EB}$, which imposes a nonlinear equality constraint in the joint configuration q.
- Inequality constraint: joint position limit and joint velocity limit.

These constraints can be expressed into the Throwing Feasibility Problem (TFP) in Appendix A.1. The independent variables are $\{\overrightarrow{AB}, q, \dot{r}, \dot{z}\}$ and hence the optimization problem (TFP) can be restated as:

$$\text{Find } \left\{\overrightarrow{AB}, q, \dot{r}, \dot{z}\right\}$$
$$\text{such that: } \begin{cases} q_{\min} \le q \le q_{\max} \\ \dot{q}_{\min} \le J^{\dagger}(q)\vec{v}(\overrightarrow{AB}, q, \dot{r}, \dot{z}) \le \dot{q}_{\max} \\ f_{BRT}(r(\overrightarrow{AB}, q), z(\overrightarrow{AB}, q), \dot{r}, \dot{z}) \le 0 \end{cases}$$

(TFP-Reduce)

2.3 Solution of optimization problem

2.3.1 Robot Kinematics and Velocity Hedgehog

The optimization problem (TFP-Reduce) is solved by separating the last constraint from the rest of the problem. Other than the last constraint, we are left with finding $\{\overrightarrow{AB}, q, \dot{r}, \dot{z}\}$ such that the joint angles and velocities respect the kinematic constraints. We build a dictionary called the Velocity Hedgehog (VH) which efficiently stores the joint configuration q and the maximum end-effector velocity (which in polar coordinates is described by v_{max}, ϕ and γ as shown in Figure 2) corresponding to an end-effector height z. The velocity hedgehog discretizes z, ϕ and γ into cells. To summarize, each cell stores,

- The maximum feasible end-effector speed v_{max} at height z along direction (ϕ, γ);
- The robot configuration q that enables v_{max}.

A typical velocity hedgehog is shown in Fig. 4, indicating that the feasible velocity set is not convex but appears as needles with different lengths along different directions and resembles a hedgehog. We discretize the throwing height with grid $Z = [0 : 0.05 : 1.2]$, the throwing yaw angle with grid $\Phi = [-90 : 15 : 90]$ and the throwing pitch angle with grid $\Gamma = [20 : 5 : 70]$, to yield 3289 cells of robot throwing candidates. The velocity hedgehog of the Franka Emika Panda is generated using Algorithm 1 (in Appendix A.2) with 1 million joint state samples.

2.3.2 Combining VH with BRT to determine throwing configurations

We first group the BRT data according to height z and throwing pitch angle γ. Then for BRT data in bin indexed by $\hat{z}$ and $\hat{\gamma}$, the initial guesses for throwing state are the BRT data whose flying speed is smaller than the maximum feasible velocity at a certain $(\hat{z}, \hat{\gamma})$ stored in velocity hedgehog. From this matching operation, we can also read out the corresponding joint configurations and throwing yaw angles from velocity hedgehog, resulting in initial guesses for throwing configurations (q, ϕ, x). The obtained feasible initial guesses can be fed into Problem TFP-Reduce to be further refined. With (q, ϕ, x) determined, all the other decision variables can be written in closed-form as shown in Problem TFP. As a result, the overall architecture to solve TFP efficiently and reliably is shown in Fig. 5.

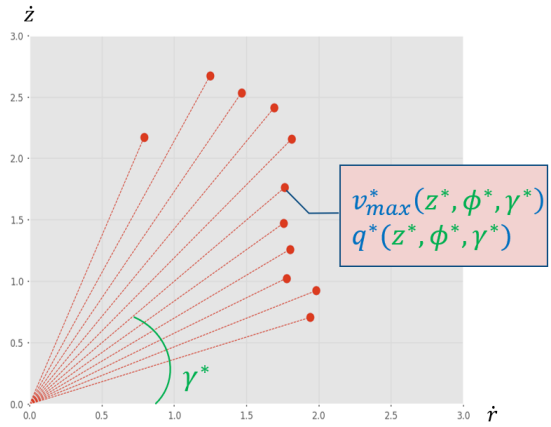


Figure 4: Illustration of a typical velocity hedgehog. Red needles represent the feasible velocities along different throwing pitch angles with fixed throwing height z and throwing yaw angle ϕ, which are stored in velocity hedgehog. Given a certain throwing height and throwing direction (z^*, ϕ^*, γ^*) query, velocity hedgehog gives maximum feasible velocity v^*_{max} and the joint configuration q^* that enables v^*_{max}.

2.3.3 Trajectory generation towards a throwing configuration

Given an initial state of the robot $(q_0, \dot{q}_0)$ and feasible throwing configuration $(q_d, \dot{q}_d)$, we use Ruckig [5] to generate a velocity, acceleration, jerk-limited time-optimal trajectory to move the robot towards the throwing configuration from an arbitrary initial robot configuration. The software is publicly available at https://github.com/pantor/ruckig under MIT License. We rule out the trajectories which violate the joint position and torque constraints.

The trajectories generated by Ruckig cannot be implemented on the robot and hence to throw with the DARKO platform. Moreover, we ignore many throwing configurations due to inavailability of a feasible time optimal trajectory. To address this issue, EPFL has developed an MPC planner (details in the Section 2.3.4) to generate time optimal trajectories which respect the position, velocity, acceleration and jerk and torque constraints for arbitrary candidate throwing configurations. Besides the box constraints at the joints, an additional height constraint is also incorporated to avoid collision with the mounting surface of the robot.

2.3.4 Generating a dynamically feasible trajectory

The MPC planner (available at https://github.com/AlbericDeLajarte/mpc_motion_planner) generates a trajectory between the initial joint state of the robot and a selected `throw_candidate` with the function

```
build/offline_trajectory
```

It uses PolyMPC (https://gitlab.epfl.ch/listov/polympc freely available under Mozilla Public License Version 2.0) as the MPC solver together with an initial guess trajectory given by Ruckig.

2.4 Preliminary Robot experiments

OptiTrack is used to detect position of the target box. The software module to in Section 2.6.1 is used to generate `throw_candidate` and select one of them in the Throwing Plan-

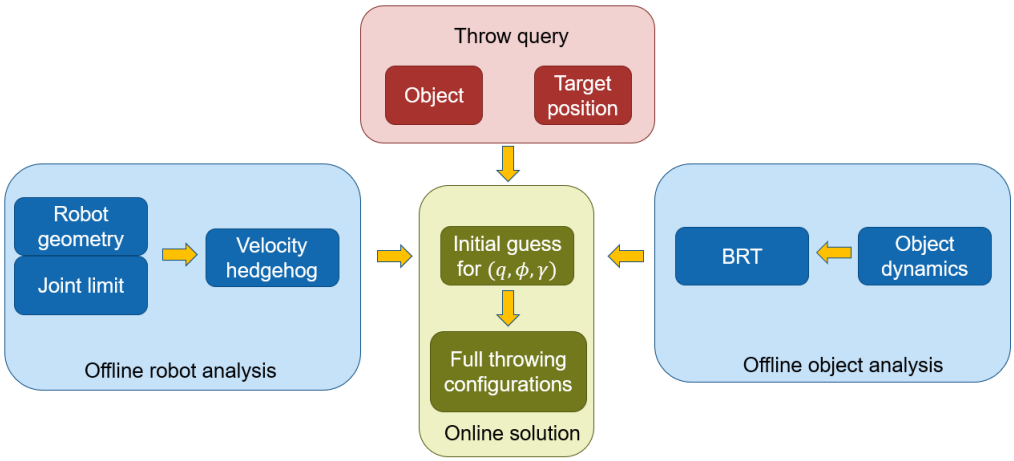


Figure 5: Framework to obtain throwing configurations. The velocity hedgehog and BRT are combined as outlined in Section 2.3.2.

.

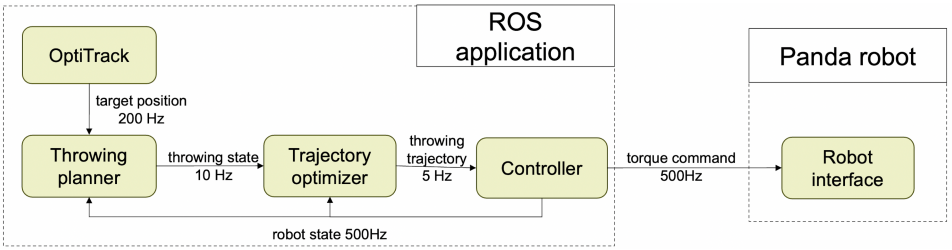


Figure 6: Pipeline of robot implementation

.

ner (in Figure 6). The trajectory optimizer (in Figure 6) uses the MPC algorithm in Section 2.3.4 to compute a dynamically feasible trajectory to reach the desired `throw_candidate` in 0.5ms. The controller computes the torque command to follow the generated trajectory in the robot joint space. The feedback loop in Figure 6 allows for re-computation of the torque command at 5Hz for a new target box position. The feedback loop is therefore adaptive to changes in the box position.

The experiment is performed for a target box size of size $33cm \times 23cm$ and box positions in the range $(0.6m, 1.3m)$ along with box heights relative to base of the robot in the range $(-0.3m, -0.05m)$. The maximum throw distance from base is $1.3m$ as the Cartesian velocity limit of the Franka Panda is $1.7ms^{-1}$. We considered two objects: a 3D printed ball with weighing 100g and 60 mm in diameter and, a cardboard box from DARKO weighing 120 g and measuring $80mm \times 65mm \times 60mm$. The results obtained are summarized in Table 1. The approach shows promising success rate for both the objects considered.

Table 1: Performance of throwing methodology

	3D-printed ball	Cardboard box
Success rate (landed in box)	88% (22 out of 25)	79% (23 out of 29)
Landing error range	$1.31cm$ to $15.36cm$	$4.15cm$ to $26.36cm$
Landing error (mean $\pm$ std.)	$7.60cm \pm 4.13cm$	$12.48cm \pm 5.79cm$

(a) 3D-printed ball **(b)** Cardboard box (DARKO object)

2.5 Opensource Code Release

EPFL has made publicly available the software package in https://github.com/epfl-lasa/mobile-throwing for its throwing approach. In this package, the input is the target box position in Cartesian coordinates and the output is a dynamically feasible trajectory. The object released at the end of this trajectory lands in the box. The simulation is performed in Python and executed in PyBullet. The object is a soccer ball which is pre-modelled in PyBullet. The Franka Emika Panda and Robotnik Kairos urdf which are publicly available are used for the simulation. We also provide a docker container for the Python code, which enables it to run on any operating system. EPFL has also performed robot experiments to study the performance of the throwing methodology on objects from DARKO. In order to do so, EPFL has designed a reactive feedback loop which offers quick adaptation to new target box positions. EPFL has also developed an MPC planner to generate position, velocity, acceleration and jerk limited, torque constrained trajectories for arbitrary candidate throwing configurations.

2.6 Software Description

2.6.1 Generating a throwing configuration

The software code (https://github.com/epfl-lasa/mobile-throwing) to generate a throwing configuration is written in Python and is packaged in a Docker container which can be built locally using

```
build-docker.sh
```

It is hence versatile to be implemented on any Linux, Windows, or MacOS computer providing the user with OS level virtualization. We provide a simulation tool with PyBullet to visualize the planned throw with the DARKO robot platform. The function

```
ms1_demo_mobile_manipulator_throw.py
```

admits the box position (x-y-z) position relative to base of the robot as input. The joint limits of the robot are incorporated from https://frankaemika.github.io/docs/control_parameters.html. The BRT data computed as elaborated in Section 2.2.1 is stored in

```
brt_path = "object_data/brt_gravity_only"
```

and the velocity hedgehog dictionary computed from Algorithm 13 in Section 2.3.1 and is stored in

```
robot_path = "robot_data/panda_5_joint_dense_1_dataset_15"
```

The intersection of these point clouds (as described in Section 2.3.2) for a target box at height z is performed in the function

```
brt_chunk_robot_data_matching(z,robot_path,brt_path)
```

which outputs candidate `q_candidates, phi_candidates throw_candidates` in task space in accordance with the algorithm presented in Section 2.3.2. For each of the `throw_candidates` and for the current state of the robot, we compute a jerk limited trajectory using Ruckig in `get_traj_from_ruckig`. Out of the `throw_candidates` we select the candidate with which requires the minimum time to execute its trajectory. We obtain one such trajectory in 0.1 ms. The function `throw_simulation_mobile` simulates the throw using the selected `throw_candidate` and its corresponding trajectory.

3 Dynamic Throwing With Pneumatic Hand-Tool (UNIPI)

With the aim of overcoming the intrinsic limitations of a collaborative robot as the Franka Emika Panda in very dynamic tasks, UNIPI has extended the use of the pneumatic hand-tool also for throwing (see D2.1 for a description of the pneumatic hand-tool itself from a hardware point of view). The pneumatic hand-tool is indeed mainly designed for picking objects placed in an intricate way inside the box or, alternatively, well-ordered objects close to one another in a way that the gripper have no physical space for grasping them. In this deliverable, we describe how the hand-tool can be also profitably used for exerting a pushing action on the picked object, by inverting the airflow. By regulating both the pressure and airflow opening time, it is possible to launch the object in a desired target box beyond the working area of the robot. This way of throwing is intrinsically more safe as the manipulator does not need to dynamically move, with evident dangers for near persons. Moreover, it is able to overcome the limited performance of the Franka Emika Panda in terms of throwing distance and precision.

In this deliverable, we mainly describe the software developed by UNIPI for the use of the hand-tool for throwing. The software consists of three components: 1) a ROS service implementing the *human-like* Cartesian motion planning (see D4.1 for a detailed description of this technique) for moving the manipulator between two planned configurations (the current one and the desired grasping/throwing one), 2) the ROS publisher implementing the pneumatic circuit controller for activating the Venturi pump to pick up the object or the pressure of outgoing airflow to throw it, 3) a ROS node implementing the neural network to set the robot configuration (position of the hand-tool and the orientation) and the pneumatic circuit parameters (the pressure of the outgoing airflow and the opening time of the solenoid valve) for throwing the object towards a desired destination.

3.1 Brief description of the pneumatic hand-tool

The pneumatic hand-tool consists of a handle to be gripped by an anthropomorphic gripper (such as the one developed within the DARKO project, D1.2) and a suction cup at its end. The hand-tool is designed to be anchored to the shelving (by means of magnets), just above the boxes that contain the objects (see D8.1) the mobile manipulator has to grasp. The hand-tool is identified by using the onboard cameras, which allow obtaining the position and orientation of e.g. a marker rigidly attached to the hand-tool itself or to the shelving at a fixed and known position near the hand-tool.
The hand-tool is connected via a compressed air hose to the output of a 3/2 (Normally Closed) solenoid valve powered at 24 V(DC), which allows switching between the gripping and the pushing functions. The gripping function is obtained by the pneumatic vacuum circuit, consisting of a Venturi pump powered at 24 V(DC) that generates a certain degree

of vacuum. The pushing function instead is obtained by a second pneumatic circuit where a pressure regulator, powered by 24 V(DC), can be tuned with the aim of generating different pushing forces and hence the desired throwing distance. The pressure regulator, which accepts input voltage values between 0 and 10 V, allows adjusting the pressure linearly with respect to the input voltage, in a range between 0 and the maximum available pressure (10 bar for this first prototype). Both circuits are connected via quick coupling fittings to the connections of an air compressor.

3.2 Software description for picking and throwing with the pneumatic hand-tool

Human-like Cartesian path planning. The planning phase that allows the robot to reach the hand-tool, the object to be launched, and the throwing configuration is based on a *human-like* Cartesian planning (similarly to [3] which is at the joint level), which is based on the functional principal component analysis of the human arm [4]. The basic unit that allows planning is written using a ROS service.

```
darko_arm_planning:
type: git;
url: https://gitsvn-nt.oru.se/darko/software/darko_arm_planning.git
version: master-unipi
```

through the definition of an appropriate message ("`pose_plan.srv`") inside the package. The service requires as inputs the current pose of the manipulator and the desired grasping/throwing pose as a "`geometry_msgs::PoseStamped`" type object. Both poses are expressed in terms of the three spatial coordinates x, y and z and the orientation as a quaternion. The ROS service furnishes as output the calculated trajectory as a "`geometry_msgs::PoseStamped`" type array, which contains the sequence of intermediate planned poses that connect the current pose to the desired grasping/throwing pose. The trajectory obtained is performed by means of a Cartesian impedance controller,

```
darko_arm_control:
type: git
url:https://gitsvn-nt.oru.se/darko/software/darko_arm_control.git
version: master-unipi
```

which is implemented according to the ROS Control architecture available in ROS. The codebase has been adapted for ROS Melodic to align with the version installed in the DARKO platform.
To carry out the throwing task with the pneumatic hand-tool, the cobot equipped with an anthropomorphic gripper has to sequentially perform the following subtasks: firstly, the robot has to reach the pneumatic tool and grasp it through the anthropomorphic gripper, secondly, the robot has to move towards the selected object and pick it up by exerting a vacuum action with the suction cup, and finally, once the object has been picked up, the robot has to reach the desired Cartesian configuration to carry out the launch of the selected object by reversing the airflow. From the point of view of the code, the task can be rewritten as a design and execution sequence, starting from a series of Cartesian poses of "`geometry_msgs::PoseStamped`" type, suitably saved and loaded during the execution of the code in a specific configuration file (`file pose.YAML`), containing respectively the Cartesian pose of the pneumatic tool, the one of the object and the one for throwing. As already mentioned, the Cartesian pose of the pneumatic hand-tool is given w.r.t. a fixed marker placed near the hand-tool or in a specific place on the shelving, the one of the selected object to be picked up is given as an output of T2.2 while the final configuration of the robot for throwing is selected by a neural network as described in the following.

During code execution, a ROS node will sequentially call the basic service (ROS service) described above to get the human-like path between two consecutive poses. Of course, this code can be easily adapted to launch several objects in sequence, updating the pneumatic tool gripping, vacuum gripping and throwing poses through a ROS Service (ROS Service) which updates the three required poses through a specific map.

Pneumatic circuit controller. The vacuum gripping of the object and the reversal of the air flow to launch the object are carried out using a ROS library for Arduino ("`ros serial Arduino`")

```
ros-drivers/rosserial:
type: git
url: https://github.com/ros-drivers/rosserial.git
version: melodic-devel
```

Inside the pneumatic system controller, an *Arduino UNO* board manages the control part relating to the solenoid valves which allows switching between the gripping and the pushing functions. Through this library, it is possible to interface the ROS package described in the previous section with the pneumatic system for vacuum gripping and throwing of objects. In particular, the vacuum gripping and the launch of the object are carried out by sending an appropriate "`std_msgs::Empty`" type message to the ROS node implemented within the Arduino board, which allows activating the pneumatic valve responsible for generating the vacuum (Venturi pump) and the solenoid valve responsible for selecting the pneumatic circuit that reverses the airflow.

Tuning of the pneumatic circuit parameters. This paragraph describes the neural network that allows to set the several variables and parameters that affect the launch of objects with the pneumatic hand-tool. The main throwing parameters are the outgoing airflow pressure, the opening time of the solenoid valve to regulate the duration of the outgoing airflow, while the variables are the height and angle of the hand-tool. Of course, they are only partially responsible for the ballistics, which is indeed also highly and unpredictably influenced by the shape of the object and its aerodynamic drag, as well as by the point on the surface of the object where the outgoing airflow exerts the pushing force which in turns depends on the picking up phase. The neural network should capture these aspects that are difficult to model.
In order to create the network, a database is collected from several throwing tests of objects chosen within the DARKO set and of which we know the mass, the shape and the size. For this reason, an experimental setup was created to carry out the data collection as follows: the vacuum gripping system with suction cup was fixed to an appropriate support, in such a way as to acquire the ballistics (e.g. the outgoing velocity of the object, the distance of throwing etc.) of the object launched using the commercial software Kinovea[1].
Tests were initially performed with two objects (see figure 8: the *Pollen Filter* (weight < 50 g, dimension: $150x100x45$ mm) and the *Dishwasher Filter* (weight 140 g, dimension: $170x100x100$ mm), which are both in the DARKO set (future works will be done to test other objects).
The Dishwasher Filter and the Pollen filter were repeatedly launched with different pressure values, ranging from 2 to 5 bar, and by setting the height, the angle and the opening time of the solenoid valve. In addition, it was assumed that the object was picked up at approximately the same location (see figure 9).
The collected data are analysed using Kinovea and data are reorganized in such a way as

[1] https://www.kinovea.org/

Figure 8: Pollen filter (a) and Dishwasher filter (b) used for training the neural network and testing the throwing with the pneumatic hand-tool.

Figure 9: Experimental setup used for training the neural network to be used to tune the variables and parameters that control the outgoing airflow.

to associate the throwing distance obtained with a specific pressure value and loaded in the MATLAB Toolbox *Regression Learner*[2]. Within the toolbox, each set of data has been divided into *training* data to train the net, *validation* data to validate the net and, finally, *testing* data to test the goodness and network accuracy. Subsequently, a model of the created neural network (one for each object) was exported and the MATLAB code was appropriately converted into C++ language in order to be easily integrated within the ROS package previously described.

3.3 Preliminary experiments with the pneumatic hand-tool

Our machinery has been tested with the DARKO platform, where the whole software has been integrated. In particular, our trained neural networks have been used for throwing the Dishwasher Filter and Pollen filter at a given distance (0.85 m beyond the working area of the robot) that was different w.r.t. the ones used for training. The neural network, given the characteristics of the object to be launched and the desired distance, outputs the needed level of pressure of the outgoing airflow (with all the other parameters fixed at the training values). We repeated the full sequence of actions several times to tests the repeatability of the hand-tool for picking up the objects and throwing it. We observed that the desired distance was reached with a maximum error of 15 cm. Figures 10 and 11 show some phase of the throwing task with the pneumatic hand-tool.

[2]https://ch.mathworks.com/help/stats/regressionlearner-app.html

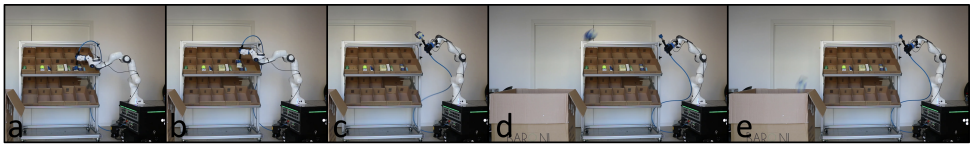


Figure 10: Dishwasher Filter throwing with the pneumatic hand-tool. Pictures (a) and (b) show the vacuuming phase, while pictures (c) - (e) the throwing phase.

Figure 11: Pollen Filter throwing with the pneumatic hand-tool. Pictures (a) and (b) show the vacuuming phase, while pictures (c) - (e) the throwing phase.

4 Conclusion and outlook

This deliverable presents two novel methodologies. The first developed by EPFL, is a generic solution to throwing objects with any fingered gripper and the second developed by UNIPI is a mechanism to throw with the pneumatic hand tool. Both the methods are able to successfully throw objects included in the DARKO project from BOSCH on the DARKO platform with high rate of success.

In order to compute the BRT (Section 2.2.1), the flying dynamics of an object is considered to be known a priori. In the current deliverable, the BRT is derived from projectile motion of the object. In future deliverables, EPFL will utilise the learnt flying dynamics of DARKO objects from state-of-the-art system identification methods [19] or machine learning techniques [9]. The BRT estimation will hence be more accurate leading to improved throwing accuracy. These methods will also be used by UNIPI to rapidly train the neural network used for tuning the several variables and parameters of the pneumatic hand-tool (and not only the outgoing airflow pressure) when used for throwing objects of different weights, sizes and shapes.

The opening delay of the gripper is significant (from 40ms to up to 140 ms) for DARKO objects some of which are susceptible to deformations upon grasping. This delay has a strong effect on the landing position of the object. EPFL has recently developed a method to generate a dynamic throwing trajectory which stays inside the BRT until the object is completely released. In the near future this contribution will be incorporated into the software package. EPFL shall also perform robot implementation with other DARKO objects to obtain a thorough performance evaluation of the methodology proposed.

Specialized end-effectors such as the compact soft articulated parallel elastic wrist developed by UNIPI (details in [17] and in D1.2) may be used to increase the reachable workspace of the Franka Emika Panda by overcoming the throwing capabilities of the cobot. The main purpose of the elastic wrist is indeed to augment the manipulation dexterity and compliance in narrow settings as requested in high density logistic warehouses and in particular within the DARKO project where the manipulator is asked to pick objects from boxes placed on a shelving in a very constrained scenario. It offers increased dexterity upon re-orienting the end-effector by adopting a spherical wrist mechanism. The velocity hedgehog can be modified to take into account the additional task space configurations which are accessible to the Franka Emika Panda with the help of the wrist. Of course, the

elastic wrist can also be exploited by optimally planning the robot actions to temporarily store energy in the spring and realize it at the throwing instant. Similarly to [7] for the hammering task, UNIPI in collaboration with EPFL will develop optimal control strategies to optimally use elasticity for improving the throwing capabilities of the Franka Emika Panda.

References

[1] John Amend, Nadia Cheng, Sami Fakhouri, and Bill Culley. Soft robotics commercialization: Jamming grippers from research to product. *Soft robotics*, 3(4):213–222, 2016.

[2] John R. Amend, Eric Brown, Nicholas Rodenberg, Heinrich M. Jaeger, and Hod Lipson. A positive pressure universal gripper based on the jamming of granular material. *IEEE Transactions on Robotics*, 28(2):341–350, 2012.

[3] Giuseppe Averta, Danilo Caporale, Cosimo Della Santina, Antonio Bicchi, and Matteo Bianchi. A technical framework for human-like motion generation with autonomous anthropomorphic redundant manipulators. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3853–3859, 2020.

[4] Marco Baracca, Paolo Bonifati, Ylenia Nisticò, Vincenzo Catrambone, Gaetano Valenza, A. Bicchi, Giuseppe Averta, and Matteo Bianchi. Functional analysis of upper-limb movements in the cartesian domain. In Diego Torricelli, Metin Akay, and Jose L. Pons, editors, *Converging Clinical and Engineering Research on Neurorehabilitation IV*, pages 339–343, Cham, 2022. Springer International Publishing.

[5] Lars Berscheid and Torsten Kröger. Jerk-limited real-time trajectory generation with arbitrary target states. *Robotics: Science and Systems XVII*, 2021.

[6] Yizhi Gai, Yukinori Kobayashi, Yohei Hoshino, and Takanori Emaru. Motion control of a ball throwing robot with a flexible robotic arm. *International Journal of Computer and Information Engineering*, 7(7):937–945, 2013.

[7] Manolo Garabini, Andrea Passaglia, Felipe Belo, Paolo Salaris, and Antonio Bicchi. Optimality principles in variable stiffness control: The vsa hammer. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3770–3775, 2011.

[8] Harshit Khurana, Michael Bombile, and Aude Billard. Learning to hit: A statistical dynamical system based approach. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9415–9421. IEEE, 2021.

[9] Seungsu Kim and Aude Billard. Estimating the non-linear dynamics of free-flying objects. *Robotics and Autonomous Systems*, 60(9):1108–1122, 2012.

[10] Jens Kober, Katharina Muelling, and Jan Peters. Learning throwing and catching skills. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5167–5168. IEEE, 2012.

[11] Yang Liu, Aradhana Nayak, and Aude Billard. A solution to adaptive mobile manipulator throwing. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1625–1632. IEEE, 2022.

[12] Ferenc Lombai and Gábor Szederkényi. Throwing motion generation using nonlinear optimization on a 6-degree-of-freedom robot manipulator. In *2009 IEEE International Conference on Mechatronics*, pages 1–6. IEEE, 2009.

[13] Kevin M Lynch and Matthew T Mason. Dynamic nonprehensile manipulation: Controllability, planning, and experiments. *The International Journal of Robotics Research*, 18(1):64–92, 1999.

[14] Matthew T Mason and Kevin M Lynch. Dynamic manipulation. In *Proceedings of 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'93)*, volume 1, pages 152–159. IEEE, 1993.

[15] Hideyuki Miyashita, Tasuku Yamawaki, and Masahito Yashima. Control for throwing manipulation by one joint robot. In *2009 IEEE International Conference on Robotics and Automation*, pages 1273–1278. IEEE, 2009.

[16] Wataru Mori, Jun Ueda, and Tsukasa Ogasawara. 1-dof dynamic pitching robot that independently controls velocity, angular velocity, and direction of a ball: Contact models and motion planning. In *2009 IEEE International Conference on Robotics and Automation*, pages 1655–1661. IEEE, 2009.

[17] Francesca Negrello, Sariah Mghames, Giorgio Grioli, Manolo Garabini, and Manuel Giuseppe Catalano. A compact soft articulated parallel wrist for grasping in narrow spaces. *IEEE Robotics and Automation Letters*, 4(4):3161–3168, 2019.

[18] Alexander Pekarovskiy and Martin Buss. Optimal control goal manifolds for planar nonprehensile throwing. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4518–4524. IEEE, 2013.

[19] Thomas B Schön, Adrian Wills, and Brett Ninness. System identification of nonlinear state-space models. *Automatica*, 47(1):39–49, 2011.

[20] Taku Senoo, Akio Namiki, and Masatoshi Ishikawa. High-speed throwing motion based on kinetic chain approach. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3206–3211. IEEE, 2008.

[21] Avishai Sintov and Amir Shapiro. A stochastic dynamic motion planning algorithm for object-throwing. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2475–2480. IEEE, 2015.

[22] Andy Zeng, Shuran Song, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser. Tossingbot: Learning to throw arbitrary objects with residual physics. *IEEE Transactions on Robotics*, 36(4):1307–1319, 2020.

A Appendix

A.1 Throwing Feasibility Problem

Let $fkine(\cdot)$ denote the robot forward kinematics, and $\overrightarrow{EB}_z$ the projection of $\overrightarrow{EB}$ along vertical direction z. The throwing Feasibility Problem (TFP) is as follows:

$$
\text{Find } \left\{ r, z, \dot{r}, \dot{z}, \overrightarrow{EB}, \overrightarrow{AB}, q, \dot{q} \right\}
$$

$$
\text{such that: } \begin{cases}
r = \left\| \overrightarrow{EB}_{xy} \right\| \\
z = -\overrightarrow{EB}_{z} \\
\overrightarrow{EB} = \overrightarrow{AB} - \overrightarrow{AE} \\
\overrightarrow{AE} = fkine(q) \\
\vec{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} \dot{r} \frac{\overrightarrow{EB}_{xy}}{\left\| \overrightarrow{EB}_{xy} \right\|} \\ \dot{z} \end{bmatrix} \\
\dot{q} = J^{\dagger}(q)\vec{v} \\
q_{\min} \leq q \leq q_{\max} \\
\dot{q}_{\min} \leq \dot{q} \leq \dot{q}_{\max} \\
f_{BRT}(r, z, \dot{r}, \dot{z}) \leq 0
\end{cases}
\tag{TFP}
$$

A.2 Algorithm for Velocity Hedgehog (VH) generation

Algorithm 1: Algorithm to generate velocity hedgehog

Input : robot model with forward kinematics and differential forward kinematics $robot$
Output : max_z_phi_gamma, q_z_phi_gamma
Data : robot joint position limit (q_{min}, q_{max}),
robot joint velocity limit $(\dot{q}_{min}, \dot{q}_{max})$,
joint grid size Δq,
velocity hedgehog grids Z, Φ, Γ

/* Build robot dataset $\mathscr{X}$ */
$\mathscr{Q} \leftarrow ComputeMesh(q_{min}, q_{max}, \Delta q)$
/* Filter out q with small singular value */
$\mathscr{X} \leftarrow FilterBySinguarValue(\mathscr{Q})$
/* Group data by Z */
$\{\mathscr{X}_z\} \leftarrow GroupBy(\mathscr{X}, Z)$
/* Initialize velocity hedgehog */
max_z_phi_gamma $\leftarrow zeros([\#Z, \#\Phi, \#\Gamma])$
q_z_phi_gamma $\leftarrow arrays([\#Z, \#\Phi, \#\Gamma, \#joints])$
/* Build velocity hedgehog */
for $[z, \phi, \gamma, data] \in Z \times \Phi \times \Gamma \times \mathscr{X}_z$ **do**
  /* Get max. speed along (ϕ,γ) at joint configuration q */
  $res \leftarrow LP(\phi, \gamma, data.q, robot, \dot{q}_{min}, \dot{q}_{max})$
  if $res > max\_z\_phi\_gamma(z, \phi, \gamma)$ **then**
    max_z_phi_gamma$(z, \phi, \gamma) \leftarrow res$
    q_z_phi_gamma$(z, \phi, \gamma) \leftarrow data.q$
  end
end
return max_z_phi_gamma, q_z_phi_gamma

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101017274