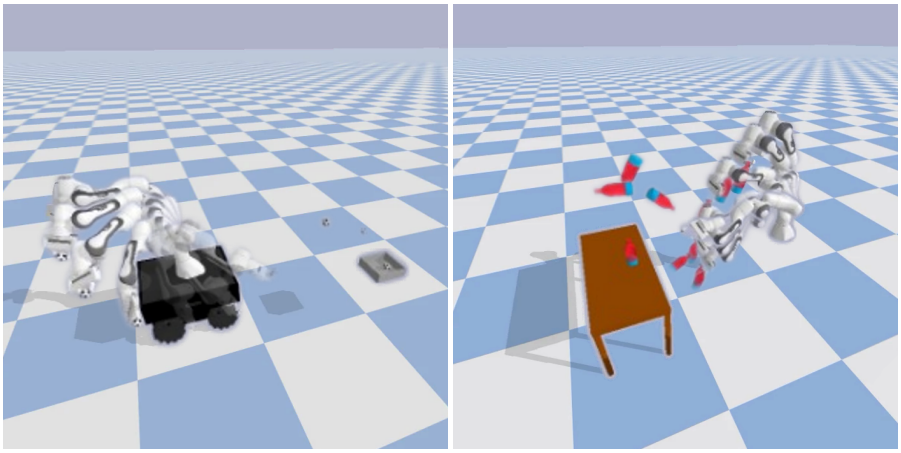
# Velocity Hedgehog for Fixed-base Manipulator Throwing




Semester Project
Autumn Semester 2022-2023
in LASA Lab - EPFL
by Jiangfan Li

Assistants:

Yang Liu
Dr. Aradhana Nayak



Lausanne, EPFL, 2023

# Contents

**Introduction** **1**

**1 Velocity Hedgehog** **3**
1.1 Throwing Problem . . . 3
1.2 Velocity Hedgehog . . . 5

**2 From mobile to fixed-base** **9**
2.1 Direct Filter . . . 9
2.2 Expand the hedgehog . . . 10
2.3 Summary . . . 11

**3 Algorithm Acceleration** **13**
3.1 Iterative Search . . . 13
3.2 Sorted Search . . . 14
3.3 Vectorization . . . 14
3.4 Performance evaluation . . . 16

**4 Towards Desired Landing Orientation** **19**
4.1 Projectile Motion with Orientation . . . 20
4.2 Sim-to-Real and Real-to-Sim . . . 23

**Conclusion** **25**

# Introduction

Throwing is a dynamic manipulation strengthening a given robotic manipulator by causing motions outside its workspace [4]. This ability is highly desirable for industrial applications. However, because of the nonlinear relationship between robot joint space and throwing task space, throwing problems are generally difficult to solve efficiently.

As a bridge between joint space and task space, velocity hedgehog [3] is a hedgehog-shaped data structure that was proposed for dynamic manipulation tasks, e.g. robotic throwing. It stores a dictionary of appropriate robot joint configurations that can generate high Cartesian velocity along query direction under limited joint velocity for throwing. Combining with an online search algorithm, feasible and valid throwing configurations for mobile manipulators can be generated efficiently.

The existing algorithm heavily relies on the spatial invariance property of the mobile manipulator's omnidirectional wheel and planar flying dynamics of a point mass. Under these assumptions, the mobile manipulator throwing problem can be described by a compact set of independent variables, and hence the size of velocity hedgehog is manageable and efficient algorithm can be devised.

However, for the sake of generic robotic throwing, the assumptions need to be relaxed and the set of independent variables in order to describe the throwing system will inevitably be enlarged. For example, for fixed-based manipulator throwing, to account for the lost spatial invariance of the mobile base, a naive extension based on the current method would add two extra keys to the dictionary, representing the end-effector's horizontal positions. In this way, the throwing dictionary would be indexed by a 5-dimensional mesh (instead of a 3-dimensional one in the mobile manipulator throwing case). As the size of the velocity hedgehog and hence the online query time depends exponentially on the number of indexes. Higher dimensional velocity hedgehog, brought by more generic throwing problems, might lose the intended computational efficiency. In order to investigate the applicability of velocity hedgehog for more general throwing setups, in this work, we focus on the following problem:

*Throwing a point mass to desired position with a fixed-base manipulator*

Alongside solving this problem, we also contribute to proposing a better strategy for online velocity hedgehog matching, which speeds up the online solver 24 times.

Furthermore, we analyze the throwing problem with constraints on landing orientation and propose a solution with ideal settings.

In the following pages, this report will first introduce the throwing problem and explain the original solution [3] for the mobile manipulator in chapter 1. Chapter 2 shows how this solution adapts to the fixed-base manipulator. Utilizing the fact that the first joint of the Panda manipulator is vertical, only one extra key representing the end-effector's distance to the base is added upon the solution for mobile manipulation throwing as done in [3]. Despite this insight, the enlarged hedgehog of the fixed-base case with one extra dimension results in a higher computational cost. Trying to address this issue, in chapter 3, two methods for algorithm acceleration are applied and compared. Finally, a more difficult task of throwing towards desired landing orientation is investigated in chapter 4.

The project code is avaliable on Github https://github.com/RLi43/mobile-throwing. The code for chapter 2 and chapter 3 is in default branch `fixed-base`, and the code for chapter 4 is in branch `orientation`.

# 1 Velocity Hedgehog

## 1.1 Throwing Problem

To throw an object into a box, the first question we need to ask is that, under which motion states will the object drop into the box ultimately? One solution for the this 2-dimension throwing problem is Backward Reachable Tube(BRT) which can also be applied to arbitrary dynamics, allowing for the possible future extension to more complex environment.

We denote the flying state as $x = [r, z, \dot{r}, \dot{z}]^\top$. Given the flying dynamics described by a first-order differential equation $\dot{x} = f(x)$, $x \in \mathbb{R}^4$, the flying trajectories of $f$ starting from state $x_0$ is denoted as $\zeta_{f,x_0}(t) : [0, +\infty] \to \mathbb{R}^4$.

Given the flying dynamics $f$ and a landing target set $\mathscr{X}_l \subseteq \mathbb{R}^4$, describing the allowed landing position slack and allowed range of landing velocities, BRT is a set of beginning motion states that will be in $\mathscr{X}_l$ at some point.

$$BRT_{f,\mathscr{X}_l} = \{x_0 \,|\, \exists\, t \geq 0, \zeta_{f,x_0}(t) \in \mathscr{X}_l\}.$$

Starting from the desired landing pose, the valid motion states can be calculated by solving ordinary differential equations backward in time.

Here comes another question, is the robot capable to move the object to the BRT? Assuming the gripper can hold the object tightly and there's no relative motion between the gripper and the object, this question becomes that, is the end effector able to reach these states? Due to joint limits, the pose and the possible motion state of the end effector are confined within a small range. Humans can throw a golf-ball-size object with a release speed of about 20m/s [5] which is far beyond the limit of normal collaborative robots, for example, Panda robot we will use in the following experiments has a maximum translational velocity of 1.7 m/s.

When the robot is mobile – the relative position of the target from the robot is flexible – it depends mainly on the maximum velocity the end effector can archive in a specific height,

pitch angle, and yaw angle.

The velocity $v$ and the yaw angle $\gamma$ of the motion state $x$ can be computed from the previous representation as

$$v = \|(\dot{r}, \dot{z})\|_2$$
$$\gamma = \arctan\left(\frac{\dot{z}}{\dot{r}}\right)$$

As shown in Figure 1.1, we assign the base of the robot(A) as the coordination origin and denote the position of the box(B) and the end effector(E) in horizontal polar coordination and height, as $p_B = [r_B, \beta, z_B]^T$ and $p_E = [r_E, \alpha, z_E]^T$ respectively. The end effector's velocity pitch angle is denoted as $\gamma_E$ and velocity yaw angle as $\varphi_E$.

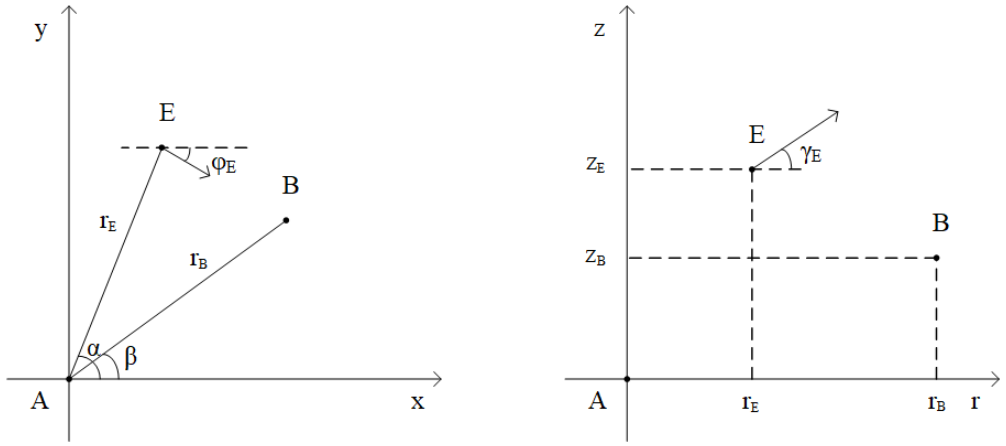



Figure 1.1: The position of the robot end effector and the box and the velocity of the end effector

One valid flying state $x$ should satisfy these geometric relationships:

$$\begin{cases} z = z_E - z_B \\ r^2 = b^2 + a^2 - 2abcos(\alpha - \beta) \\ \gamma = \gamma_E \\ -2arcos(\alpha - \varphi_E) = a^2 + r^2 - b^2 \end{cases}$$

And the release speed $v$ should not exceed the maximum possible speed of the end effector with joint configuration $q$ can archive at the direction $(\varphi_E, \gamma_E)$, which is denoted as $v_{max}(q, \varphi_E, \gamma_E)$.

$$v \leq v_{max}(q, \varphi_E, \gamma_E)$$

We can acquire $v_{max}(q, \varphi_E, \gamma_E)$ by solving the equation LP

$$\begin{aligned} & \max \ s \\ s.t. \quad & \dot{q}_{min} \leq J^{\dagger}(q)\vec{v} \leq \dot{q}_{max} \\ & \vec{v} = s \cdot \begin{bmatrix} \cos(\gamma_E) \cdot \cos(\varphi_E) \\ \cos(\gamma_E) \cdot \sin(\varphi_E) \\ \sin\gamma_E \end{bmatrix} \end{aligned} \tag{LP}$$

where $J^{\dagger}(q)$ is the pseudo-inverse of the Jacobian matrix at the joint configuration $q$.

## 1.2 Velocity Hedgehog

To solve equation 1.1 rapidly to get initial guesses online, the velocity hedgehog, a data structure, was proposed to store intermediate data offline, namely, the maximum velocities $v_m(z, \gamma, \phi)$ the robot can archive at different heights, yaw and pitch angles and the corresponding configuration $q_m(z, \gamma, \phi)$. Algorithm 1 shows how to generate a velocity hedgehog from the given robot. And Figure 1.2 shows three typical maximum velocity distributions at different heights. The blue dot and the red rod represent the maximum speed at certain throw angles(yaw and pitch).

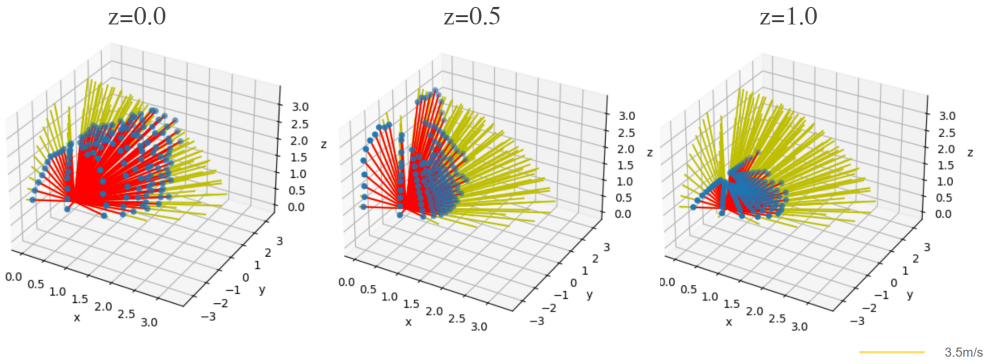



Figure 1.2: Velocity Hedgehog at different height

For a mobile manipulator, the problem can be decoupled into two problems – first find the possible candidates $\{q_{candidate}, \varphi_{candidate}, x_{candidate}\}$ with velocity hedgehog and BRT states computed offline, which is illustrated in Algorithm 2, and then compute where the robot base should be whose cost should be negligible.

The overall framework of this velocity hedgehog solution is illustrated in Figure 1.3. This framework is designed to be generic for any mobile manipulator and any point-mass planar flying dynamics. In this report, we use Panda robot and BRT data generated by gravity-only flying dynamics.

**Algorithm 1:** Algorithm to get velocity hedgehog [3]

**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 $\Delta q$,
velocity hedgehog grids $Z, \Phi, \Gamma$

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

**Algorithm 2:** Algorithm to get initial guesses [3]

**Input** : BRT dataset $\mathcal{X}_{BRT}$, velocity hedgehog grids $Z, \Phi, \Gamma$
**Output** : Set of initial guess for joint configuration and throwing configuration $(q, \varphi, x)$

/* Group data by $Z, \Gamma$ */
$\mathcal{X}_{z,\gamma} \leftarrow GroupBy(\mathcal{X}_{BRT}, Z, \Gamma)$
$q\_guess, \phi\_guess, x\_guess \leftarrow empty$
$idxs \leftarrow where(max\_z\_phi\_gamma > \mathcal{X}_{z,\gamma})$
$q\_guess = q\_z\_phi\_gamma[idxs]$
$\phi\_guess = \Phi[idxs]$
$x\_guess = \mathcal{X}_{z,\gamma}[idxs]$
**return** $q\_guess, \phi\_guess, x\_guess$

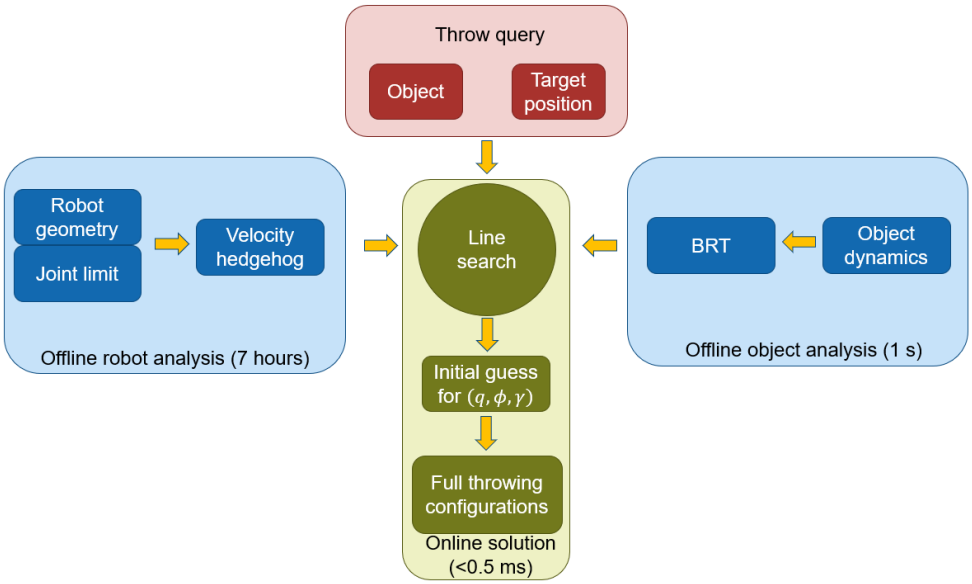



Figure 1.3: Velocity Hedgehog at different height[3]

# 2 From mobile to fixed-base

One might believe that the fixed-base case is easier than the mobile one at the first glance as it has fewer degrees of freedom. But through our previous analysis, the fixed-base manipulator loses exactly those degrees of freedom that simplify the mobile throwing problem. In other words, for the fixed-base manipulator, the relative position to the box, $r_b$ and $\beta$, are fixed, which means we can not decouple the problem as we did in the mobile case. We have to consider more constraints either online or offline.

## 2.1 Direct Filter

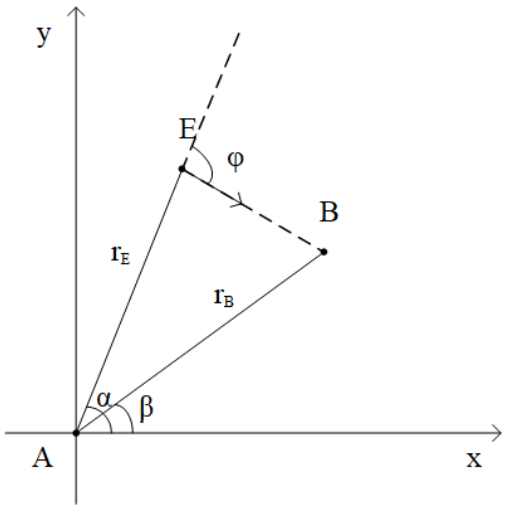



Figure 2.1: Illustration for filter conditions

One of the most intuitive ways is to search for solutions from the previous algorithm that don't ask the robot to move. Appending a filter to the previous solution can easily give us some results. The filter conditions[1] are:

[1] In practice we add some slack factors for our discrete data

- the end effector throw towards the box

$$\varphi = atan2d(r_B \sin(\beta - \alpha), r_B \cos(\beta - \alpha) - r_E)$$

- the flying distance is exactly the same as the BRT states

$$z^2 = r_E^2 + r_B^2 - 2 r_E r_B \cos(\beta - \alpha)$$

However, we are not satisfied with it since there are always few results after filtering (Figure 2.2 left).

## 2.2 Expand the hedgehog

Another idea is to expand the hedgehog by introducing the position of the end effector, $r_E$ and $\alpha$, into the parameter set of the velocity hedgehog as well. However, it's likely to have a huge matrix as it grows exponentially with the number of dimensions, which is hard to answer a query in real-time. Practically, the original hedgehog with a grid listed in Table 2.1 has a size of 26.7KB. If we add two more dimensions of the position using similar graduation, that is $X = [-1.2 : 0.05 : 1.2]$, $Y = [-1.2 : 0.05 : 1.2]$, then the size of the new hedgehog will increase greatly to 65.6MB, which will severely impair the query efficiency.

Table 2.1: Velocity hedgehog grid mesh

| Parameter | Range | Count |
|---|---|---|
| Z | [0:0.05:1.2] | 25 |
| Φ | [-90:15:90] | 13 |
| Γ | [20:5:70] | 11 |

To reduce the size of the hedgehog, we can utilize a condition that holds in common situations – the first joint of the robot is perpendicular to the ground. With this assumption, only one parameter, the distance of the end effector from the base $d$, is needed to record the position of the end effector. With a distance range $D = [0 : 0.05 : 1.2]$, the size of the hedgehog is about 495KB which is relatively reasonable for the query step. Algorithm 3 shows how to generate this new hedgehog.

The distance $d$ and the yaw angle $\varphi_E$, along with the given box position $p_B = (r_B, \beta, z_B)$, determine the flying distance $r$ and the new first joint position $q_0'$.

$$r = \sqrt{r_B^2 - d^2 + (d\cos\varphi)^2} - d\cos\varphi$$

$$q_0' = q_0 + \frac{r_E - r\cos\varphi}{b} \cdot sign(r_E) + \beta - \alpha$$

**Algorithm 3:** Algorithm to get velocity hedgehog for fixed-base

**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 $\Delta q$,
velocity hedgehog grids $D, Z, \Phi, \Gamma$

$\mathcal{Q}_{raw} \leftarrow ComputeMesh(q_{min}, q_{max}, \Delta q)$
$\mathcal{Q}_f \leftarrow FilterBySinguarValue(\mathcal{Q}_{raw})$
$\{\mathcal{Q}\} \leftarrow GroupBy(\mathcal{Q}_f, Z, D)$
vel_max $\leftarrow zeros([\#Z, \#\Phi, \#\Gamma])$
q_max $\leftarrow arrays([\#Z, \#\Phi, \#\Gamma, \#joints])$
**for** $[d, z] \in D \times Z$ **do**
  **for** $[\phi, \gamma, q] \in \Phi \times \Gamma \times \mathcal{Q}_z(z)$ **do**
    $res \leftarrow LP(\varphi, \gamma, q, robot, \dot{q}_{min}, \dot{q}_{max})$
    **if** $res > vel\_max(d, z, \varphi, \gamma)$ **then**
      vel_max$(d, z, \varphi, \gamma) \leftarrow res$
      q_max$(d, z, \varphi, \gamma) \leftarrow q$
    **end**
  **end**
**end**
**return** vel_max, q_max

## 2.3 Summary

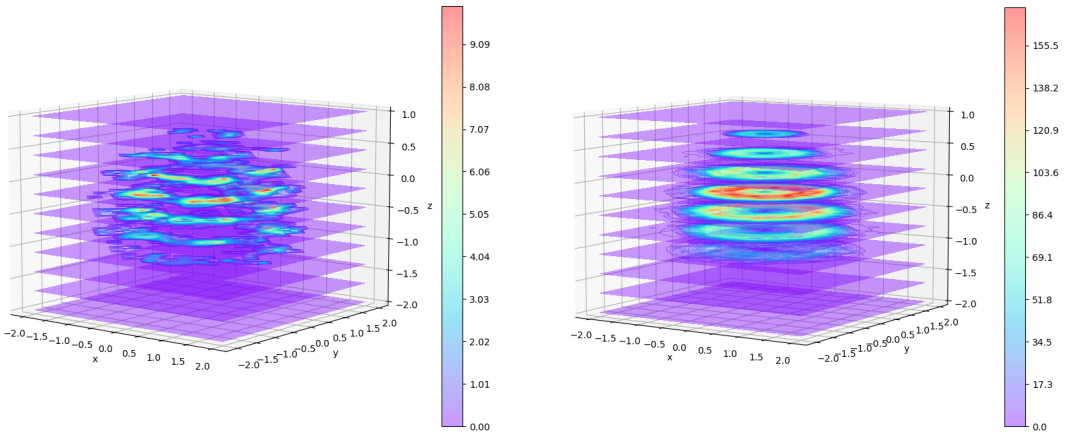



Figure 2.2: Number of the solutions found for different target box positions by direct filtering(left) and expanded hedgehog(right)

As shown in Figure 2.2, there are much more solutions found by the expanded hedgehog

compared to the direct filtering method. We can imagine that, given a target, it's of high possibility that one can't find a valid throwing configuration with the naive direct filtering method. This is much less likely to happen with the expanded hedgehog.

One potential problem of the algorithm is that the error comes from discretization will be enlarged during the extra computation step for the fixed-base case, though the deviation is acceptable for now with the parameter set we use in experiments and we don't need to give up either the accuracy or the efficiency of the solution.

# 3 Algorithm Acceleration

As we expanded the hedgehog to adapt the original solution to the fixed-base manipulator which slows down the online query step, it becomes more urgent to optimize the online query code in order to enable better reactiveability.

The bottleneck of the query is the matching step(specifically, line 3 in Algorithm 2) to find valid pairs of motion state from BRT and configuration from velocity hedgehog. Here, valid means the speed of the motion state from BRT is smaller than the maximum speed from the hedgehog.

## 3.1 Iterative Search

The original solution iterates every $z, \varphi, \gamma$ through triple nested loop(Algorithm 4).

**Algorithm 4:** Original Match Step

**Input :** BRT dataset grouped by z, $\gamma, \mathscr{X}$,
velocity hedgehog $max\_vel$ with grids $Z, \Phi, \Gamma$,
velocity threshold, $threshold$

**Output :** index pairs of the valid throw configuration $idxs$

```
1 for i_z ∈ #Z do
2     for i_γ ∈ #Γ do
3         for i_φ ∈ #Φ do
4             i_valid = where(𝒳(Z[i_z], Γ[i_γ]) < max_vel(Z[i_z], Φ[i_φ], Γ[i_γ] − threshold)
5             idxs ← idxs + [i_z, i_γ, i_valid]
6         end
7     end
8 end
9 return idxs
```

## 3.2 Sorted Search

Inspecting the current solution we can find that it can be understood in another way: at each $z$ and $\gamma$, there are a bunch of the candidate motion states and an array from velocity hedgehog with different yaw angle $\varphi$ (Figure 3.1). In other words, we need to find all the pairs of velocity hedgehog entry and object motion state whose robot feasible velocity is smaller than the object motion velocity. Therefore, an insight is that some comparisons can be avoided by sorting these two sets first.

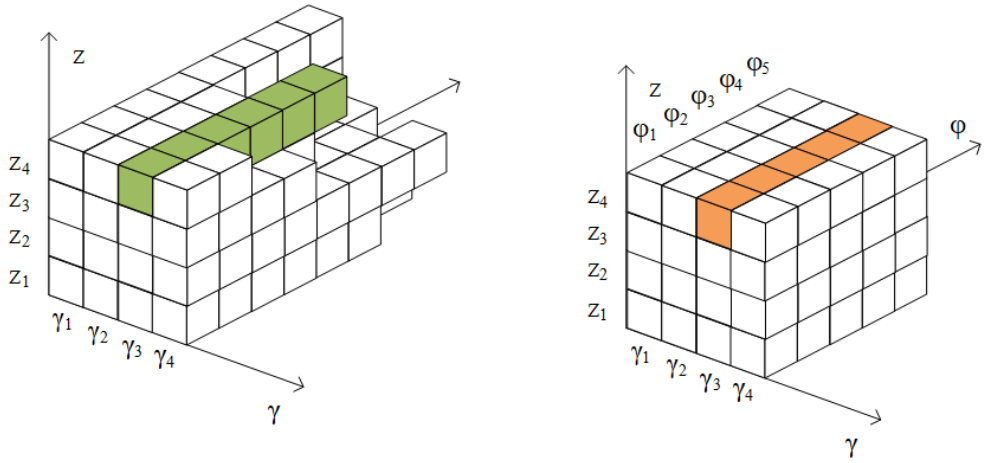



Figure 3.1: The two arrays at certain $z$ and $\gamma$

Algorithm 5 utilizes the sorted arrays to avoid redundant comparisons. Specifically, it starts with the minimum robot velocity, and records the maximum BRT velocity that's smaller than it. For the next robot velocity it can start comparison from the previous maximum BRT velocity instead of the whole array cutting down the comparison times. It runs until the maximum robot velocity or that the current robot velocity is lager than the maximum BRT velocity. Therefore, it runs comparison for at most the sum of the arrays' length times.

Assuming the lengths of two array is $m$ and $n$ respectively, the original loop of the comparison has a time complexity of $\Theta(m \cdot n)$[1]. Using sorted arrays, there are at maximum $m+n$ times comparison. Considering the cost of sorting, the overall time complexity is $\Theta(m \log m) + \Theta(n \log n) + \mathcal{O}(m+n) = \Theta(m \log m) + \Theta(n \log n)$. Knowing both m and n's values are around 10, the sorted search should accelerate the query step theoretically.

## 3.3 Vectorization

Vectorization is a programming technique that deals with entire arrays instead of individual elements. The programming language we used for experiments, Python, is a dynamically

[1]The notation $f(n) = \Theta(g(n))$ means the function $g(n)$ is the asymptotically tight bound of the function $f(n)$. The symbol $\mathcal{O}$ stands for the asymptotically upper bound

**Algorithm 5:** Sorted Search

**Input :** BRT dataset grouped by z, $\gamma, \mathscr{X}$,
velocity hedgehog $max\_vel$ with grids $Z, \Phi, \Gamma$,
velocity threshold, $threshold$

**Output :** index pairs of the valid throw configuration $idxs$

**for** $i_z \in \#Z$ **do**
  **for** $i_\gamma \in \#\Gamma$ **do**
    **if** $\mathscr{X}(z,\gamma)$ *is not empty* **then**
      /* sort two arrays and record the indice of the sorted arrays */
      $v_{ind} = argsort(max\_vel(z,\gamma))$
      $b_{ind} = argsort(\mathscr{X}(z,\gamma))$
      $p_b = 0$
      **for** $i_v \in \#max\_vel(z,\gamma)$ **do**
        /* Find the index of the maximum BRT velocity in the sorted array that's less than max_vel */
        $p_a = bisect\_left(\mathscr{X}(z,\gamma)[b_{ind}[p_b : end]],$
        $max\_vel(z,\gamma)(i_v) - threshold)$
        **if** $p_a > 0$ **then**
          $idx_x = \mathscr{X}(z,\gamma)[b_{ind}[p_b : p_b + index]] \times (\#max\_vel(z,\gamma) - i_v)$
          $idx_q = [i_z, i_\gamma, v_{ind}[i_v : end]]$
          $idx \leftarrow idx + (idx_x, idx_q)$
          **if** $p_a \geq \#\mathscr{X}(z,\gamma) - p_b$ **then**
            /* even the smallest max_vel is bigger than the brt vel */
            break
          **end**
        **end**
        **else**
          break
        **end**
      **end**
    **end**
  **end**
**end**
**return** idxs

typed language. It infers the properties of the variables during runtime. NumPy library [1] provides `ndarray`, a fixed-size homogeneous n-dimensional array object, with underlying operations done using methods including but not confined to using efficient C operations, parallel processing and dedicated optimization for specific mathematical operations. This allows for vectorization.

One problem to apply the vectorization is the variable length of the BRT bunches (Figure 3.2).

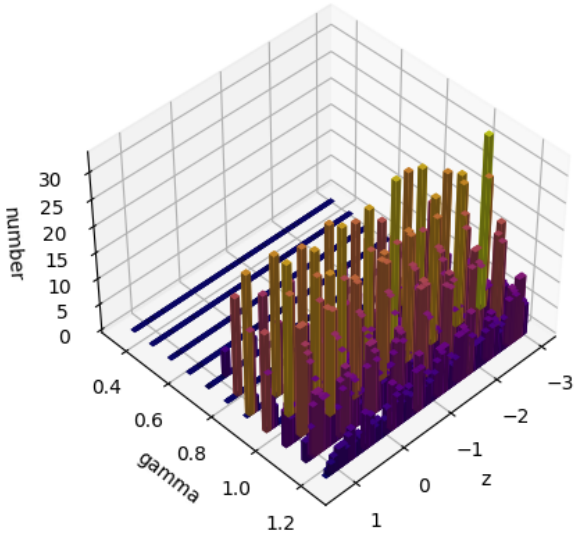



Figure 3.2: The length of a typical BRT bunches

Our solution is to construct a matrix with a length of the maximum length of the bunches, and fill the empty cells with `numpy.nan`, a special value representing 'not a number', with which any comparison will return `False`. Therefore, the empty cells will be automatically filtered out in the online matching phase.

## 3.4 Performance evaluation

The query time of each method is shown in Figure 3.3. Apparently, the vectorization method is the fastest one and is about 24 times faster than the original one, allowing for more adaptive control in real-time.

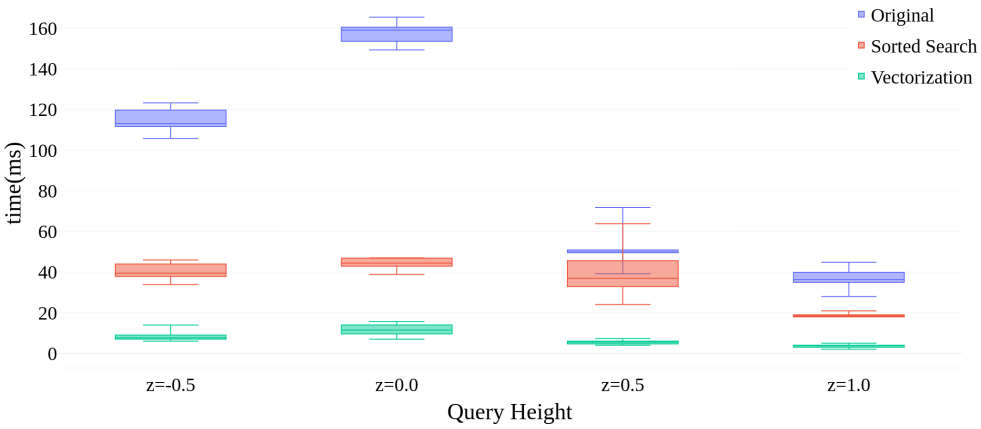



Figure 3.3: The query time of three methods

# 4 Towards Desired Landing Orientation

In addition to solving the throwing problem with target landing position and translational velocity, we also explore the control of the landing orientation.

Figure 4.1: Example scenes: dart, bottle juggling and pitch-pot

There are some common scenes requiring the control of the landing orientation after throwing, for example, darts, bottle juggling and pitch-pot(Figure 4.1[1]). An observation of these tasks is that we are not targetting arbitrary landing orientation in SO(3), but are only interested in the orientations perpendicular to the throwing plane. In this regard, we choose bottle flipping challenge as our target task. Bottle flipping challenge was a trend attempting to throw a half-filled water bottle into the air so that it rotates and lands upright on its base. We choose this task not only for its impressive demonstration, but also because the half-filled bottle has a low center-of-mass, and hence is robust to landing orientation error. However, it's still very challenging – the bottle may bounce or slip after landing; the bottle with water can be too heavy to manipulate; the angular speed to finish the rotation in short flying time may exceed the robot hardware limits; the fluid inside the bottle makes it hard to simulate or predict the

[1] Source: https://ci.jumia.is/unsafe/fit-in/680x680/filters:fill(white)/product/58/953351/1.jpg?4266, https://resources.chainbox.io/1/pegani/public/pim/5b80bd05-0c8a-46c4-b313-abc86045efd3/790_default.JPG, http://img.mp.itc.cn/upload/20170417/7317e0d854a9489bb324a92a78a2ff36_th.jpeg

motion state function, etc.

We try to solve this task by starting with some ideal assumptions. Since it is quite hard to search for a solution under both constraints on landing orientation and position, the following analysis will only focus on landing orientation.

## 4.1 Projectile Motion with Orientation

First, we use the ideal projectile motion model to estimate the constraints for the release states. Compared to the models we use in the previous chapters, this time the one orientation perpendicular to the throwing plane, $\theta$, is also modeled. For simplicity, we assume that there is no rolling friction. In other words, the rotational speed $\omega$ is constant when the object is in the air.

$$\omega = \frac{d\theta}{dt}$$

For a given release joint configuration $q_0$, the position of the end effector $z_E$ and $r_E$ and its pitch angle $\theta_E$ can be computed by robot's forward kinematics.

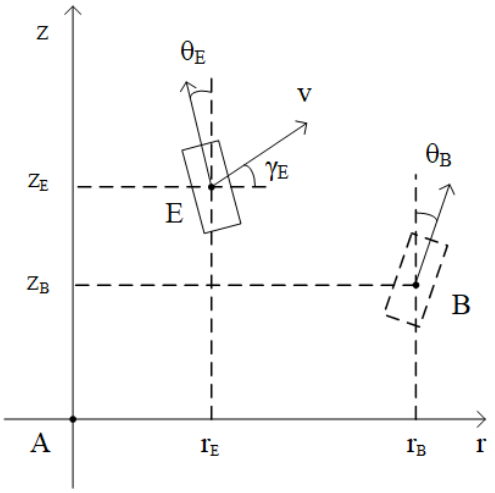



Figure 4.2: Throwing plane for projectile motion with orientation

Given a desired landing height $z_B$ and pitch angle $\theta_B$, the motion the object should complete in the air should be:

$$\Delta z = z_B - z_E$$
$$\Delta\theta = \theta_B - \theta_E$$

For a better visual effect, $\Delta\theta$ is clipped to have a absolute value that is in between $\pi$ to $2\pi$ so that it will at least rotate half a turn in the air. The typical maximum $\omega$ the robot can archive is about 5 rad/s, and the typical flying time is about 0.7s (when the landing plane is as high as the robot base), so it is almost impossible for the robot to flip the bottle around.

We denote the flying time as $t_{fly}$, which is determined by release velocity on the y-axis and the height drop $\Delta z$. The rotational speed $\omega$ and the required joint velocity are determined afterward.

$$t_{fly}(v_z) = -\frac{v_z}{g} + \sqrt{\left(\frac{v_z}{g}\right)^2 + 2\left(\frac{\Delta z}{g}\right)}$$

$$\omega(v_z) = \frac{\Delta\theta}{t_{fly}(v_z)}$$

$$\dot{q} = J^{\dagger}(q_0) \cdot [v_r, 0, v_z, 0, \omega, 0]^T$$

Therefore, given a release joint position, we have to find the possible $v_x$ and $v_y$ that meet the joint velocity limits.

Our way to acquire one solution is to solve eq. (4.1), whose infeasibility indicates that there is no solution for the given release joint position $q$.

$$\begin{aligned} & \min \ \|\dot{q}\|_1 \\ s.t. \quad & \dot{q}_{min} \le J^{\dagger}(q)[v_r, 0, v_z, 0, \omega, 0]^T \le \dot{q}_{max} \\ & 0 \le v_r \\ & 0 \le v_z \\ & \omega = \Delta\theta / t_{fly}(v_z) \end{aligned} \tag{4.1}$$

To validate the solution, we build a 'solid water bottle' mimicking a partially filled Evian water bottle. To avoid fluid simulation and as a result making the task too complex, the 'water' inside the bottle is built as a solid block with a similar density as water. We call this a heavy base bottle. The center of mass is close to the bottom, providing some tolerance for rotation error, which may come from the biased inertia or the error of the pitch angle due to unmodelled air drag. They are shown in Figure 4.3.

Figure 4.4 shows an example trajectory we found in simulation to flip the bottle with upright landing posture.

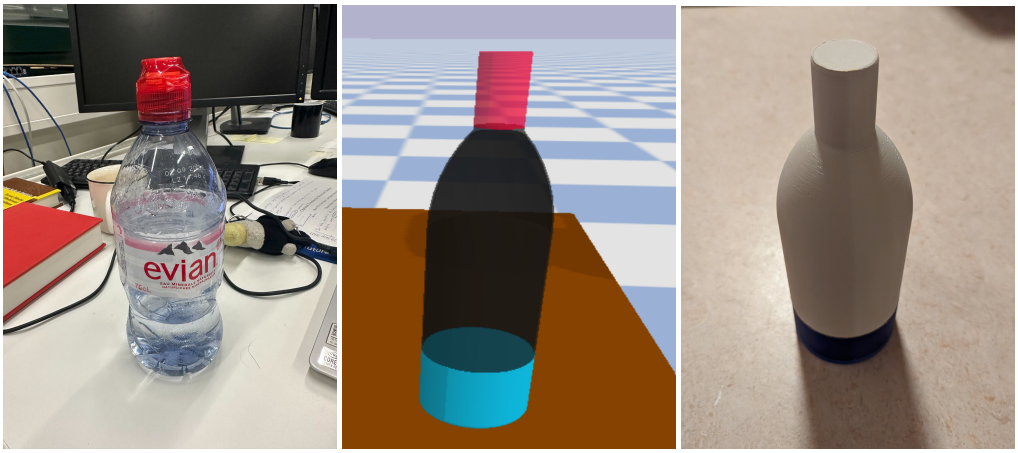



Figure 4.3: The real Evian bottle, the heavy base bottle model for simulation, and the 3D printed model

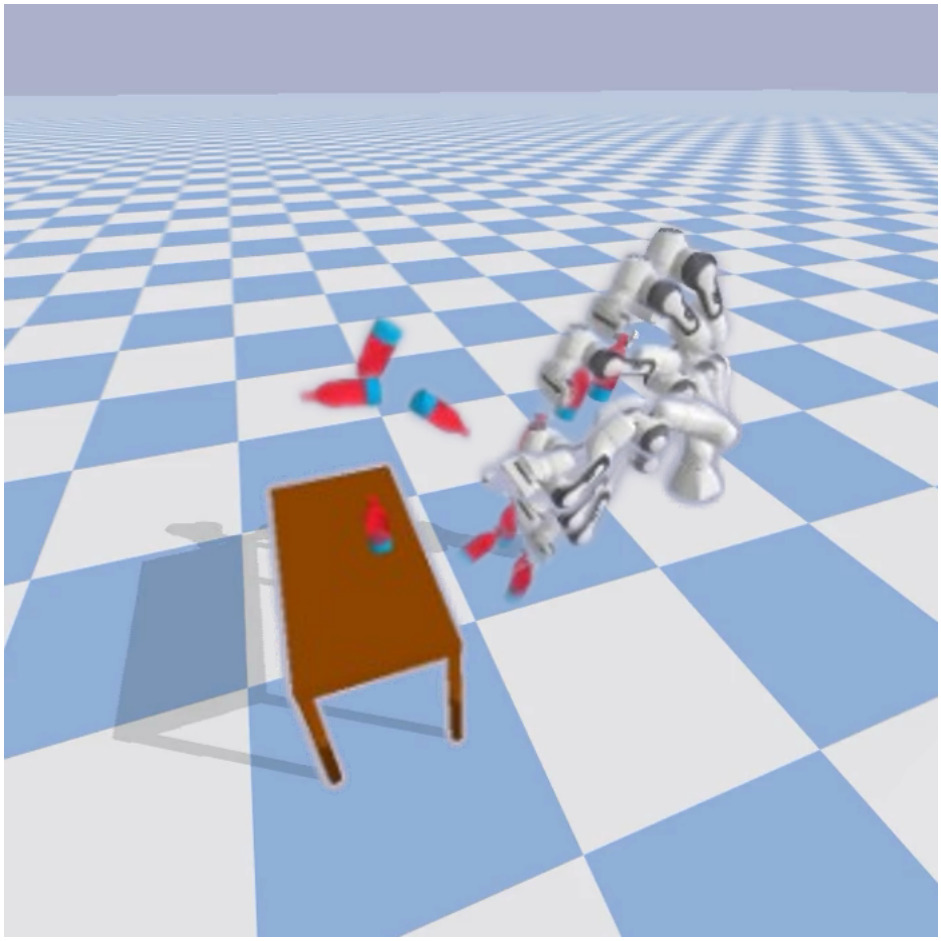


Figure 4.4: A snapshot of bottle flip with feasible robot motion

## 4.2 Sim-to-Real and Real-to-Sim

Unfortunately, The solutions we found in simulation don't meet the real robot hardware limit. There is an extra maximum Cartesian speed limit on the real robot control. Therefore, even though one trajectory doesn't exceed in the joint limit, the real robot controller refuses to execute it.

On the other hand, we found a solution for a real robot by hand tuning. However, when we replay the robot trajectory in simulation, the bottle won't have a enough angular speed to flip it over.

Several factors might explain it. What we believe as the major cause is the interaction during the release phase. First, there is a delay between the time we send the release command and the actual execution. Second, during the release phase, the friction causes an acceleration of the bottle's rotation. Therefore, we would like to model the effect of the friction on the heavy base bottle.

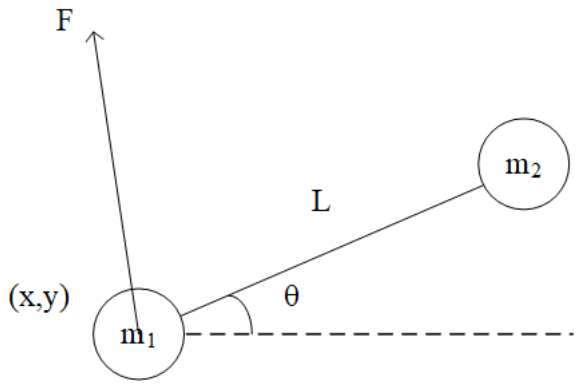



Figure 4.5: The heavy base bottle model with a force applied on one node

The heavy base bottle can be reasonably modelled as a rod with two mass points, as shown in Figure 4.5. By solving Lagrangian equation, we get the differential equation where $F_x, F_y$ are the decomposed force on the x-axis and the y-axis of the external force $F$.

$$\begin{bmatrix} m_1+m_2 & 0 & -m_2L\sin\theta \\ 0 & m_1+m_2 & -m_2L\cos\theta \\ -m_2L\sin\theta & m_2L\cos\theta & m_2L^2 \end{bmatrix} \cdot \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} F_x + m_2\dot{\theta}^2L\cos\theta \\ F_y + (m_1+m_2)g + m_2\dot{\theta}^2L\sin\theta \\ m_2gL\cos\theta \end{bmatrix}$$

A simulation using a time-step of 0.001s on Matlab(Figure 4.6) shows that even a small friction

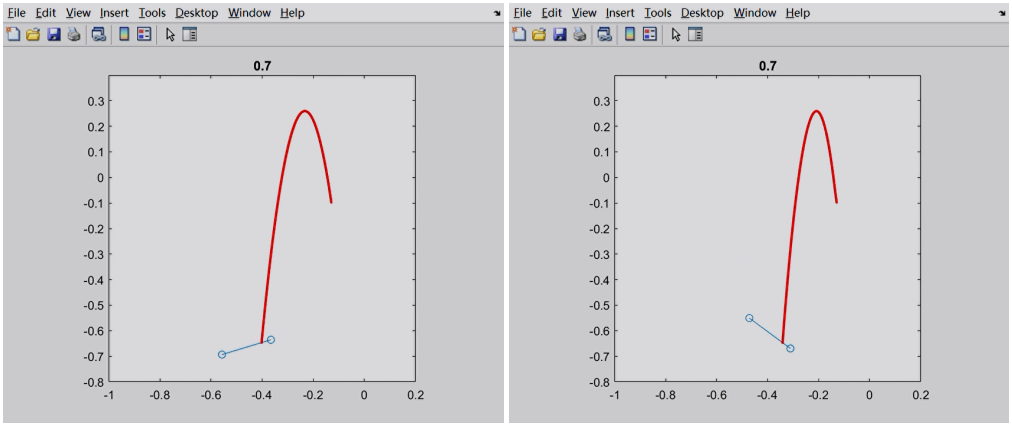



Figure 4.6: The bottle states 0.7s after the release. The right one was applied a force $F_x = 1N$ for 20ms after the release.

can change the landing orientation much. These two bottles are release with initial state

$$x = 0$$
$$y = 0$$
$$\dot{x} = 0.1$$
$$\dot{y} = 2.0$$
$$\theta = -2.49$$
$$\dot{\theta} = -5.0$$

, and the right one was applied a force $F_x = 1N$ for 20ms after the release. The rotational speeds after 0.7s are -5.07 and -6.34.

This provides a good insight for our future study about how to utilize the friction to gain a higher rotational speed without exceeding the robot limit. This also indicates that our previous assumption, that the object state is the same as the gripper center until release, confines the possible solutions we can find with regard to the hardware limit. One example is the solution found by Everyday Robots[2]. On the other hand, another model is needed to analyze the interaction of the object and the gripper during and before the release phase, whose validation on the real robot, unfortunately, might not be a easy task currently because the minimum grasping force of the gripper we are using is 10N which will already hold the object very tight.

[2]https://www.linkedin.com/posts/everydayrobots_join-us-everyday-robots-activity-6958775189736202241-Fgz1

# Conclusion

This project is based on the previous work [3] at LASA, and extends the idea to more generic robotic throwing tasks.

Overall, the first part of this project is to extend the original solution for mobile manipulators to fixed-base manipulators. Utilizing the robot's first vertical joint, a new hedgehog with an extra key is proposed to solve the fixed-base throwing problem. For better reactiveablity, the solution is refined by optimizing the online matching algorithm. The second part of this project is the attempt to control the landing orientation. We propose a relatively ideal problem model with target orientation and a solution for it, providing a preliminary solution that can be validated in simulation. And the interaction between the object and the gripper has been modelled and proved to be one of the major causes of the sim-to-real gap. These works can be a good start for future research.

As for the future works, we can

- Test and refine the algorithm on the real robot. In addition to the inconsistent physics of the simplified simulation and the real world, there are plenty of practical issues one might neglect in simulation. Section 4.2 shows an typical example.
- Refine the initial guess by differentiating the model. Velocity hedgehog is a discrete method with inevitable discretization error. It might be possible to 'push' our solution towards the target with a differentiable model. This would be desirable for scenarios where accuracy is the top priority.
- Explore and build a fine model to analyze the interaction between the object and the gripper and then utilize it for throwing. As we mentioned previously, throwing extends a given robotic manipulator's workspace. Utilizing this interaction will be likely to strengthen the manipulator with a even bigger workspace.
- Extend the model to control the full pose of generic objects, whose flying dynamics are described by 3D translation and 3D rotation [2]. This will allow for more precise manipulation.

This project is an interesting experience for me and I'm glad to learn about the related field

and contribute to it. One significant note I learn is that, do not overlook the gap between the ideal settings in mind and reality. Robot may look fine in simulation, but it can be totally different when you actually run it in reality.

# Bibliography


[1] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[2] Seungsu Kim and Aude Billard. “Estimating the non-linear dynamics of free-flying objects”. In: *Robotics and Autonomous Systems* 60.9 (2012), pp. 1108–1122.

[3] Yang Liu, Aradhana Nayak, and Aude Billard. “A Solution to Adaptive Mobile Manipulator Throwing”. In: *arXiv preprint arXiv:2207.10629* (2022).

[4] M.T. Mason and K.M. Lynch. “Dynamic manipulation”. In: *Proceedings of 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS ’93)*. Vol. 1. 1993, 152–159 vol.1. DOI: 10.1109/IROS.1993.583093.

[5] Qin Zhu and Geoffrey P Bingham. “Is hefting to perceive the affordance for throwing a smart perceptual mechanism?” In: *J Exp Psychol Hum Percept Perform* (2008).