HAL
open science

Collision Detection Accelerated: An Optimization Perspective

Louis Montaut, Quentin Le Lidec, Vladimir Petrik, Josef Sivic, Justin Carpentier

To cite this version:

Louis Montaut, Quentin Le Lidec, Vladimir Petrik, Josef Sivic, Justin Carpentier. Collision Detection Accelerated: An Optimization Perspective. RSS 2022 - Robotics: Science and Systems, Jun 2022, New York, United States. hal-03662157

HAL Id: hal-03662157

https://hal.science/hal-03662157v1

Submitted on 9 May 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Collision Detection Accelerated: An Optimization Perspective

Louis Montaut*†, Quentin Le Lidec†, Vladimir Petrik*, Josef Sivic* and Justin Carpentier†
*Czech Institute of Informatics, Robotics and Cybernetics,
Czech Technical University, Prague, Czech Republic
firstname.lastname@cvut.cz
†Inria and Département d'Informatique de l'École Normale Supérieure,
PSL Research University, Paris, France
firstname.lastname@inria.fr

***Abstract*—Collision detection between two convex shapes is an essential feature of any physics engine or robot motion planner. It has often been tackled as a computational geometry problem, with the Gilbert, Johnson and Keerthi (GJK) algorithm being the most common approach today. In this work we leverage the fact that collision detection is fundamentally a convex optimization problem. In particular, we establish that the GJK algorithm is a specific sub-case of the well-established Frank-Wolfe (FW) algorithm in convex optimization. We introduce a new collision detection algorithm by adapting recent works linking Nesterov acceleration and Frank-Wolfe methods. We benchmark the proposed accelerated collision detection method on two datasets composed of strictly convex and non-strictly convex shapes. Our results show that our approach significantly reduces the number of iterations to solve collision detection problems compared to the state-of-the-art GJK algorithm, leading to up to two times faster computation times.**

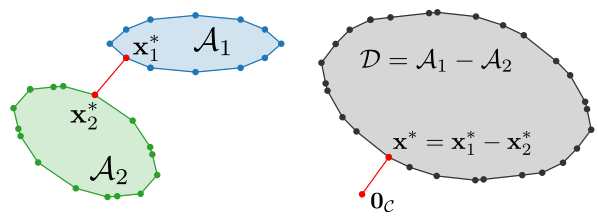
I. Introduction

Physics engines designed to simulate rigid bodies are an essential tool used in a wide variety of applications, notably in robotics, video games, and computer graphics [7, 30, 37]. Collision detection, a crucial feature of any physics engine or robot motion planer [26, 38, 35], consists of finding which objects are colliding or not, *i.e.* are sharing at least one common point or if there exists a separating hyper-plane between both. As simulation often needs to deal with multiple objects and run in real-time (*i.e.*, in video games) or at very high-frequencies (*i.e.*, in robotics), it is crucial for collision detection to be carried out as fast as possible. In order to reduce computational times, collision detection is usually decomposed into two phases thoroughly covered in [10]. The first phase is the *broad phase* which consists in identifying which pair of simulated objects are potentially colliding. Objects in the pair selected during the broad phase are close to each other, hence the collision is uncertain and needs to be considered carefully. The second phase is the *narrow phase* in which each pair identified in the broad phase is tested to check whether a collision is indeed occurring. Collision detection during the narrow phase is the focus of this paper.

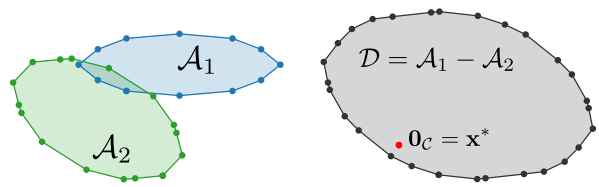
Problem formulation. We consider two convex shapes $\mathcal{A}_1$ and $\mathcal{A}_2$ in $\mathbb{R}^n$ (with $n = 2$ or 3 in common applications). If the shapes are not convex, we use their respective convex-hulls or decompose the shapes into a collection of convex sub-shapes [25]. The *separation distance* between $\mathcal{A}_1$ and $\mathcal{A}_2$, denoted by $\text{dist}(\mathcal{A}_1, \mathcal{A}_2) \in \mathbb{R}_+$, can be formulated as a minimization problem of the form:

$$\begin{aligned} d_{1,2} &= \min_{\boldsymbol{x}_1 \in \mathcal{A}_1, \boldsymbol{x}_2 \in \mathcal{A}_2} \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|^2 \\ \text{and } \text{dist}(\mathcal{A}_1, \mathcal{A}_2) &= \sqrt{d_{1,2}}\,, \end{aligned} \tag{1}$$

where $\boldsymbol{x}_1 \in \mathcal{A}_1$ and $\boldsymbol{x}_2 \in \mathcal{A}_2$ are both vectors in $\mathbb{R}^n$, $d_{1,2}$ is the optimal value of (1) and $\|\cdot\|$ is the Euclidian norm of $\mathbb{R}^n$. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are intersecting (*i.e.* in collision), we have $\text{dist}(\mathcal{A}_1, \mathcal{A}_2) = 0$. If the two shapes do not intersect, we have $\text{dist}(\mathcal{A}_1, \mathcal{A}_2) > 0$. These two cases are illustrated in



(a) **Left:** two distant shapes. **Right:** $\mathbf{0}_\mathcal{C}$ lies outside of the Minkowski difference of the shapes.



(b) **Left:** two overlapping shapes. **Right:** $\mathbf{0}_\mathcal{C}$ lies inside of the Minkowski difference of the shapes.

Fig. 1: **Distant vs. overlapping pairs of shapes and their respective Minkowski difference.** Left column: two convex shapes in 2D. Right column: the Minkowski difference $\mathcal{D}$ of $\mathcal{A}_1$ and $\mathcal{A}_2$. Since $\mathcal{A}_1$ and $\mathcal{A}_2$ are convex, $\mathcal{D}$ is also convex. In (a), the shapes are not in collision hence the origin of the configuration space $\mathcal{C}$, $\mathbf{0}_\mathcal{C}$ (in red) lies outside the Minkowski difference, $\mathbf{0}_\mathcal{C} \notin \mathcal{D}$. The vector $\boldsymbol{x}^* = \boldsymbol{x}_1^* - \boldsymbol{x}_2^*$ separates $\mathcal{A}_1$ from $\mathcal{A}_2$. It is also equal to the projection of $\mathbf{0}_\mathcal{C}$ onto the Minkowski difference $\mathcal{D}$, $\boldsymbol{x}^* = \text{proj}_\mathcal{D}(\mathbf{0}_\mathcal{C})$. In (b), the shapes are overlapping, thus $\mathbf{0}_\mathcal{C} \in \mathcal{D}$. In this case, we have $\boldsymbol{x}^* = \text{proj}_\mathcal{D}(\mathbf{0}_\mathcal{C}) = \mathbf{0}_\mathcal{C}$.

Fig. 1.

Problem (1) allows us to encapsulate and tackle both the *distance computation* problem and the computationally cheaper *Boolean collision check* into one single convex optimization problem. In the distance computation problem, we aim at computing the separation distance between $\mathcal{A}_1$ and $\mathcal{A}_2$, denoted $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$, *i.e.* the distance between their closest points. This distance is useful in some applications such as collision-free path planning [13, 36], especially for pairs of objects entering the narrow phase. If a pair of objects has not been selected by the broad phase, a cheap estimate of $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$ is usually enough [10]. In the Boolean collision check we only aim at determining if $\mathcal{A}_1$ and $\mathcal{A}_2$ intersect, and computing $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$ is unnecessary. However, we will later see that the Boolean collision check is a sub-problem of the distance computation problem: solving (1) can be early-stopped once a separating plane between $\mathcal{A}_1$ and $\mathcal{A}_2$ has been found. In the rest of this paper, we will use the generic term "collision detection" when we mean to encapsulate both distance computation and Boolean collision checking. We will specify when the distinction is needed.

Related work. The most well-known algorithm for collision detection between two convex shapes is the so-called Gilbert-Johnson-Keerthi algorithm (GJK) [14]. It can handle both the distance computation and the Boolean collision check [39]. Most alternatives to GJK found in the literature focus on computing collisions between convex polyhedra such as the Lin-Canny algorithm [24] or the V-Clip [27] algorithm. Although GJK is equivalent in performance to these algorithms [3], it is not restricted to convex polyhedra. The strength of GJK is formulating the collision detection problem on the Minkowski difference. The properties of the Minkowski difference are used to cleverly compute support vectors on the Minkowski difference (these notions will be introduced in detail in Sec. II). GJK is thus able to handle collision detection and distance computation for many different shapes such as convex polyhedra and basic primitives (*i.e.*, spheres, ellipsoids, cylinders, capsules etc.) [39, 40, 10]. The generality of GJK as well as its efficiency, good precision and ease of implementation makes it the state-of-the-art algorithm for collision detection between two convex shapes. The computation time of GJK has been reduced thanks to improvements to its sub-operations [28, 39]. However, to the best of our knowledge, no previous work has focused explicitly on lowering the number of iterations of GJK. This work notably shows how to accelerate collision detection by directly lowering the number of iterations needed to solve a collision problem instance compared to the vanilla GJK algorithm.

Over the years, collision detection has often been seen as a computational geometry problem. However we argue that this view has limited the improvement of collision detection and propose to reframe it as a convex optimization problem. As briefly mentioned already in their 1988 paper [14] and brought-up again by [34], the ideas developed by Gilbert, Johnson and Keerthi are rooted in convex optimization, notably in the work of Wolfe [41] and Gilbert [15]. Similarly to GJK, these works aim at solving a Minimum-Norm Point problem and are sub-cases of a more general algorithm: the Frank-Wolfe method, also known as the conditional gradient descent.

The Frank-Wolfe algorithm (FW) dates back to 1956 and is one of the first convex optimization algorithms. It has been heavily studied over the years by the optimization community. This algorithm iterates over the computation of *support points* in order to approach the optimal solution. The undesired zig-zagging behavior of FW, already identified by its authors, has been addressed by introducing corrections to the original FW method [12, 15, 16, 18, 20, 21, 41]. In [18] and [21], widely used corrections of the FW algorithm are analyzed as well as their convergence properties. In this work, we notably show in Sec. II that the GJK algorithm is an instance of the *fully-corrective* Frank-Wolfe algorithm, covered in [21], applied to solving a Minimum-Norm Point problem (MNP). Finally, recent works have also tried to accelerate the FW algorithm by applying the so-called Nesterov acceleration [29], a classic acceleration technique in unconstrained optimization. Nesterov momentum has been successfully added by Li et al. [22] in order to accelerate FW. In [34], Qin and An take a different approach as they are interested in the general problem of projecting a point onto a Minkowski difference in *any* dimension. In order to accelerate the theoretical convergence of the 1966 Gilbert algorithm, the authors devise the NESMINO algorithm which exploits the classic Nesterov acceleration. However, the minimization problem (1) is modified by introducing a smoothing term. By doing so, the authors rely on successive projections onto the original shapes instead of computing support points. This makes the NESMINO algorithm similar to the projected-gradient descent method. Unlike NESMINO, GJK and our work are sub-cases of the FW algorithm. As pointed out by the authors, although the NESMINO algorithm makes use of the Nesterov acceleration, it does not accelerate over the original 1966 Gilbert algorithm. In Sec. IV, we experimentally show that the NESMINO algorithm is slower when compared to GJK and our method.

Contributions. Our work builds on the seminal works by Frank and Wolfe [11] and Gilbert et al. [14] as well as on the work of Li et al. [22] to globally accelerate distance computation and collision checking algorithms between convex shapes. We make these three main contributions:

- ↪ We recast the collision detection problem as a convex optimization problem which can be solved by the FW algorithm. Using the ideas developed by Gilbert, Johnson and Keerthi, we show that GJK is in fact a sub-case of the fully-corrective FW algorithm;
- ↪ We adapt recent works on Nesterov-accelerated FW in order to accelerate both the distance computation and the Boolean collision check problems;
- ↪ We empirically analyse the convergence of our proposed approach on two large shape benchmarks. Results

show a faster convergence of our approach leading to a computational time up to two times faster than the state-of-the-art GJK algorithm, on both distance computation and Boolean collision checking.

Paper outline. The paper is organized as follows. In Sec. II, we recast the distance computation problem as a Frank-Wolfe instance. We introduce the duality-gap of the FW method, allowing us to bound the distance to the optimal solution of the distance computation problem. We also present the fully-corrective version of FW and show the link between GJK and FW. In Sec. III, we introduce recent work on Nesterov-accelerated FW and show how to adapt it for both distance computation and Boolean collision checking. For distance computation, we adapt the convergence criterion of FW when using Nesterov acceleration in order to retain the bound on the distance to the optimal solution. We also propose to adapt the Nesterov acceleration for non-strictly convex shapes. In Sec. IV, we evaluate our approach against the state-of-the-art GJK algorithm on two benchmarks containing both strictly convex shapes and non-strictly convex shapes.

II. Collision Detection from a Frank-Wolfe Perspective

In this section, we highlight the natural connection between computing the distance between convex shapes and convex optimization, particularly within the frame of the Frank-Wolfe setting. We notably show that the GJK algorithm can be seen as a variant of the Frank-Wolfe algorithm that leverages properties of convex 3D shapes to drastically lower the computational complexity.

Distance computation and Boolean collision checking. As recalled in Sec. I, collision detection is a sub-case of distance computation: $\text{dist}(\mathcal{A}_1, \mathcal{A}_2) > 0$ means that the two shapes do not overlap while $\text{dist}(\mathcal{A}_1, \mathcal{A}_2) = 0$ means that the shapes are in collision. In the particular case of $\text{dist}(\mathcal{A}_1, \mathcal{A}_2) > 0$, it is sufficient to find a strictly positive lower bound on $d_{1,2}$ to solve the collision problem. In the context of convex shapes, this is often simpler than computing the distance between the two shapes [36] and can be done by finding a plane separating $\mathcal{A}_1$ from $\mathcal{A}_2$. In the rest of the paper, we focus on the generic problem of computing the distance between $\mathcal{A}_1$ and $\mathcal{A}_2$, as it encapsulates the simpler Boolean collision check covered later in this section. Results for the particular Boolean collision checking case are analyzed in the experimental section IV.

Recasting the distance computation problem onto the Minkowski difference. The first important idea of Gilbert et al. [14] is to recast the distance computation problem onto the Minkowski difference $\mathcal{D}$ of the shapes, illustrated in Fig. 1 and defined as follows:

$$\mathcal{D} = \mathcal{A}_1 - \mathcal{A}_2 = \{\boldsymbol{x} = \boldsymbol{x}_1 - \boldsymbol{x}_2 \mid \boldsymbol{x}_1 \in \mathcal{A}_1, \boldsymbol{x}_2 \in \mathcal{A}_2\} \subset \mathcal{C}\,, \tag{2}$$

where $\mathcal{C} = \mathbb{R}^n$ is the so-called *collision space*. The shapes $\mathcal{A}_1$ and $\mathcal{A}_2$ lie in the shape space, and the Minkowski difference $\mathcal{D}$ lies in the collision space. Although both the shape space and the collision space are in $\mathbb{R}^n$, we make a distinction between the two to highlight the change in perspective. We stress that the Minkowski difference $\mathcal{D}$ is *specific* to shapes $\mathcal{A}_1$ and $\mathcal{A}_2$. If the relative position or relative orientation between $\mathcal{A}_1$ and $\mathcal{A}_2$ changes, their Minkowski difference changes accordingly.

The following properties hold for the Minkowski difference $\mathcal{D}$:

1) Since $\mathcal{A}_1$ and $\mathcal{A}_2$ are convex sets, $\mathcal{D}$ is also convex.
2) If $\mathcal{A}_1$ and $\mathcal{A}_2$ are intersecting, the origin of $\mathcal{C}$, denoted as $\mathbf{0}_\mathcal{C}$, lies inside the Minkowski difference $\mathcal{D}$, *i.e.* $\mathbf{0}_\mathcal{C} = \boldsymbol{x}_1 - \boldsymbol{x}_2$ for some $\boldsymbol{x}_1 \in \mathcal{A}_1$ and $\boldsymbol{x}_2 \in \mathcal{A}_2$.
3) If $\mathcal{A}_1$ and $\mathcal{A}_2$ are not intersecting, the projection of $\mathbf{0}_\mathcal{C}$ onto $\mathcal{D}$, $\boldsymbol{x}^* = \text{proj}_\mathcal{D}(\mathbf{0}_\mathcal{C})$, corresponds to two vectors $\boldsymbol{x}_1^* \in \mathcal{A}_1$ and $\boldsymbol{x}_2^* \in \mathcal{A}_2$, also called witness vectors in the computational geometry literature [10]. Contrary to $\boldsymbol{x}^*$, these vectors $\boldsymbol{x}_1^*$ and $\boldsymbol{x}_2^*$ are not necessarily unique, as is the case for non-strictly convex shapes such as two parallel boxes.
4) Finally, we always have $\|\boldsymbol{x}^*\| = \text{dist}(\mathcal{A}_1, \mathcal{A}_2)$.

This final remark allows us to recast the distance computation problem (1) onto the Minkowski difference as follows:

$$d_{1,2} = \min_{\boldsymbol{x} \in \mathcal{D}} \|\boldsymbol{x} - \mathbf{0}_\mathcal{C}\|^2 = \min_{\boldsymbol{x} \in \mathcal{D}} \|\boldsymbol{x}\|^2 \,. \tag{3}$$

The convex optimization problem (3) is equivalent to (1) and is known as a Minimum-Norm Point problem in the optimization literature [1, 21, 41]. In our case, $\mathbf{0}_\mathcal{C} \in \mathcal{C} = \mathbb{R}^n$ is the null vector *i.e.* the origin of the collision space. We thus aim at finding the point in $\mathcal{D}$ with the lowest norm. This vector $\boldsymbol{x}^*$ is the *optimal solution* to (3), given by $d_{1,2} = \|\boldsymbol{x}^*\|^2 = \text{dist}(\mathcal{A}_1, \mathcal{A}_2)^2$.

Directly computing the Minkowski difference $\mathcal{D}$ is neither analytically tractable nor computationally efficient. Most of the first and second order methods for constrained convex optimization problems, such as projected gradient descent or interior point methods [2], are thus sub-optimal choices. However, computing support vectors of the Minkowski difference $\mathcal{D}$, a notion defined hereafter in this section, is relatively simple and largely demonstrated by Gilbert et al. [14]. Solving convex optimization problems by computing support vectors is the foundation of the Frank-Wolfe algorithm and its variants [18], as we discuss next.

Distance computation using the Frank-Wolfe algorithm. The Frank-Wolfe algorithm (FW) [11] is one of the oldest convex optimization methods and solves the following constrained optimization problem:

$$f(\boldsymbol{x}^*) = \min_{\boldsymbol{x} \in \mathcal{D}} f(\boldsymbol{x}), \tag{4}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is a convex and differentiable function and $\mathcal{D}$ is a compact convex set. For our distance computation problem (3), we use $f(\boldsymbol{x}) = \|\boldsymbol{x}\|^2$ and the Minkowski difference $\mathcal{D}$ as convex constraint set. As a side note, the following discussed algorithms all require an initial starting point $\boldsymbol{x}_0 \in \mathcal{D}$. Shapes used in physics engines are usually attached to a frame in order to keep track of their position

Algorithm 1 Frank-Wolfe algorithm with linesearch [18]

Let $\boldsymbol{x}_0 \in \mathcal{D}$, $\epsilon > 0$

For `k=0, 1, ...` **do**

1: $\boldsymbol{d}_k = \nabla f(\boldsymbol{x}_k)$ ▷ Direction of support

2: $\boldsymbol{s}_k \in \arg\min_{\boldsymbol{s}\in\mathcal{D}} \langle \boldsymbol{d}_k, \boldsymbol{s}\rangle (= S_{\mathcal{D}}(\boldsymbol{d}_k))$ ▷ Support (7)

3: **If** $g_{FW}(\boldsymbol{x}_k) \leq \epsilon$**, return** $f(\boldsymbol{x}_k)$ ▷ Duality gap (15)

4: $\gamma_k = \arg\min_{\gamma\in[0,1]} f(\gamma \boldsymbol{x}_k + (1-\gamma)\boldsymbol{s}_k)$ ▷ Linesearch

5: $\boldsymbol{x}_{k+1} = \gamma_k \boldsymbol{x}_k + (1-\gamma_k)\boldsymbol{s}_k$ ▷ Update iterate

In the case of the distance computation problem (3)*, where* $f(\boldsymbol{x}) = \|\boldsymbol{x}\|^2$*, line 4-5 correspond to projecting* $\mathbf{0}_{\mathcal{C}}$ *on the segment* $[\boldsymbol{x}_k, \boldsymbol{s}_k]$:

6: $\boldsymbol{x}_{k+1} = \text{proj}_{[\boldsymbol{x}_k,\boldsymbol{s}_k]}(\mathbf{0}_{\mathcal{C}})$ ▷ Project $\mathbf{0}_{\mathcal{C}}$ on $[\boldsymbol{x}_k, \boldsymbol{s}_k]$

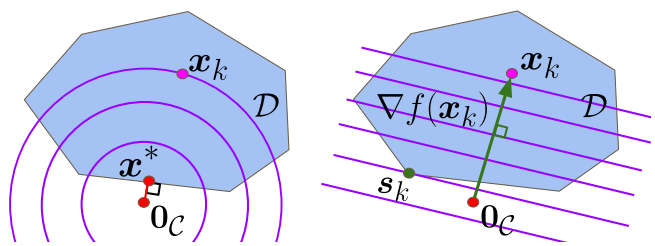


Fig. 2: **Computing a support vector $\boldsymbol{s}_k$ in direction $\nabla f(\boldsymbol{x}_k)$ on convex set $\mathcal{D}$.** We illustrate with the example of distance computation. On the left, we draw the Minkowski difference $\mathcal{D}$ which point of minimum norm (MNP) is $\boldsymbol{x}^*$ *i.e.* $\boldsymbol{x}^*$ is the projection of $\mathbf{0}_{\mathcal{C}}$ onto $\mathcal{D}$, $\boldsymbol{x}^* = \text{proj}_{\mathcal{D}}(\mathbf{0}_{\mathcal{C}})$. The iterate at iteration k of the FW algorithm is $\boldsymbol{x}_k$. In purple we draw the level sets of the function $f(\boldsymbol{x}) = \|\boldsymbol{x}\|^2$. On the right, we draw in purple the level sets of the linearization of f at iterate $\boldsymbol{x}_k$, h_k. The first step of the FW algorithm is to compute support vector $\boldsymbol{s}_k$ in the direction of $\nabla f(\boldsymbol{x}_k)$ (green arrow), $\boldsymbol{s}_k \in S_{\mathcal{D}}(\nabla f(\boldsymbol{x}_k))$. In the second step of the FW algorithm, we compute $\boldsymbol{x}_{k+1}$ as a convex combination of $\boldsymbol{x}_k$ and $\boldsymbol{s}_k$ *i.e.* $\boldsymbol{x}_{k+1}$ is a point on the segment $[\boldsymbol{x}_k, \boldsymbol{s}_k]$.

and orientation in space. We denote $\boldsymbol{c}^1 \in \mathcal{A}_1$ and $\boldsymbol{c}^2 \in \mathcal{A}_2$ the origins of the frames attached to $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. In the rest of this paper, we take $\boldsymbol{x}_0 = \boldsymbol{c}^1 - \boldsymbol{c}^2$.

The FW algorithm, summarized in Alg. 1, is a gradient-descent method. It consists in iteratively applying two steps in order to converge towards the optimal solution $\boldsymbol{x}^*$ of (4). If we denote by $\boldsymbol{x}_k$ the estimate of $\boldsymbol{x}^*$ at iteration k, these two steps correspond to:

1) First, we compute a support vector $\boldsymbol{s}_k$ in the direction of $\nabla f(\boldsymbol{x}_k)$, by solving a linear optimization problem on $\mathcal{D}$.
2) Second, we update our current iterate $\boldsymbol{x}_k$ to obtain $\boldsymbol{x}_{k+1}$, by taking a convex combination of the current iterate $\boldsymbol{x}_k$ and the computed support vector $\boldsymbol{s}_k$.

In the following we detail these steps in the context of distance computation. At iteration k, the current iterate $\boldsymbol{x}_k$ is the estimate of the optimal solution $\boldsymbol{x}^*$ and $f(\boldsymbol{x}_k)$ is the estimate of the optimal value of (4), $f(\boldsymbol{x}^*)$, at iteration k. We write the linearization of the function f at $\boldsymbol{x}_k$ and denote it as h_k:

$$h_k(\boldsymbol{s}) = f(\boldsymbol{x}_k) + \langle \nabla f(\boldsymbol{x}_k), \boldsymbol{s} - \boldsymbol{x}_k\rangle \tag{5}$$

where $\boldsymbol{s}$ is a vector of $\mathbb{R}^n$, $\nabla f(\boldsymbol{x}_k)$ is the gradient of f at $\boldsymbol{x}_k$ and $\langle\cdot,\cdot\rangle$ denotes the dot product between two vectors of $\mathbb{R}^n$.

↪ **Step 1.** The first step of the FW algorithm at iteration k consists of finding a minimizer $\boldsymbol{s}_k \in \mathcal{D}$ of h_k on the convex set $\mathcal{D}$ (line 2 in Alg. 1). Such a vector $\boldsymbol{s}_k$ is called a *support vector* of $\mathcal{D}$ or simply a *support* and is defined as follows:

$$\boldsymbol{s}_k \in \arg\min_{\boldsymbol{s}\in\mathcal{D}} h_k(\boldsymbol{s}) = \arg\min_{\boldsymbol{s}\in\mathcal{D}} \langle \nabla f(\boldsymbol{x}_k), \boldsymbol{s}\rangle\,. \tag{6}$$

Fig. 2 gives a graphical understanding of support $\boldsymbol{s}_k$. The vector $\boldsymbol{s}_k$ belongs to $\mathcal{D}$ and is in the *most* opposite direction w.r.t. $\nabla f(\boldsymbol{x}_k)$. In order to highlight the importance of the direction in which a support $\boldsymbol{s}_k$ is computed, we now introduce the notion of *support direction* and *support function*. Given a support direction $\boldsymbol{d} \in \mathbb{R}^n$, the support function $S_{\mathcal{D}}$ returns a set of $\mathcal{D}$ and is defined as:

$$S_{\mathcal{D}}(\boldsymbol{d}) = \arg\min_{\boldsymbol{s}\in\mathcal{D}} \langle \boldsymbol{d}, \boldsymbol{s}\rangle \subset \mathcal{D}\,. \tag{7}$$

The support function $S_{\mathcal{D}}$ may return a set with more than one vector. We only need to use one vector of this set. Thinking in terms of direction of support allows us to understand that this direction can be rescaled while preserving the output of the support function:

$$\forall \boldsymbol{d} \in \mathbb{R}^n,\ \forall \alpha > 0,\ S_{\mathcal{D}}(\alpha \boldsymbol{d}) = S_{\mathcal{D}}(\boldsymbol{d}). \tag{8}$$

A support $\boldsymbol{s}_k \in \mathcal{D}$ at iteration k is thus computed in the direction $\boldsymbol{d}_k = \nabla f(\boldsymbol{x}_k)$ and belongs to $S_{\mathcal{D}}(\nabla f(\boldsymbol{x}_k))$, $\boldsymbol{s}_k \in S_{\mathcal{D}}(\nabla f(\boldsymbol{x}_k))$.

We now explain how to compute the support vector $\boldsymbol{s}_k$ in the case of the distance computation problem (3) where we minimize $f(\boldsymbol{x}) = \|\boldsymbol{x}\|^2$ on the Minkowski difference $\mathcal{D}$ of $\mathcal{A}_1$ and $\mathcal{A}_2$. First, we have $\nabla f(\boldsymbol{x}) = 2\boldsymbol{x}$. Therefore, in the case of problem (3), it follows that:

$$\boldsymbol{s}_k \in S_{\mathcal{D}}(\boldsymbol{x}_k) = \arg\min_{\boldsymbol{s}\in\mathcal{D}} \langle \boldsymbol{x}_k, \boldsymbol{s}\rangle\,. \tag{9}$$

As demonstrated by Gilbert et al. [14], any vector $\boldsymbol{s} \in S_{\mathcal{D}}(\boldsymbol{d})$ related to the Minkowski difference can be decomposed as the difference between two support vectors $\boldsymbol{s}_{\mathcal{A}_1} \in S_{\mathcal{A}_1}(\boldsymbol{d})$ and $\boldsymbol{s}_{\mathcal{A}_2} \in S_{\mathcal{A}_2}(-\boldsymbol{d})$ over the two individual shapes, leading to the following relation:

$$\boldsymbol{s} = \boldsymbol{s}_{\mathcal{A}_1} - \boldsymbol{s}_{\mathcal{A}_2} \in S_{\mathcal{D}}(\boldsymbol{d}). \tag{10}$$

Equation (10) shows that we can construct a support of the Minkowski difference from the supports of the original shapes. This property highlights the powerful change of perspective of working on the Minkowski difference. Indeed, there exists a large number of shapes for which computing supports is simple: spheres, ellipsoids, cylinders, capsules, polytopes etc. [10, 39, 40]. Fig. 3 illustrates the construction of a support of the Minkowski difference $\mathcal{D}$ using the supports of the original shapes $\mathcal{A}_1$ and $\mathcal{A}_2$.

↪ **Step 2.** Once a support vector $\boldsymbol{s}_k \in S_{\mathcal{D}}(\boldsymbol{x}_k)$ has been computed, we update the iterate $\boldsymbol{x}_k$ to obtain $\boldsymbol{x}_{k+1}$ by taking a convex combination between $\boldsymbol{s}_k$ and $\boldsymbol{x}_k$. The original FW algorithm uses a parameter-free update:

$$\boldsymbol{x}_{k+1} = \gamma_k \boldsymbol{x}_k + (1-\gamma_k)\boldsymbol{s}_k\,, \tag{11}$$

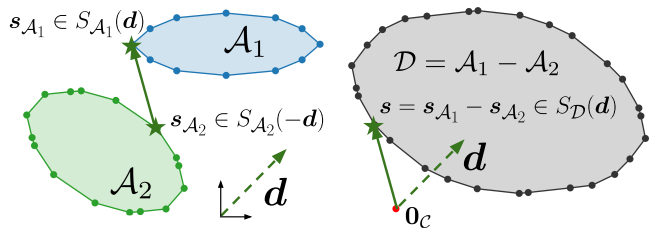


Fig. 3: **Computing a support vector on the Minkowski difference using support vectors on the individual shapes**. The vector $\boldsymbol{s}_{\mathcal{A}_1}$ is a support vector of shape $\mathcal{A}_1$ in direction $\boldsymbol{d}$. The vector $\boldsymbol{s}_{\mathcal{A}_2}$ is a support vector of shape $\mathcal{A}_2$ in direction $-\boldsymbol{d}$. The constructed vector $\boldsymbol{s} = \boldsymbol{s}_{\mathcal{A}_1} - \boldsymbol{s}_{\mathcal{A}_2}$ is a support vector of the Minkowski difference $\mathcal{D}$ in the direction $\mathcal{D}$.

where $\gamma_k = \frac{k+1}{k+2} \in [0,1]$ controls the step size. Alternatively, a linesearch can be carried out to find a better iterate $\boldsymbol{x}_{k+1}$ (line 4 in Alg. 1):

$$\begin{aligned} \boldsymbol{\gamma}_k &= \arg\min_{\gamma \in [0,1]} f(\gamma \boldsymbol{x}_k + (1-\gamma)\boldsymbol{s}_k) \\ \boldsymbol{x}_{k+1} &= \gamma_k \boldsymbol{x}_k + (1-\gamma_k)\boldsymbol{s}_k. \end{aligned} \tag{12}$$

In the distance computation case where $f(\boldsymbol{x}) = \|\boldsymbol{x}\|^2$, this linesearch (12) is equivalent to projecting $\mathbf{0}_{\mathcal{C}}$ onto the segment $[\boldsymbol{x}_k, \boldsymbol{s}_k]$, $\boldsymbol{x}_k = \text{proj}_{[\boldsymbol{x}_k, \boldsymbol{s}_k]}(\mathbf{0}_{\mathcal{C}})$ (line 4 in Alg. 1). Since $\mathcal{D}$ is convex, both (11) and (12) updates are guaranteed to remain in $\mathcal{D}$.

Stopping criteria. As Frank-Wolfe deals with convex problems, the *duality gap* associated to problem (4) can be used as a stopping criterion. Due to its convexity, the function f is always above its linearization. Otherwise said, for any $\boldsymbol{x} \in \mathbb{R}^n$ and any $\boldsymbol{s} \in \mathbb{R}^n$:

$$f(\boldsymbol{s}) \geq f(\boldsymbol{x}) + \langle \nabla f(\boldsymbol{x}), \boldsymbol{s} - \boldsymbol{x} \rangle \,. \tag{13}$$

Reworking this inequality and applying the $\min$ operator enables us to compute the Frank-Wolfe duality gap $g_{\text{FW}}(\boldsymbol{x}) \in \mathbb{R}_+$ which gives an upper-bound on the difference $f(\boldsymbol{x}) - f(\boldsymbol{x}^*)$:

$$f(\boldsymbol{x}) - f(\boldsymbol{x}^*) \leq -\min_{\boldsymbol{s} \in \mathcal{D}} \langle \nabla f(\boldsymbol{x}), \boldsymbol{s} - \boldsymbol{x} \rangle = g_{\text{FW}}(\boldsymbol{x}) \,. \tag{14}$$

In particular, at iteration k of the FW algorithm, we have:

$$f(\boldsymbol{x}_k) - f(\boldsymbol{x}^*) \leq g_{\text{FW}}(\boldsymbol{x}_k) = \langle \nabla f(\boldsymbol{x}_k), \boldsymbol{x}_k - \boldsymbol{s}_k \rangle \,, \tag{15}$$

where $\boldsymbol{s}_k \in S_{\mathcal{D}}(\nabla f(\boldsymbol{x}_k))$ is the support vector computed at iteration k in the direction of $\nabla f(\boldsymbol{x}_k)$. The duality-gap $g_{\text{FW}}(\boldsymbol{x}_k)$ serves as a convergence criterion for the Frank-Wolfe method and is cheap to compute. Applied to the distance computation problem (3), the duality gap at iteration k, $g_{\text{FW}}(\boldsymbol{x}_k)$, guarantees that:

$$\|\boldsymbol{x}_k\|^2 - \|\boldsymbol{x}^*\|^2 \leq g_{\text{FW}}(\boldsymbol{x}_k) = 2\langle \boldsymbol{x}_k, \boldsymbol{x}_k - \boldsymbol{s}_k \rangle. \tag{16}$$

Using the triangular inequality of the Euclidian norm and the convexity of the Minkowski difference $\mathcal{D}$, we can show that:

$$\|\boldsymbol{x}_k - \boldsymbol{x}^*\|^2 \leq \|\boldsymbol{x}_k\|^2 - \|\boldsymbol{x}^*\|^2 \leq g_{\text{FW}}(\boldsymbol{x}_k) \,. \tag{17}$$

Algorithm 2 Boolean collision checking: separating plane condition

Insert after line 2 in Alg. 1:

1: **If** $\langle \boldsymbol{d}_k, \boldsymbol{s}_k \rangle > 0$**, return** `False`
If after termination $d_{1,2} = 0$**, return** `True`

Inequality (17) is useful in practice as it allows to finely control the desired tolerance on the distance to the optimal solution $\boldsymbol{x}^*$ (line 3 in Alg. 1). Indeed, if ones wants to compute an estimate $\boldsymbol{x}$ of the optimal solution $\boldsymbol{x}^*$ at precision ϵ, meaning that $\|\boldsymbol{x} - \boldsymbol{x}^*\| \leq \sqrt{\epsilon}$, it is sufficient to check that $g_{\text{FW}}(\boldsymbol{x}) \leq \epsilon$.

Boolean collision checking. As mentioned earlier, the problem of distance computation encompasses the problem of collision checking. Indeed, in collision checking, we are only interested in finding a separating plane between $\mathcal{A}_1$ and $\mathcal{A}_2$, if it exists. This is equivalent to finding a separating plane between $\mathcal{D}$ and $\mathbf{0}_{\mathcal{C}}$. For *any* support direction $\boldsymbol{d}$, if we have:

$$\langle \boldsymbol{d}, \boldsymbol{s} \rangle > 0, \; \boldsymbol{s} \in S_{\mathcal{D}}(\boldsymbol{d}), \tag{18}$$

then the plane supported by the vector $\boldsymbol{d}$ separates $\mathcal{D}$ from $\mathbf{0}_{\mathcal{C}}$ [39]. This also means that, in the case where the two shapes intersect, collision checking has the same computational complexity than distance computation. As shown in Alg. 2, we add this separating plane condition before line 2 in Alg. 1.

The Frank-Wolfe active-set. As many gradient-descent algorithms, the FW method tends to zig-zag towards the optimal solution [21], slowing down the convergence to the optimum. This behavior is undesired and amplified if the optimal solution $\boldsymbol{x}^*$ lies close to the boundary of the constraint set $\mathcal{D}$. In collision detection, this corresponds to the case where the two shapes are not intersecting. This zig-zagging behavior is due to the way that Frank-Wolfe approaches the set of active constraints [21], also called *active-set* in the optimization literature [2]. In the FW setting, the active-set at iteration k, denoted $W_k = \{\boldsymbol{s}^0, ..., \boldsymbol{s}^r\} \subset \mathcal{D}$, is the set of vectors in $\mathcal{D}$ used by the algorithm to maintain a convex combination of the iterate $\boldsymbol{x}_k$:

$$\boldsymbol{x}_k = \sum_{i=0}^{r} \lambda^i \boldsymbol{s}^i, \; \sum_{i=0}^{r} \lambda^i = 1 \text{ with } \boldsymbol{s}^i \in W_k \subset \mathcal{D} \text{ and } \lambda^i > 0. \tag{19}$$

In Alg. 3, we rewrite the FW algorithm with linesearch (Alg. 1) in order to highlight the notion of active-set:

- A iteration k, the active-set is only composed of $\boldsymbol{x}_k$, $W_k = \{\boldsymbol{x}_k\}$.
- The active-set W_k is then augmented by computing a support $\boldsymbol{s}_k$ (line 2 in Alg. 3) to obtain $\widetilde{W}_{k+1} = \{\boldsymbol{x}_k, \boldsymbol{s}_k\}$ (line 4 in Alg. 3).
- We then minimize function f on the convex-hull of $\widetilde{W}_{k+1}$, $\text{conv}(\widetilde{W}_{k+1})$, which is simply the segment $[\boldsymbol{x}_k, \boldsymbol{s}_k]$. For the distance computation problem (3), this linesearch operation is equivalent

to projecting $\mathbf{0}_{\mathcal{C}}$ onto the segment $[\boldsymbol{x}_k, \boldsymbol{s}_k]$ (line 5 in Alg. 3).

- Finally, the active-set is updated $W_{k+1} = \{\boldsymbol{x}_{k+1}\}$ (line 6 in Alg. 3).

In practice, discarding previously computed supports when updating the active-set is inefficient and causes the zig-zagging phenomenon observed in the FW algorithm [21]. In the optimization literature, a rich and wide variety of variants of the FW algorithm have been introduced to efficiently cope with the active-set in order to improve the convergence rate of the FW method [12, 16, 17, 19, 41]. However, these variants remain too generic and are not suited for the specific problem of collision detection. Instead, we propose next to incorporate the active-set strategy used in GJK within the Frank-Wolfe setting.

Connection between GJK and Frank-Wolfe. In the case of collision detection, Gilbert et al. [14] developed an efficient strategy to handle the active-set at a minimal cost. To represent the current estimate $\boldsymbol{x}_k$ and the optimal solution $\boldsymbol{x}^*$, GJK exploits the concept of *simplexes* in $\mathbb{R}^3$. A simplex in $\mathbb{R}^n$ corresponds to a set containing *at most* $n+1$ vectors of $\mathbb{R}^n$ and the *rank* r of a simplex is the number of vectors it contains ($0 < r \leq n+1$). For 3-dimensional spaces, a simplex corresponds either to a point ($r = 1$), a segment ($r = 2$), a triangle ($r = 3$) or a tetrahedron ($r = 4$). Similarly to the simplex methods for Linear Programming [8], the Carathéodory theorem [4] motivates the use of simplexes. Let $\mathcal{Y}$ be a set of $N \geq n$ vectors in $\mathbb{R}^n$, $\mathcal{Y} = \{\boldsymbol{y}^i \in \mathbb{R}^n\}_{0 \leq i \leq N}$. The Carathéodory theorem states that any vector $\boldsymbol{x} \in \text{conv}(\mathcal{Y})$ can be expressed as the convex combination of at most $n+1$ vectors of $\mathcal{Y}$:

$$\boldsymbol{x} = \sum_{j=0}^{r} \lambda^j \boldsymbol{y}^j, \text{ with } \boldsymbol{y}^j \in \mathcal{Y},\ \lambda^j > 0,\ \sum_{i=0}^{r} \lambda^j = 1. \tag{20}$$

Hence, any vector in $\mathcal{D}$, and particularly the optimal solution $\boldsymbol{x}^* \in \mathcal{D} = \text{conv}(\mathcal{D})$ of the distance computation problem (3), can be identified as a convex combination of the vectors composing a simplex W. Relying on simplexes is attractive as there is no need to run any algorithm to compute the convex-hull of a simplex as they are convex by construction. Frank-Wolf algorithms may operate on more complex active-sets, which might become hard to tackle from a computational point of view [18, 21]. In other words, the problem of finding the optimal solution $\boldsymbol{x}^*$ can be reformulated as the problem of identifying the optimal simplex W^* on which $\boldsymbol{x}^*$ can be decomposed into a convex combination. This is exactly the approach followed by GJK that we now detail.

At iteration k of GJK, the current iterate $\boldsymbol{x}_k$ is a convex combination of the vectors composing the simplex W_k of rank $r_k \leq n$. To update $\boldsymbol{x}_k$ and W_k, the following procedure is applied:

- After computing support vector $\boldsymbol{s}_k$ (line 2 in Alg. 3), we add $\boldsymbol{s}_k$ to W_k to obtain $\widetilde{W}_{k+1} = W_k \cup \{\boldsymbol{s}_k\}$ (line 4 in Alg. 3). The set $\widetilde{W}_{k+1}$ is now a simplex of rank $\widetilde{r}_{k+1} \leq n+1$.

Algorithm 3 Frank-Wolfe algorithm with line-search (see Alg. 1) rewritten with active-sets and applied to the distance computation problem (3)

Let $\boldsymbol{x}_0 \in \mathcal{D}$, $W_0 = \{\boldsymbol{x}_0\}$, $\epsilon > 0$
For `k=0, 1, ...` **do**

1: $\boldsymbol{d}_k = \boldsymbol{x}_k$ ▷ Direction of support
2: $\boldsymbol{s}_k \in S_{\mathcal{D}}(\boldsymbol{d}_k)$ ▷ Support (7)
3: **If** $g_{FW}(\boldsymbol{x}_k) \leq \epsilon$, **return** $f(\boldsymbol{x}_k)$ ▷ Duality gap (15)
4: $\widetilde{W}_{k+1} = W_k \cup \{\boldsymbol{s}_k\}$ ▷ Augment active-set
5: $\boldsymbol{x}_{k+1} = \text{proj}_{\text{conv}(\widetilde{W}_{k+1})}(\mathbf{0}_{\mathcal{C}})$ ▷ Project $\mathbf{0}_{\mathcal{C}}$ on conv$(\widetilde{W}_{k+1})$
6: $W_{k+1} = \{\boldsymbol{x}_{k+1}\}$ ▷ Update active-set

- We then minimize function $f(\boldsymbol{x}) = \|\boldsymbol{x}\|^2$ on $\widetilde{W}_{k+1}$ to obtain $\boldsymbol{x}_{k+1}$, corresponding to projecting $\mathbf{0}_{\mathcal{C}}$ onto $\widetilde{W}_{k+1}$: $\boldsymbol{x}_{k+1} = \text{proj}_{\text{conv}(\widetilde{W}_{k+1})}(\mathbf{0}_{\mathcal{C}})$[1] (line 5 in Alg. 3).
- We then have two cases, summarized in Alg 4:

↪ If $\boldsymbol{x}_{k+1} = \mathbf{0}_{\mathcal{C}}$, the algorithm is stopped. Thus, we have $\boldsymbol{x}^* = \mathbf{0}_{\mathcal{C}}$ and $d_{1,2} = 0$ in (3) (line 1 in Alg. 4).

↪ Otherwise, we construct W_{k+1} from $\widetilde{W}_{k+1}$. To do so, we retain only the minimal number of vectors in $\widetilde{W}_{k+1}$ needed to express $\boldsymbol{x}_{k+1}$ as a convex combination (line 2 in Alg. 4). Indeed, as $\mathbf{0}_{\mathcal{C}} \notin \widetilde{W}_{k+1}$, the projection $\boldsymbol{x}_{k+1}$ of $\mathbf{0}_{\mathcal{C}}$ on $\widetilde{W}_{k+1}$ necessarily lies on a face of $\widetilde{W}_{k+1}$, and can be expressed as a convex combination of the vectors composing this face. This ensures that W_{k+1} is necessarily of rank $r_{k+1} < \widetilde{r}_{k+1} \leq n+1$.

Through this discussion, it is clear that GJK is a particular case of Frank-Wolfe. More specifically, it is a sub-case of the fully-corrective Frank-Wolfe algorithm analyzed by Lacoste-Julien and Jaggi [21]. The strategy used by GJK to handle the active-set has proved to be very efficient in practice and renders the GJK algorithm state of the art for collision detection. In the next section, we propose to leverage the formulation of collision detection as a Frank-Wolfe sub-case to accelerate its convergence following the well-established Nesterov acceleration paradigm [29].

III. Accelerating Collision Detection

Gradient descent (GD) is the backbone of many convex optimization methods and relies solely on the gradient of the objective function. Second order methods [2] such as Newton methods have faster convergence rates than GD at the price of requiring the computation and the inversion of the Hessian quantities. Momentum methods have thus been introduced in the optimization literature to provide gradient-based methods with improved convergence rates without requiring costly Hessian evaluation. In this section we use recent work linking the *Nesterov acceleration* of GD to the FW algorithm to globally accelerate collision detection. This global acceleration

[1]The efficient projection onto simplexes in $\mathbb{R}^3$, named the *distance sub-algorithm* by Gilbert et al. [14], is thoroughly covered in [10, 39] and its robustness is improved in [28].

Algorithm 4 Fully-corrective FW using simplexes, applied to the distance computation problem (3). This algorithm is identical to GJK [14]

In Alg. 3, let $W_0 = \emptyset$ and replace line 6 by:

1: **If** $\boldsymbol{x}_{k+1} = \mathbf{0}_{\mathcal{C}}$, **return** 0
If the algorithm has not terminated, update $\widetilde{W}_{k+1}$ to retain only the smallest number of vectors needed to express $\boldsymbol{x}_{k+1}$:

2: $W_{k+1} = \{\boldsymbol{s}^1, ..., \boldsymbol{s}^r\}$ where $\boldsymbol{s}^1, ..., \boldsymbol{s}^r$ are the smallest number of vectors in $\widetilde{W}_{k+1}$ such that $\boldsymbol{x}_{k+1}$ is a convex combination of $\boldsymbol{s}^1, ..., \boldsymbol{s}^r$.

of collision detection is experimentally evaluated in Sec. IV on several benchmarks.

Nesterov acceleration for unconstrained optimization. We initially consider the following *unconstrained* minimization problem:

$$f(\boldsymbol{x}^*) = \min_{\boldsymbol{x} \in \mathbb{R}^n} f(\boldsymbol{x}), \tag{21}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is a convex and differentiable function. The vanilla gradient-descent algorithm consists of following the slope of f given by its gradient ∇f. The following scheme is applied iteratively until a given convergence criterion is met (e.g., $\|\nabla f(\boldsymbol{x}_k)\| < \epsilon$, with ϵ the desired precision):

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \nabla f(\boldsymbol{x}_k), \tag{22}$$

where $\boldsymbol{x}_k \in \mathbb{R}^n$ is the current iterate and $\alpha_k \in \mathbb{R}$ is the gradient step. This standard setting leads to a simple implementation with linear convergence rate ($O(1/k)$).

To go beyond this linear convergence regime, acceleration techniques have been devised in the optimization community to provide quadratic convergence rate ($O(1/k^2)$) or more [9], by relying on relatively cheap gradient evaluations. Among these gradient-descent acceleration techniques, the Nesterov acceleration [29] is one of the better studied and most popular in practice [9]. It is based on accumulating previously computed gradients in a *momentum* term $\boldsymbol{d}_k$ and using this momentum $\boldsymbol{d}_k$ to update the current iterate $\boldsymbol{x}_k$ as:

$$\boldsymbol{y}_k = \boldsymbol{x}_k + \delta_k \boldsymbol{d}_{k-1} \tag{23a}$$
$$\boldsymbol{d}_k = \delta_k \boldsymbol{d}_{k-1} + \alpha_k \nabla f(\boldsymbol{y}_k) \tag{23b}$$
$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{d}_k \tag{23c}$$

where $\delta_k \in \mathbb{R}$ is the momentum parameter and $\boldsymbol{y}_k \in \mathbb{R}^n$ is an intermediate quantity. The role of momentum $\boldsymbol{d}_k$ is to smooth the trajectory of iterates converging towards the optimum by geometrically averaging previously computed gradients. The δ_k momentum parameter is selected to prevent damping or overshooting of the iterate trajectory when going towards the optimal solution $\boldsymbol{x}^*$. Note that the current iterate $\boldsymbol{x}_k$ is extrapolated using the momentum term $\boldsymbol{d}_k$ to compute the intermediate vector $\boldsymbol{y}_k = \boldsymbol{x}_k + \delta_k \boldsymbol{d}_k$. The gradient is then computed at the vector $\boldsymbol{y}_k$. Computing the term $\boldsymbol{y}_k$ leads to an anticipatory behavior in similar spirit to extra-gradient methods [9].

Algorithm 6 Nesterov-accelerated Frank-Wolfe [22]

In Alg. 1 and Alg. 3, let $\boldsymbol{d}_{-1} = \boldsymbol{s}_{-1} = \boldsymbol{x}_0$, $\delta_k = \frac{k+1}{k+3}$ and replace line 1 by:

1: $\boldsymbol{y}_k = \delta_k \boldsymbol{x}_k + (1 - \delta_k)\boldsymbol{s}_{k-1}$
2: $\boldsymbol{d}_k = \delta_k \boldsymbol{d}_{k-1} + (1 - \delta_k)\nabla f(\boldsymbol{y}_k)$

The Frank-Wolfe algorithm and Nesterov acceleration. Recent works of Li et al. [22, 23] have proposed to adapt the Nesterov acceleration to the FW setting. We propose to leverage and adapt this FW acceleration scheme to the context of collision detection, by notably extending the FW formulation of collision detection previously developed in Sec. II.

In the original FW algorithm, the support vector at iteration k, $\boldsymbol{s}_k$, is computed in the direction of the gradient $\nabla f(\boldsymbol{x}_k)$ (line 1 in Alg. 1). In the Nesterov acceleration of FW proposed by Li et al. [22], the direction of support for computing $\boldsymbol{s}_k$ is instead defined by:

$$\boldsymbol{y}_k = \delta_k \boldsymbol{x}_k + (1 - \delta_k)\boldsymbol{s}_{k-1} \tag{24a}$$
$$\boldsymbol{d}_k = \delta_k \boldsymbol{d}_{k-1} + (1 - \delta_k)\nabla f(\boldsymbol{y}_k) \tag{24b}$$
$$\boldsymbol{s}_k = S_{\mathcal{D}}(\boldsymbol{d}_k), \tag{24c}$$

where $\boldsymbol{s}_{k-1}$ is the support computed at iteration $k-1$, $\delta_k = \frac{k+1}{k+3} \in [0, 1]$ is the momentum parameter and $S_{\mathcal{D}}$ is the support function as defined in (7). As in the Nesterov acceleration for GD, $\boldsymbol{y}_k$ is an intermediary vector used to evaluate the gradient $\nabla f(\boldsymbol{y}_k)$. To ensure $\boldsymbol{y}_k$ stays in $\mathcal{D}$, it is a convex combination of $\boldsymbol{x}_k$ and $\boldsymbol{s}_{k-1}$, both vectors of $\mathcal{D}$. The direction of support is then obtained by taking a convex combination of the previous support direction $\boldsymbol{d}_{k-1}$ and the gradient $\nabla f(\boldsymbol{y}_k)$.

Li et al. [22] have experimentally shown that this acceleration strategy leads to a better convergence rate of the FW algorithm when compared to the original FW algorithm. In the following, we explain how to adapt the Nesterov acceleration of FW to collision detection.

Adapting Nesterov fully-corrective Frank-Wolfe to distance computation. Preserving GJK's simplex strategy is crucial for collision detection as it greatly speeds up the vanilla FW algorithm. Therefore, we adapt (24) accordingly as:

$$\boldsymbol{y}_k = \delta_k \boldsymbol{x}_k + (1 - \delta_k)\boldsymbol{s}_{k-1} \tag{25a}$$
$$\boldsymbol{d}_k = \delta_k \boldsymbol{d}_{k-1} + (1 - \delta_k)\nabla f(\boldsymbol{y}_k) \tag{25b}$$
$$\boldsymbol{s}_k = S_{\mathcal{D}}(\boldsymbol{d}_k), \tag{25c}$$
$$\widetilde{W}_{k+1} = W_k \cup \{\boldsymbol{s}_k\}, \tag{25d}$$
$$\boldsymbol{x}_{k+1} = \text{proj}_{\text{conv}(\widetilde{W}_{k+1})}(\mathbf{0}_{\mathcal{C}}). \tag{25e}$$

These steps are also summarized in Alg. 7. The update of simplex W_{k+1} from $\widetilde{W}_{k+1}$ is then identical to the one described in Alg. 4. The original duality gap defined in Sec. II (Eq. 15) can no longer be used as a convergence

Algorithm 7 Nesterov-accelerated GJK

Let $\boldsymbol{x}_0 \in \mathcal{D}$, $W_0 = \emptyset$, $\boldsymbol{d}_{-1} = \boldsymbol{s}_{-1} = \boldsymbol{x}_0$, $\epsilon > 0$

For `k=0, 1, ...` **do**

1: $\delta_k = \frac{k+1}{k+3}$ ▷ Momentum parameter value

2: $\boldsymbol{y}_k = \delta_k \boldsymbol{x}_k + (1-\delta_k)\boldsymbol{s}_{k-1}$ ▷ Intermediary point (25a)

3: $\boldsymbol{d}_k = \delta_k \boldsymbol{d}_{k-1} + (1-\delta_k)\nabla f(\boldsymbol{y}_k)$ ▷ Support dir. (25b)

4: $\boldsymbol{s}_k \in S_{\mathcal{D}}(\boldsymbol{d}_k)$ ▷ Support (7)

5: **if** $g(\boldsymbol{x}_k) \leq \epsilon$ **then** ▷ Fixed-point condition (29)

6: **If** $\boldsymbol{d}_k = \boldsymbol{x}_k$ **, return** $f(\boldsymbol{x}_k)$ ▷ Algorithm terminates

7: $\boldsymbol{s}_k \in S_{\mathcal{D}}(\nabla f(\boldsymbol{x}_k))$ ▷ Compute $\boldsymbol{s}_k$ in dir. $\nabla f(\boldsymbol{x}_k)$

Replace line 3 by: $\boldsymbol{d}_k = \boldsymbol{x}_k$ **until termination.**

8: $\widetilde{W}_{k+1} = W_k \cup \{\boldsymbol{s}_k\}$ ▷ Augment active-set

9: $\boldsymbol{x}_{k+1} = \text{proj}_{\text{conv}(\widetilde{W}_{k+1})}(\mathbf{0}_{\mathcal{C}})$ ▷ Project $\mathbf{0}_{\mathcal{C}}$ on conv$(\widetilde{W}_{k+1})$

10: **If** $\boldsymbol{x}_{k+1} = \mathbf{0}_{\mathcal{C}}$, **return** 0

11: $W_{k+1} = \{\boldsymbol{s}^1, ..., \boldsymbol{s}^r\}$ where $\boldsymbol{s}^1, ..., \boldsymbol{s}^r$ are the smallest number of vectors in $\widetilde{W}_{k+1}$ such that $\boldsymbol{x}_{k+1}$ is a convex combination of $\boldsymbol{s}^1, ..., \boldsymbol{s}^r$.

criterion. Indeed, the following inequality:

$$\|\boldsymbol{x}_k - \boldsymbol{x}^*\|^2 \leq g_{\text{FW}}(\boldsymbol{x}_k) = 2\langle \boldsymbol{x}_k, \boldsymbol{x}_k - \boldsymbol{s}_k \rangle,\ \boldsymbol{s}_k \in S_{\mathcal{D}}(\boldsymbol{x}_k),$$

is no longer valid because the support vector $\boldsymbol{s}_k$ is no longer computed in the direction of the gradient $\nabla f(\boldsymbol{x}_k) = 2\boldsymbol{x}_k$. Next we will show that the original stopping criterion devised in Sec. II cannot be used and we need to derive a new one.

Stopping criterion. As the number of iteration k increases, $\delta_k \underset{k\to\infty}{\to} 1$ in (25). Therefore, $\boldsymbol{d}_k$ tends to be equal to $\boldsymbol{d}_{k-1}$ (25b) and thus $\boldsymbol{s}_k = \boldsymbol{s}_{k-1}$ (25c). As a consequence, augmenting W_k with $\boldsymbol{s}_k$ to construct $\widetilde{W}_{k+1}$ (see (25d)) and then projecting $\mathbf{0}_{\mathcal{C}}$ onto $\widetilde{W}_{k+1}$ (25e) will not result in any progress. Therefore, $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k$: the algorithm reaches a fixed point and is stuck on constant support direction $\boldsymbol{d}$.

In order to cope with this issue, we use the following strategy. Suppose $\boldsymbol{x}_k \neq \mathbf{0}_{\mathcal{C}}$. Since $\boldsymbol{x}_k = \text{proj}_{\text{conv}(W_k)}(\mathbf{0}_{\mathcal{C}})$ we have:

$$\forall \boldsymbol{s}^i \in W_k,\ \langle \boldsymbol{x}_k, \boldsymbol{x}_k - \boldsymbol{s}^i \rangle = 0. \tag{26}$$

After computing $\boldsymbol{s}_k \in S_{\mathcal{D}}(\boldsymbol{d}_k)$, if we have:

$$\langle \boldsymbol{x}_k, \boldsymbol{x}_k - \boldsymbol{s}_k \rangle \neq 0, \tag{27}$$

then $\boldsymbol{s}_k$ is not a linear combination of vectors in W_k. Therefore, augmenting W_k with $\boldsymbol{s}_k$ to obtain $\widetilde{W}_{k+1}$ and projecting $\mathbf{0}_{\mathcal{C}}$ onto conv$(\widetilde{W}_{k+1})$ to obtain $\boldsymbol{x}_{k+1}$ will result in the algorithm progressing toward the optimum $\boldsymbol{x}^*$. Suppose on the contrary that:

$$\langle \boldsymbol{x}_k, \boldsymbol{x}_k - \boldsymbol{s}_k \rangle = 0, \tag{28}$$

then $\boldsymbol{s}_k$ is a linear combination of vectors in W_k. Adding $\boldsymbol{s}_k$ to W_k will thus not result in any progress towards the optimum. As a consequence, Eq. (28) encompasses two cases:

- If the support direction $\boldsymbol{d}_k$ is aligned with $\nabla f(\boldsymbol{x}_k)$, Eq. (28), corresponding to $g_{\text{FW}}(\boldsymbol{x}_k) = 0$, matches the termination criteria of the distance computation problem and therefore we have reached the optimum leading to $\boldsymbol{x}_k = \boldsymbol{x}^*$.
- Otherwise, if $\boldsymbol{d}_k$ is not aligned with $\nabla f(\boldsymbol{x}_k)$, the algorithm cannot stop as a null duality gap is not met. The algorithm thus enters a cycle where it iterates until Eq. (28) does not hold. To cope with this undesired behavior we simply stop the Nesterov acceleration as soon as Eq. (28) is met and switch back to the non-accelerated version Alg. 4.

We thus define the function g such that for any $\boldsymbol{s}_k \in \mathcal{D}$:

$$g(\boldsymbol{x}_k) = 2\langle \boldsymbol{x}_k, \boldsymbol{x}_k - \boldsymbol{s}_k \rangle, \tag{29}$$

g is used in Alg. 7 as an optimality criterion ($g \leq \epsilon$) either for stopping the Nesterov acceleration in order to continue with the vanilla GJK, or as stopping criteria qualifying an optimal solution, in which case $g = g_{\text{FW}}$ and (17) holds. The entire algorithm is summarized in Alg. 7.

Nesterov acceleration for non-strictly convex shapes. Let us explain the effect of the Nesterov acceleration on the support direction update (25b) and distinguish between strictly convex and non-strictly convex $\mathcal{D}$:

- If $\mathcal{D}$ is strictly convex, any vector $\boldsymbol{s}$ belonging to the surface of $\mathcal{D}$ has a unique corresponding direction $\boldsymbol{d}$ such that $\boldsymbol{s} = S_{\mathcal{D}}(\boldsymbol{d})$. Here, we stress the fact that the support function $S_{\mathcal{D}}$ returns only *one* vector. Consequently, we have $\boldsymbol{d}_k \neq \boldsymbol{d}_{k-1}$ and therefore $\boldsymbol{s}_k \neq \boldsymbol{s}_{k-1}$. The fixed point condition (28) is thus not met unless $\delta_k = 1$ and Nesterov acceleration continues to be applied in Alg. 7. In practice, the algorithm runs until δ_k gets close to 1 or $\boldsymbol{x}_k$ gets close to $\mathbf{0}_{\mathcal{C}}$. The condition (28) is then satisfied as the algorithm starts to cycle. The Nesterov acceleration is thus removed and the algorithm runs until the convergence criteria is satisfied, guaranteed by the Frank-Wolfe algorithm.
- Otherwise, if $\mathcal{D}$ is non-strictly convex, multiple support directions $\{\boldsymbol{d}^1, ..., \boldsymbol{d}^m, ...\}$ can yield the same support vector $\boldsymbol{s} \in S_{\mathcal{D}}(\boldsymbol{d}^1) = ... = S_{\mathcal{D}}(\boldsymbol{d}^m) = ...$ etc. Consequently, it is possible to have $\boldsymbol{d}_{k-1} \neq \boldsymbol{d}_k$ and $\boldsymbol{s}_k = \boldsymbol{s}_{k-1}$. Therefore, even though δ_k is not close to 1, the fixed point condition (28) can be verified. The Nesterov acceleration is stopped, possibly prematurely.

The latter case is especially problematic when shapes $\mathcal{A}_1$ and $\mathcal{A}_2$ are in close-proximity, which is ultimately the type of collision problems which are commonly encountered in simulation or motion planning with contacts. In (25b), this is due to the norm of $\boldsymbol{d}_{k-1}$ being greatly predominant over the norm of $\boldsymbol{y}_k$ as k increases, $\|\boldsymbol{d}_{k-1}\| \gg \|\nabla f(\boldsymbol{y}_k)\|$. To prevent this phenomenon on non-strictly convex $\mathcal{D}$, we propose to replace (25b) by a simple heuristic which normalizes the gradient and momentum directions as follows:

$$\boldsymbol{d}_k = \delta_k \frac{\boldsymbol{d}_{k-1}}{\|\boldsymbol{d}_{k-1}\|} + (1-\delta_k)\frac{\nabla f(\boldsymbol{y}_k)}{\|\nabla f(\boldsymbol{y}_k)\|}, \tag{30}$$

summarized in Alg. 8. In Sec. IV, we experimentally prove this heuristic to significantly reduce the number of steps for distance computations for non-strictly convex shapes.

Algorithm 8 Normalize direction for non-strictly convex shapes

Replace line 3 in Alg. 7 by:

1: $\boldsymbol{d}_k = \delta_k \frac{\boldsymbol{d}_{k-1}}{\|\boldsymbol{d}_{k-1}\|} + (1-\delta_k)\frac{\nabla f(\boldsymbol{y}_k)}{\|\nabla f(\boldsymbol{y}_k)\|}$

IV. Experiments

In this section, we study the performance of Nesterov-accelerated GJK (Alg. 7) against the vanilla GJK (Alg. 4) algorithm. We use the HPP-FCL C++ library [32, 33] and its implementation of GJK as a starting point for the implementation of the proposed Nesterov-accelerated GJK (Alg. 7). To distinguish between pairs of strictly convex and non-strictly convex shapes, we build a first benchmark only composed of pairs of ellipsoids (strictly convex shapes) and a second benchmark using pairs of standard meshes (represented by their convex hulls) which are taken from the commonly-used ShapeNet dataset [6]. Finally, we empirically show that the simplex strategy used by GJK and our method (discussed in Sec. II) is crucial for efficient collision detection. We show that GJK and our method significantly outperform the original FW algorithm and the recent NESMINO [34] algorithm. Although it differs from FW algorithms, we include the NESMINO algorithm in this analysis as its projected-gradient descent procedure is accelerated using the classic Nesterov acceleration scheme [29]. The code to run the benchmarks is made freely available at: https://github.com/lmontaut/collision-detection-benchmark.

Shape datasets. In the ellipsoids benchmark, the ellipsoids are randomly generated by sampling positive-definite matrices. In total, we generate 1000 random pairs of ellipsoids. Given a pair of ellipsoids, we randomly sample relative poses between the two shapes such that both objects do not intersect. This then allows us to finely control the distance $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$ between the objects, enabling us to measure the influence of the separation distance on the performance of the studied algorithms. The values used for $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$ range from $-0.1\,\text{m}$ to $1\,\text{m}$. Negative values correspond to scenarii where the shapes are intersecting and $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$ is the norm of the separating vector. The separating vector is the vector of smallest norm needed to translate one of the two shapes such that the two shapes do not intersect. Therefore, for each pair of ellipsoids, 100 random relative poses are sampled such that the shapes do not intersect. For each relative pose, we translate the shapes along the axis given by their closest-points to study the impact of $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$. We then set $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$ to fixed values between $-0.1\,\text{m}$ to $1\,\text{m}$. On the other hand, the mesh dataset contains roughly 35k shapes but we only use 1.8k randomly selected shapes from 10 categories of the dataset to set-up the mesh benchmark. To generate the mesh benchmark, we sample $10\,$k random pairs of meshes and represent each shape by its convex hull. The resulting meshes contain between 10 and 5000 vertices. About 50% of meshes contain between 100 and 1000 vertices. The remaining shapes are equally distributed between less than 100 vertices or more than 1000 vertices. As in the ellipsoids benchmark, 100 random relative poses are sampled for each pair such that the shapes do not intersect and then set $\text{dist}(\mathcal{A}_1, \mathcal{A}_2)$ to fixed values between $-0.1\,\text{m}$ and $1\,\text{m}$. In both benchmarks, the characteristic sizes of the shapes range from a few centimeters up to a meter. Finally, for the distance computation problem, we select a convergence tolerance of $\epsilon = 10^{-8}$.

Metrics. In order to measure the performances of the Nesterov-accelerated GJK and the vanilla GJK algorithms, we measure the number of iterations $\mathtt{N}^{\mathtt{k}}$ to solve a given collision problem. For the mesh benchmark, we also measure the execution time $\mathtt{T}^{\mu}$ of both methods. To cope with CPU throttling, we solve generated collision problems 100 times. We then report the average of the 90% lowest computation times. Since both algorithms have identical iterations which differ only in the computation of the support direction, the number of iterations is preferred to study the performance of the algorithms; the execution time serves as a reference but can vary based on the implementation and the hardware used.

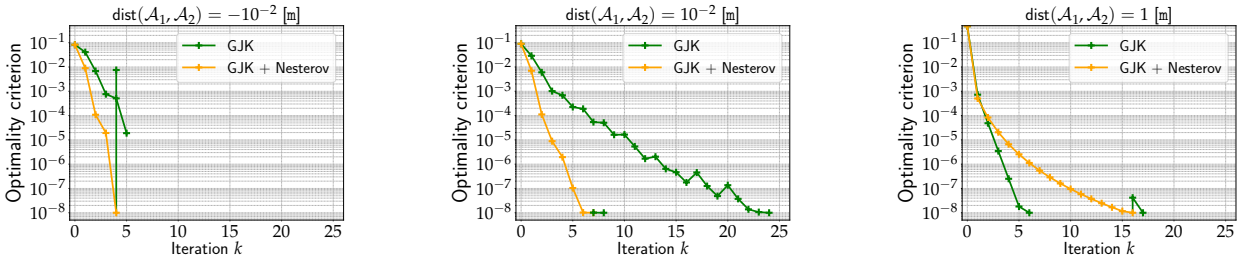
A. Strictly convex shapes: ellipsoids

Single collision pair. To qualitatively understand the effect of the Nesterov acceleration, we first study the evolution of the optimality criterion (17) in the case of distance computation on a single pair of ellipsoids (Fig. 4).

In Fig. 4a, the shapes are intersecting and both algorithms converge to $\boldsymbol{x}^* = \mathbf{0}_{\mathcal{C}}$. The optimality criterion (29) is not used by any of the two methods to terminate. Indeed, they stop when the projection onto the simplex $\widetilde{W}_k$ yields $\mathbf{0}_{\mathcal{C}}$ (line 10 in Alg. 7 and line 1 in Alg. 4). However, the criterion is still used in Nesterov-accelerated GJK to switch to GJK. Both methods converge rapidly. As explained in Sec. II, this is expected as the optimal solution $\boldsymbol{x}^* = \mathbf{0}_{\mathcal{C}}$ lies inside $\mathcal{D}$.

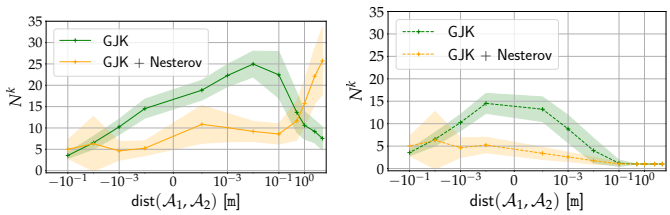
In Fig. 4b, the shapes are close together with a separating distance of $\text{dist}(\mathcal{A}_1, \mathcal{A}_2) = 0.01\,\text{m}$. Along with the previous intersecting case, this figure represents a typical scenario one could encounter in the narrow phase of the collision detection problem. Contrary to the intersecting case, the solution $\boldsymbol{x}^*$ now lies on the boundary of $\mathcal{D}$. The convergence of GJK is impaired due to the successive switches in support directions. By smoothing the change of support direction, the Nesterov acceleration requires less iterations and divides by more than two the number of iterations to reach the convergence criterion compared to GJK.

When shapes are distant (Fig. 4c), the problem is easier to solve for GJK. This is due to smaller angles between successive support directions which is a consequence of the large distance between $\mathbf{0}_{\mathcal{C}}$ and $\mathcal{D}$. In such a case, the Nesterov acceleration is now detrimental to the convergence of the algorithm as it induces too little successive changes in the support directions. This is a typical case where smoothing the gradient directions via the Nesterov acceleration does not help and takes more than triple the number of iterations for Nesterov-accelerated GJK to achieve convergence compared



(a) Intersecting ellipsoids. (b) Close-proximity ellipsoids. (c) Distant ellipsoids.

Fig. 4: **Convergence of GJK and Nesterov-accelerated GJK for ellipsoid collisions.** The y-axis measures the optimality criterion (29). For Nesterov-accelerated GJK, the switch from Nesterov acceleration to vanilla GJK is denoted by the switch in color from orange to green. In (a), the optimality criterion defined by (29) is not used by any of the two methods to terminate as $\mathbf{0}_{\mathcal{C}}$ lies inside $\mathcal{D}$ and both algorithms terminate when $\boldsymbol{x}_k = \mathbf{0}_{\mathcal{C}}$. It is however used by Nesterov-accelerated GJK to stop the Nesterov acceleration when a fixed-point is met, hence the sudden peak denoting the switch to GJK. Nesterov acceleration has beneficial effects in the intersecting (a) and close-proximity (b) setups which are the cases of interest, for instance, in the context of simulation and planning with contacts.



(a) Distance computation. (b) Collision detection.

Fig. 5: **Comparison of Nesterov-accelerated GJK and vanilla GJK on the ellipsoid benchmark.** The graphs show the number of iterations (y-axis) vs. the signed distance between the two shapes (x-axis). The curve shows the mean value over 100,000 random trials. The shaded region corresponds to the standard deviation. The Nesterov-accelerated GJK algorithm requires fewer iterations when the shapes are in close proximity.

to vanilla GJK. This behavior is however specific to ellipsoids and is not observed in meshes as shown below.

Fig. 4 highlights three different behaviors in the convergence of both algorithms: when the shapes are *intersecting*, in *close-proximity* and *distant*. Scenarii where shapes are intersecting or in close proximity are of greater importance in general, as they correspond to contexts which are evaluated during the narrow phase. On the opposite, distant scenarii are rejected by the broad phase, and other approximations of dist$(\mathcal{A}_1, \mathcal{A}_2)$ could be used if needed [10].

Statistical validation over the ellipsoids dataset. The convergence graphs in Fig. 4 directly depend on the pair of shapes considered. Therefore, to get a better statistical understanding of the performance of Nesterov-accelerated GJK and vanilla GJK, we focus on the ellipsoid benchmark.

Fig. 5a and Fig. 5b show the mean and standard-deviation of the number of iterations $\mathtt{N}^{\mathtt{k}}$ of each method for the distance computation and the Boolean collision checking problems, respectively. In the case of distance computation (Fig. 5a), the results confirm the trend shown in the convergence graphs in Fig. 4. When the shapes are shallowly intersecting, Nesterov-accelerated GJK converges with the same or even fewer number of iterations than vanilla GJK. The shallower the penetration, the more Nesterov accelerates over vanilla GJK. The irregularity in standard deviation at $-0.01\,\mathrm{m}$ is a critical zone for the momentum where the variance increases. This could be fixed thanks to a restart strategy [9] and will be analysed in a future work. When shapes are in close-proximity, the Nesterov acceleration of GJK significantly reduces the number of iterations compared to vanilla GJK. Finally, when shapes are distant, $1\,\mathrm{m} \leq \mathrm{dist}(\mathcal{A}_1, \mathcal{A}_2)$, Nesterov-accelerated GJK is detrimental to convergence on the distance computation problem. A similar pattern of speed-ups of the Nesterov-accelerated GJK over the vanilla GJK is shown for the collision detection problem in Fig. 5b.

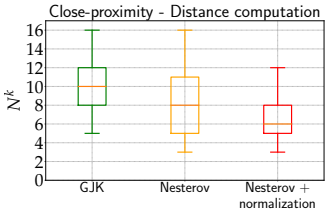
B. *Non-strictly convex shapes: meshes*

Effect of support direction normalization. For meshes, the importance of normalizing the support direction (see Eq. (30)) in the Nesterov-accelerated GJK is highlighted in Fig. 6. For both the distance computation and Boolean collision checking problems, it prevents the Nesterov acceleration from reaching a fixed-point too early and consequently it reduces the overall amount of iterations needed to converge. In the following, we thus focus only on Nesterov-accelerated GJK with support normalization and compare its performance against the vanilla GJK algorithm.

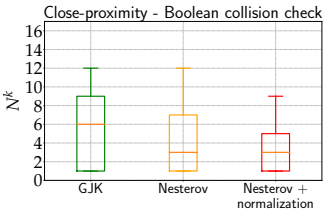
Statistical validation over the ShapeNet dataset. In Fig. 7 and Fig. 8, we report the number of iterations $\mathtt{N}^{\mathtt{k}}$ and execution time $\mathtt{T}^{\mu}$ for both Nesterov-accelerated and vanilla GJK. In Fig. 9, we report relative accelerations $\mathtt{N}^{\mathtt{k}}_{\mathrm{GJK}}/\mathtt{N}^{\mathtt{k}}_{\mathrm{Nesterov}}$ and $\mathtt{T}^{\mu}_{\mathrm{GJK}}/\mathtt{T}^{\mu}_{\mathrm{Nesterov}}$ of Nesterov-accelerated GJK compared to GJK on a given collision problem. These relative measures allow to analyze the effects of the two algorithms on exactly the same collision problems, which are not captured when using absolute values. Overall, Nesterov-accelerated GJK significantly reduces the number of iterations and execution time when compared to GJK in cases where shapes are shallowly intersecting or in close-proximity. The number of iterations and the execution times are reduced by a factor ranging from 1.5 up to 2. It is worth recalling at this stage, that when two shapes are relatively far from each other, any broad phase algorithm will automatically discard such pair. Only in a

		$N = 33$		$N = 141$		$N = 626$		$N = 2500$	
		GJK	**GJK + Nesterov**	**GJK**	**GJK + Nesterov**	**GJK**	**GJK + Nesterov**	**GJK**	**GJK + Nesterov**
	$\mathtt{T}_{\mathrm{D}}^{\mu}$	0.8 ± 0.3	0.8 ± 0.2	1.1 ± 0.4	$\mathbf{1.0 \pm 0.3}$	1.3 ± 0.4	$\mathbf{1.1 \pm 0.3}$	2.5 ± 0.6	$\mathbf{1.8 \pm 0.6}$
	$\mathtt{T}_{\mathrm{C}}^{\mu}$	0.6 ± 0.3	0.6 ± 0.3	0.8 ± 0.3	$\mathbf{0.7 \pm 0.3}$	1.1 ± 0.5	$\mathbf{0.8 \pm 0.4}$	1.9 ± 0.9	$\mathbf{1.4 \pm 0.7}$
	$\mathtt{T}_{\mathrm{D}}^{\mu}$			1.2 ± 0.3	$\mathbf{1.0 \pm 0.2}$	1.5 ± 0.4	$\mathbf{1.2 \pm 0.3}$	2.6 ± 0.7	$\mathbf{1.9 \pm 0.6}$
	$\mathtt{T}_{\mathrm{C}}^{\mu}$			0.9 ± 0.4	$\mathbf{0.7 \pm 0.3}$	1.2 ± 0.5	$\mathbf{0.9 \pm 0.4}$	2.1 ± 0.9	$\mathbf{1.4 \pm 0.7}$
	$\mathtt{T}_{\mathrm{D}}^{\mu}$					1.8 ± 0.6	$\mathbf{1.4 \pm 0.4}$	3.0 ± 0.8	$\mathbf{2.1 \pm 0.7}$
	$\mathtt{T}_{\mathrm{C}}^{\mu}$					1.4 ± 0.6	$\mathbf{1.0 \pm 0.5}$	2.5 ± 1.1	$\mathbf{1.6 \pm 0.8}$
	$\mathtt{T}_{\mathrm{D}}^{\mu}$							4.1 ± 1.1	$\mathbf{2.7 \pm 0.9}$
	$\mathtt{T}_{\mathrm{C}}^{\mu}$							3.3 ± 1.5	$\mathbf{2.2 \pm 1.1}$

TABLE I: **Computation times (μs) for distance computation ($\mathbf{T}_{\mathbf{D}}^{\mu}$) and Boolean collision checking ($\mathbf{T}_{\mathbf{C}}^{\mu}$) on the ShapeNet benchmark** for close-by or shallowly intersecting shapes where the distance between shapes is $-0.1\mathrm{m} \leq \mathrm{dist}(\mathcal{A}_1, \mathcal{A}_2) \leq 0.1\mathrm{m}$. N denotes the number of vertices for each mesh. For all depicted pairs, Nesterov-accelerated GJK always gives a better (or at least as good) timing for both collision computation and distance evaluation.



(a) Distance computation.

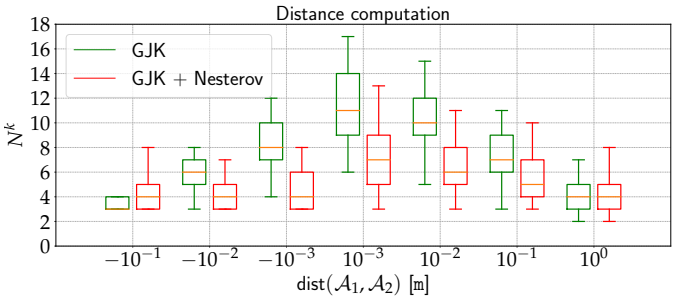


(b) Boolean collision check

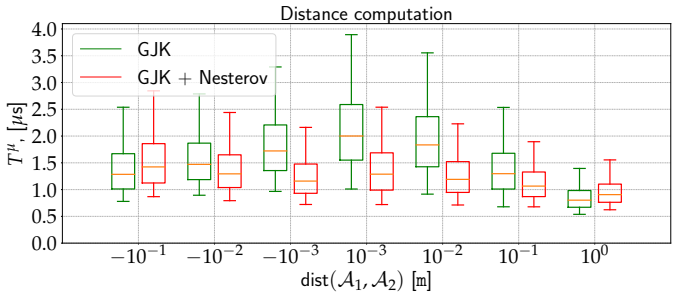
Fig. 6: **Importance of support direction normalization in Nesterov-accelerated GJK on the ShapeNet benchmark.** The graphs show the number of iterations $\mathtt{N}^{\mathtt{k}}$ (lower is better) for GJK and Nesterov-accelerated GJK with and without support direction normalization when the two shapes are in close-proximity: $0\,\mathrm{m} < \mathrm{dist}(\mathcal{A}_1, \mathcal{A}_2) \leq 0.1\,\mathrm{m}$. Using normalization of support direction is beneficial in Nesterov-accelerated GJK, reducing the overall number of iterations compared to GJK and non-normalized Nesterov-accelerated GJK.

small percentage of cases Nesterov-accelerated GJK is slower than GJK. Momentum restarts could help improve these rare cases and will be considered in future work.

In Table. I, we select 4 meshes with an increasing number of vertices to highlight the benefits of the Nesterov acceleration. For each pair, we report the mean and the standard deviation of the execution time for both distance computation and Boolean collision checking. We consider the challenging set-up of close-by or shallowly intersecting shapes in the range of separation distances $-0.1\,\mathrm{m} \leq \mathrm{dist}(\mathcal{A}_1, \mathcal{A}_2) \leq 0.1\,\mathrm{m}$. The lower mean and standard deviation show that Nesterov-accelerated GJK is faster than the vanilla GJK and reduces the spread of computation times across the different collision problems in this setting.



(a) Number of iterations $\mathtt{N}^{\mathtt{k}}$. Lower is better.

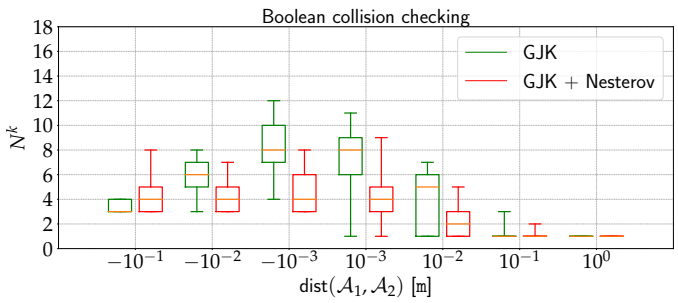


(b) Execution time $\mathtt{T}^{\mu}$. Lower is better.

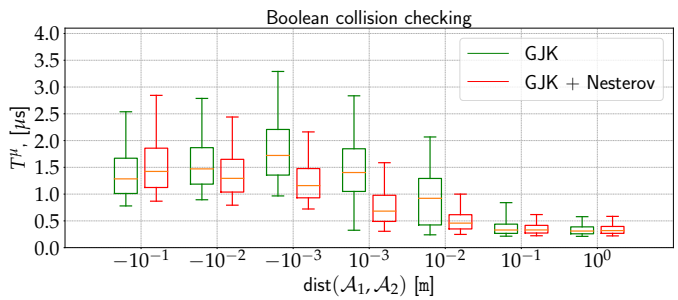
Fig. 7: **Distance computation on the ShapeNet benchmark.** The graphs show the number of iterations (a) and the execution time (b) for Nesterov-accelerated GJK with normalization (red) and vanilla GJK (green) for a range of distances (x-axis) between the shapes.

C. Importance of the simplex strategy in GJK

In the previous sections, we have experimentally shown the improvement of our method, Nesterov-accelerated GJK, over the vanilla GJK algorithm for collision problems

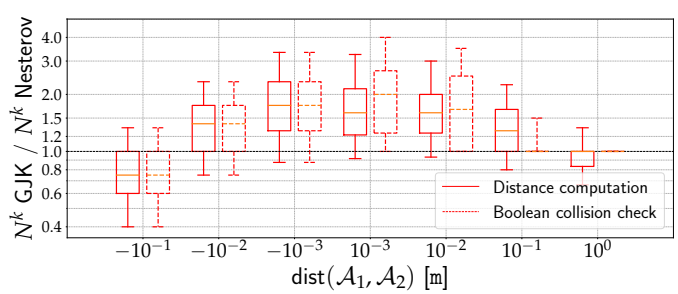


(a) Number of iterations N^{k}. Lower is better.

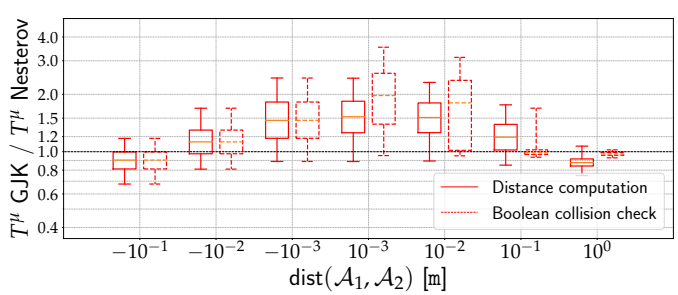


(b) Execution time T^{μ}. Lower is better.

Fig. 8: **Boolean collision check on the ShapeNet benchmark.** The graphs show the number of iterations (a) and the execution time (b) for the Nesterov-accelerated GJK with normalization (red) and vanilla GJK (green) for a range of distances (x-axis) between the shapes.



(a) Ratio of number iteration $\mathrm{N}^{\mathrm{k}}_{\mathrm{GJK}}/\mathrm{N}^{\mathrm{k}}_{\mathrm{Nesterov}}$. Higher is better.



(b) Ratio of execution time $\mathrm{T}^{\mu}_{\mathrm{GJK}}/\mathrm{T}^{\mu}_{\mathrm{Nesterov}}$. Higher is better.

Fig. 9: **Speed-ups on the ShapeNet benchmark.** The plots show ratios of the number of iterations (a) and execution times (b) of vanilla GJK and the Nesterov-accelerated GJK with normalization. Ratios over 1.0 show speed-ups of Nesterov-accelerated GJK over GJK.

which are important in practice, *i.e.* when collision pairs have not been filtered by the broad phase and are thus overlapping or in close-proximity. We conclude this section by demonstrating the importance of the simplex strategy used in GJK and our method when solving collision problems. To do so, we evaluate the performance of the Frank-Wolfe algorithm (Alg. 1), the recent NESMINO algorithm [34], GJK and our method on ellipsoids and cubes and report the results in Table. II. Altough the NESMINO algorithm is similar to projected-gradient descent and strongly differs from Frank-Wolfe like algorithms, it uses the classic Nesterov acceleration which makes it interesting to compare to our method. FW, GJK and our method stop when a tolerance of $\epsilon = 10^{-8}$ on the FW duality-gap is met. Therefore, to render the NESMINO algorithm comparable to the other considered methods, we run NESMINO until the distance between its solution and the solution found by GJK is less than $\sqrt{\epsilon} = 10^{-4}$.

In Table IIa, we consider 1000 collision problems between pairs of ellipsoids for each distance category (overlapping, close-proximity and distant). Shapes are in close-proximity when $0 \leq \mathrm{dist}(\mathcal{A}_1, \mathcal{A}_2) \leq 0.1\,\mathrm{m}$. In this first scenario, all algorithms have a comparable number of operations per iteration. Indeed, the projection operation used in NESMINO when the shapes are ellipsoids has the same complexity as the support operation used in the three other algorithms. Although GJK and our method also do a simplex projection at each iteration, this operation has about the same complexity as computing the support point. In the case of strictly-convex shapes such as ellipsoids, GJK and our method significantly outperform the FW and NESMINO algorithms. This is especially the case when the shapes are overlapping or in close-proximity where GJK or our method take 3 to 10 times less iterations compared to FW or NESMINO.

In Table IIb, we repeated the same experiments with collision pairs of cubes. Since cubes are polytopes with a small number of vertices, GJK and our method only take a few iterations to reach a tolerance of $\epsilon = 10^{-8}$. However, because cubes are non-strictly convex shapes, the convergence of the FW algorithm is $O(1/\epsilon)$, *i.e.* it takes on the order of $1/\epsilon$ iterations to reach a FW duality-gap of ϵ. The NESMINO algorithm takes less iterations than FW but more than 100 times more iterations than GJK and our method. In the specific case of polytopes, the NESMINO algorithm is also much more costly per iteration than FW, GJK or our method, as it replaces the computation of support points by much more costly projections on the original polytopes.

V. Conclusion

In this work, we have first established that the well-known GJK algorithm can be understood as a variant of the Frank-Wolfe method, well studied within the convex optimization community, and more precisely GJK can be identified as a sub-case of fully-corrective Frank-Wolfe. Subsequently, this connection has enabled us to accelerate the GJK algorithm in the sense of Nesterov acceleration, by

	FW (Alg. 1)	Nesmino [34]	GJK (Alg. 4)	Ours (Alg. 7)
Overlapping	73 ± 62	50 ± 18	$\mathbf{6 \pm 2}$	$\mathbf{6 \pm 3}$
Close-proximity	48 ± 42	74 ± 29	16 ± 5	$\mathbf{7 \pm 2}$
Distant	$\mathbf{4 \pm 1}$	18 ± 2	$\mathbf{4 \pm 1}$	13 ± 4

(a) Distance computation - Ellipsoids.

	FW (Alg. 1)	Nesmino [34]	GJK (Alg. 4)	Ours (Alg. 7)
Overlapping	$5.4\text{k} \pm 4.8\text{k}$	922 ± 244	6 ± 1	$\mathbf{5 \pm 1}$
Close-proximity	$14.3\text{k} \pm 9.7\text{k}$	828 ± 225	5 ± 1	$\mathbf{4 \pm 1}$
Distant	$13.1\text{k} \pm 13.5\text{k}$	623 ± 219	4 ± 1	$\mathbf{3 \pm 1}$

(b) Distance computation - Cubes.

TABLE II: **Number of iterations for distance computation between ellipsoids (a) and between cubes (b).** For each distance category (overlapping, close-proximity and distant), we report the mean and standard deviation of the number of iterations. Shapes are in close-proximity when $0 \leq \text{dist}(\mathcal{A}_1, \mathcal{A}_2) \leq 0.1\,\text{m}$.

adapting recent contributions on the application of Nesterov acceleration to the context of Frank-Wolfe. Through extensive benchmarks, we have shown that this acceleration is beneficial for both collision detection and distance computation settings for scenarios where shapes are intersecting or in close proximity, accelerating collision detection by up to a factor of 2. Interestingly, these two scenarios notably encompass the generic contexts of planning and control as well as physical simulation, which are essential areas of modern robotics. Therefore, although the proposed acceleration corresponds to improvements of GJK's execution time on the order of few microseconds, modern robotics applications may solve millions to billions of collision problems *e.g.* when learning a policy with RL [31]. The Nesterov acceleration of GJK is already included in the HPP-FCL library [33], notably used by the HPP framework [26] for motion planning and the Pinocchio framework [5] dedicated to simulation and modelling. One can expect this work to be largely adopted in the current available GJK implementations, as it only requires minor algorithmic changes. This work should benefit a large audience within robotics (e.g., simulation, planning, control) and beyond by addressing issues shared by other communities, including computer graphics and computational geometry.

Acknowledgments

The authors would like to warmly thank Francis Bach for fruitful discussions on Min-Norm Point algorithms. This work was partly supported by the European Regional Development Fund under the project IMPACT (reg. no. CZ.02.1.01/0.0/0.0/15 003/0000468), by the French government under management of Agence Nationale de la Recherche as part of the "Investissements d'avenir" program, reference ANR-19-P3IA-0001 (PRAIRIE 3IA Institute) and the Louis Vuitton ENS Chair on Artificial Intelligence.

References

[1] Francis Bach et al. Learning with Submodular Functions: A Convex Optimization Perspective. *Foundations and Trends in Machine Learning*, 2013.

[2] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[3] S. Cameron. A Comparison of Two Fast Algorithms for Computing the Distance between Convex Polyhedra. *IEEE Transactions on Robotics and Automation*, 1997.

[4] C. Caratheodory. Uber den Variabilitatsbereich der Koeffizienten von Potenzreihen, die gegebene Werte nicht annehmen. *Mathematische Annalen*, 1907.

[5] Justin Carpentier, Guilhem Saurel, Gabriele Buondonno, Joseph Mirabel, Florent Lamiraux, Olivier Stasse, and Nicolas Mansard. The Pinocchio C++ library – A fast and flexible implementation of rigid body dynamics algorithms and their analytical derivatives. In *IEEE International Symposium on System Integrations (SII)*, 2019.

[6] Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. ShapeNet: An Information-Rich 3D Model Repository. Technical Report arXiv:1512.03012 [cs.GR], Stanford University, Princeton University, Toyota Technological Institute at Chicago, 2015.

[7] Erwin Coumans and Yunfei Bai. PyBullet, a Python module for physics simulation for games, robotics and machine learning. http://pybullet.org, 2016–2021.

[8] George Dantzig. *Linear programming and extensions*. Princeton university press, 2016.

[9] Alexandre d'Aspremont, Damien Scieur, and Adrien Taylor. Acceleration methods. *arXiv:2101.09545*, 2021.

[10] Christer Ericson. *Real-Time Collision Detection*. The Morgan Kaufmann Series, 2004.

[11] Marguerite Frank and Philip Wolfe. An algorithm for quadratic programming. *Naval Research Logistics Quarterly*, 1956.

[12] Dan Garber and Elad Hazan. Playing Non-linear Games with Linear Oracles. In *IEEE 54th Annual Symposium on Foundations of Computer Science*, 2013.

[13] E. Gilbert and D. Johnson. Distance Functions and Their Application to Robot Path Planning in the Presence of Obstacles. *IEEE Journal on Robotics and Automation*, 1985.

[14] E.G. Gilbert, D.W. Johnson, and S.S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal on Robotics and Automation*, 1988.

[15] Elmer G. Gilbert. An Iterative Procedure for Computing the Minimum of a Quadratic Form on a Convex Set. *SIAM Journal on Control*, 1966.

[16] Jacques Guélat and Patrice Marcotte. Some Comments on Wolfe's 'Away Step'. *Mathematical Programming*, 1986.

[17] Charles A. Holloway. An Extension of the Frank and Wolfe Method of Feasible Directions. *Mathematical Programming*, 1974.

[18] Martin Jaggi. Revisiting Frank-Wolfe: Projection-free sparse convex optimization. *Proceedings of the 30th international conference on machine learning*, 2013.

[19] Thomas Kerdreux, Alexandre d'Aspremont, and Sebastian Pokutta. Restarting Frank-Wolfe. In *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics*, 2019.

[20] Thomas Kerdreux, Alexandre d'Aspremont, and Sebastian Pokutta. Projection-Free Optimization on Uniformly Convex Sets. *arXiv:2004.11053*, 2020.

[21] Simon Lacoste-Julien and Martin Jaggi. On the Global Linear Convergence of Frank-Wolfe Optimization Variants. *arXiv:1511.05932*, 2015.

[22] Bingcong Li, Mario Coutino, Georgios B. Giannakis, and Geert Leus. A Momentum-Guided Frank-Wolfe Algorithm. *IEEE Transactions on Signal Processing*, 2021.

[23] Bingcong Li, Alireza Sadeghi, and Georgios B. Giannakis. Heavy Ball Momentum for Conditional Gradient. *arXiv:2110.04243 [cs, math]*, 2021.

[24] M.C. Lin and J.F. Canny. A Fast Algorithm for Incremental Distance Calculation. In *International Conference on Robotics and Automation*, 1991.

[25] Khaled Mamou and Faouzi Ghorbel. A Simple and Efficient Approach for 3D Mesh Approximate Convex Decomposition. In *The 16th IEEE International Conference on Image Processing*, 2009.

[26] Joseph Mirabel, Steve Tonneau, Pierre Fernbach, Anna-Kaarina Seppälä, Mylene Campana, Nicolas Mansard, and Florent Lamiraux. HPP: A new software for constrained motion planning. In *International Conference on Intelligent Robots and Systems*, 2016.

[27] Brian Mirtich. V-Clip: Fast and Robust Polyhedral Collision Detection. *ACM Transactions on Graphics*, 1998.

[28] Mattia Montanari, Nik Petrinic, and Ettore Barbieri. Improving the GJK Algorithm for Faster and More Reliable Distance Queries Between Convex Objects. *ACM Transactions on Graphics*, 2017.

[29] Y. Nesterov. A method for solving the convex programming problem with convergence rate $O(1/k^2)$. *Proceedings of the USSR Academy of Sciences*, 1983.

[30] Nvidia. Persistent Contact Manifold, 2008.

[31] OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. Solving Rubik's Cube with a Robot Hand. *arXiv:1910.07113*, 2019.

[32] Jia Pan, Sachin Chitta, and Dinesh Manocha. FCL: A General Purpose Library for Collision and Proximity Queries. In *2012 IEEE International Conference on Robotics and Automation*. IEEE, 2012.

[33] Jia Pan, Sachin Chitta, Dinesh Manocha, Florent Lamiraux, Joseph Mirabel, Justin Carpentier, et al. HPP-FCL: an extension of the Flexible Collision Library. https://github.com/humanoid-path-planner/hpp-fcl, 2015–2022.

[34] Xiaolong Qin and Nguyen Thai An. Smoothing Algorithms for Computing the Projection onto a Minkowski Sum of Convex Sets. *Computational Optimization and Applications*, 2019.

[35] John Schulman, Yan Duan, Jonathan Ho, Alex Lee, Ibrahim Awwal, Henry Bradlow, Jia Pan, Sachin Patil, Ken Goldberg, and Pieter Abbeel. Motion planning with sequential convex optimization and convex collision checking. *The International Journal of Robotics Research*, 2014.

[36] Olivier Stasse, Adrien Escande, Nicolas Mansard, Sylvain Miossec, Paul Evrard, and Abderrahmane Kheddar. Real-Time (Self)-Collision Avoidance Task on a Hrp-2 Humanoid Robot. In *2008 IEEE International Conference on Robotics and Automation*, Pasadena, CA, USA, 2008. IEEE.

[37] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *International Conference on Intelligent Robots and Systems*, 2012.

[38] Marc A Toussaint, Kelsey Rebecca Allen, Kevin A Smith, and Joshua B Tenenbaum. Differentiable physics and stable modes for tool-use and manipulation planning. *Robotics: Science and Systems*, 2018.

[39] Gino Van den Bergen. A Fast and Robust GJK Implementation for Collision Detection of Convex Objects. *Journal of Graphics Tools*, 1999.

[40] Gino Van den Bergen. *Collision Detection in Interactive 3D Environments*. Taylor & Francis, 2003.

[41] Philip Wolfe. Finding the Nearest Point in A Polytope. *Mathematical Programming*, 1976.